System Adaptation, Dynamic Communities and Gypsy Agents

by

Adrian John Ryan

Dissertation submitted for fulfillment of
the requirements for the degree of

Doctor of Philosophy

at the Faculty of Information Technology,
Monash University

December, 2005

My Thoughts, My Thesis, Myself,

Dedicated to Fern

System Adaptation, Dynamic Communities and Gypsy Agents

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Adrian John Ryan
December 12, 2005

System Adaptation, Dynamic Communities and Gypsy Agents

Adrian John Ryan
Monash University, 2005

Abstract

This thesis addresses issues regarding updating of software systems in a distributed and mobile environment. Existing research has seen that updatable systems are able to alter safely their runtime code, and that, distributed cooperation allows components of systems to work together by sharing data via remotely accessible instructions. The combination of these areas could allow systems to alter their runtime code based on the resources supplied by other systems within the same network. We investigate a proximity-based approach which will allow systems within the same vicinity to share code and evolve based on the code of other neighbouring systems. Such a technique would see systems cooperating by forming localised communities in which the software resources of all systems are exchanged and used for adaptation.

Current techniques, such as autonomic and adaptive systems, allow systems to cooperate in a similar manner, however, they prove to be limited to specific devices or supply a restricted level of system cooperation. While other techniques, such as context aware systems, are able to alter behaviour in accordance with their current surrounding, this is limited to predetermined possibilities. As such, there is no current technique which allows systems to alter their current state based on the other systems within the same locality.

This thesis takes as its starting point the problem of updating software in the field. It presents novel techniques which also lead to novel applications. Building on this, we explore the idea of communities formed by collections of mobile and stationary ubiquitous devices, and examine how devices within these communities can interact and evolve. During this exploration, we assess the suitability of current system cooperation techniques for community generation and identify the attributes of a dynamic community. We find that a localised dynamic community would allow heterogeneous systems to alter dynamically their state in accordance with other systems in the same community. However, this assessment suggests

that, although, some current techniques are seemingly suitable for such a level of system cooperation, none incorporate all the required aspects that allow localised dynamic communities to be established.

To overcome these short comings, a new middleware framework, Dynamic Update Projecture Environment (DUPE), is proposed. This framework is designed for heterogeneous devices using Java's interoperability capabilities. DUPE takes into account the requirements of localised dynamic communities and provides a new means of flexible system cooperation, applicable to all system types. The internal structure of the DUPE framework is designed such that it is applicable to the many unique dynamic updating techniques for Java applications that we identify during initial discussions. However, as a result of the heterogeneity of the systems likely to be participating in dynamic communities, the specification of DUPE's cooperation and system resource sharing is tightly constrained. The research details how the framework allows systems to interact within localised dynamic communities, and argues that this can be used as a viable new technique in system evolution via software transfer.

An evaluation of the framework is provided through a series of technical and analytical tests, and the benefits the framework provides as a system evolution mechanism are presented using epidemiology modeling.

The principal contributions of this thesis are that it proposes and demonstrates that a middleware framework can be used to dynamically update heterogeneous systems, and that by this means location-based communities can be formed that can adapt and evolve, and in turn affect, physically mobile agents.

Acknowledgements

"Life is what happens while your busy making other plans"

— *John Lennon ("Beautiful Boy, (Darling Boy)", 1980)*

There are many people who have helped during the development of this thesis, both academically and emotionally.

I would like to thank my supervisor Jan Newmarch. Along with all the guidance you provided me, I am particularly grateful that you continuously encouraged me to pursue any new, or alternative research ideas that arose during my candidature. I also would like to thank my work colleagues at Monash for providing me with invaluable experiences in teaching, and for creating such a pleasant studying environment. Special thanks to Judith Morgan for providing me with many suggestions throughout the thesis production. Your help was greatly appreciated. I thank my fellow research students, particularly those from Jan's research group, for continually challenging me academically and keeping our study area an enjoyable place to be. I wish you all the best of luck in your individual endeavors.

I thank my Mum and Dad for their continual support and positive attitude toward my work. And, I thank my brothers, Justin and David, for keeping me grounded in life while at the same time showing enormous encouragement in my pursuits. Most importantly, I would like to express my thanks to my wonderful fiancee, Fern. Thank you for all the love and support that you have provided me; I hope you know that it never went unnoticed. You were the absolute driving force behind my work, and without you this thesis would never have been completed.

Adrian

Monash University
December 2005

Publications

Publications arising from this thesis include:

A. Ryan and J. Newmarch (2005), An Architecture for Component Evolution, *in the Proceedings of the Consumer, Communications and Networking Conference (CCNC 2005)*, Las Vegas, NV, US, January 2005.

A. Ryan and J. Newmarch (2003), A Dynamic, Discovery Based, Remote Class Loading Structure, *in the Proceedings of the 7th Annual IASTED Conference on Software Engineering and Applications (SEA 2003)*, Marina Del Ray, CA, US, November 2003.

Permanent Address: Faculty of Information Technology
Monash University
Australia

This dissertation was typeset with LaTeX 2$_\varepsilon$[1] by Adrian Ryan.

[1] LaTeX 2$_\varepsilon$ is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Glenn Maughan and modified by Dean Thompson and David Squire of Monash University.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The phenomenal speed of the emergence of mobile systems has resulted in the research associated with system cooperation and usability increasing in importance. Old concepts have re-emerged and new ones have arisen. Included in this is a renewed interest in dynamic system updates. The current research in this area has successfully enabled systems to safely update their runtime code state [51]. At the same time, another technological capability that has progressed is distributed system cooperation. This area of work provides remote systems with the ability to simply share resources. The nature of this is such that, systems, including mobile systems, in the same location can communicate with each other. The benefits from a combination of these areas, dynamic system updates and distributed system cooperation, are yet to be fully established. Of particular interest is the effect that such a combination may have on mobile system cooperation.

A further area of interest, is the capabilities that have been provided by the standardisation of programming languages, such as Java. These languages provide heterogeneous systems with a common medium. In fact, Java has been widely applied in advanced distributed cooperation; examples include Jini [79], JXTA [17] and RMI [105]. However, it has not been associated with updatable systems and distributed cooperation. This then presents the possibility that heterogeneous systems, in the same proximity, can update their execution according to each other's capabilities. This may be achievable using a distributed association of code constructs. Of significant interest, is the application of this concept to mobile systems. To discover any benefits from this, we need to look into how,

and where mobile systems are used.

It is inevitable that mobile systems, as a result of their prevalence in society, will be found operating in the same location. Such a congregation has many similarities to the everyday scenario of communities, except that all community members are systems. If given the ability, these systems are in the perfect situation to dynamically adapt to each other. This ability may be provided through remotely shared code constructs. If this occurs, then the community members obtain the ability to alter their execution by using code from other members. This is similar to how people learn from each other. Moreover, the mobile members in this situation are likely to move among different communities. The result of this is that they present and take components with them. This entire concept is new. The term that we apply to this type of a community is 'a dynamic community', and the term we apply to the mobile members of a dynamic community is 'physical mobile agents', or, more commonly, 'gypsy agents'[1]. The benefits that these concepts may provide to systems have not, as yet, been analysed.

It may be argued that this concept in system cooperation is similar to that presented in autonomous systems [88], adaptable software [10, 71] and context aware applications [31]. However, these techniques are either limited to specific devices, have a restricted level of system cooperation or are limited to predetermined execution changes. Therefore, it is evident that what is missing is a standardised flexible means of system adaptation.

From this arises the concept that if heterogeneous systems execute in any dynamically updatable JVM, and share Java code resources in a dynamic community scenario, they can then dynamically adapt their runtime state based on their current location. If so, as a result of system differences, a middleware framework would best facilitate the community cooperation language. Moreover, the mobile systems using this technique would present a new way to transfer code components and facilitate system evolution.

In this thesis, in response to the possibilities presented in the previous discussion, we present our argument to support our design of the Dynamic Updatable Projecture Environment (DUPE) framework for dynamic community cooperation.

[1]Others have used the term 'gypsy' within the area of mobile agents [67], however, this has a different meaning to that used in this thesis.

And, paying particular attention to mobile systems, we explore the resultant benefits of this concept and present its contributions to the field.

1.1.1 Scope of This Thesis

Our research objective is the design of the DUPE framework. The framework will provide systems with the ability to dynamically alter their runtime state as a result of their interaction in a dynamic community. Moreover, as a result of their continual movements, mobile systems bring many unique characteristics to this concept. We, therefore, target the framework design at mobile systems.

With this objective motivating our research, we pursue the following core goals:

1. The first goal is to analyse the areas of importance to the establishment of dynamic communities. We identify and discuss, in depth, the research areas of dynamic updating and distributed cooperation. In Chapters 2 and 3 respectively, both areas are analysed from the point of view of mobile systems. In Chapter 4, the concepts of dynamic updating and distributed cooperation are utilised to establish the attributes and requirements of dynamic communities.

2. The second goal is to determine the best technique for dynamic community interaction in order to design the communication language aspects of the DUPE framework. There are two steps in the construction of the DUPE framework. Firstly, we establish the core components of the framework and its association, as a middleware, with its target application. This is presented in Chapter 5. Secondly, we clearly identify the communication language for system cooperation within dynamic communities, and apply it to the framework. This second step in the construction of the framework is realised in Chapter 6, with elements specific to security finalised in Chapter 7.

3. The third goal is to analyse the benefits of the DUPE framework, particularly for mobile systems. In Chapters 8 to 10 we present the details of three framework implementations which aid our analysis. Following this, in Chapters 11 and 12, we look to discover the benefits of the framework, using our implementations, and determine other contributions to the field. This analysis will conclude that dynamic communities, and its cooperation technique, provides systems with a flexible means of adaptation that

is not available through other techniques. Other additional benefits are also identified as a result of such community interaction; for example, new techniques in software component transfer and evolution.

These research aims have been achieved and consequently we present the following as substantial, original contributions to the field.

- We identify how dynamic updates are applied to systems. This analysis can be used to determine an appropriate means of dynamic updating for heterogeneous systems. This contribution is appropriate to dynamic updating in distributed applications.
- We analyse distributed cooperation techniques from the perspective of heterogeneous systems. This analysis determines how a distribution technique may be used for location based community cooperation. This contribution is appropriate to all systems, including mobile systems.
- We identify the attributes of a dynamic community. This included differentiating dynamic community interaction from community interaction and the interaction of distributed cooperation techniques. The requirements of a dynamic community were clearly detailed. This contribution provides a basis concept for this work.
- We analyse the current research techniques that may be use for generating dynamic communities. We clearly detail the limitations of each technique's deign, and identify the additions, or changes, required for its the use in dynamic communities. This contribution can be used to determine the technique that is most appropriate for allowing heterogeneous systems to establish dynamic communities.
- We design a framework, termed the DUPE framework, that meets the requirements of a dynamic community. The DUPE framework is designed for heterogeneous systems. The DUPE community language is clearly specified using a series of Java interfaces and classes. This provides all DUPE implementations with a common communication medium. This contribution is significant, as it is a result of the analytical contributions of this work. The DUPE framework is extensively tested using three implementations. This includes the testing of probable scenarios present in DUPE communities. This work confirms our claims on the benefits of dynamic community interaction.

- We demonstrate that heterogeneous systems can cooperate in a community scenario by sharing code components. This contribution is significant, as it is a new concept in system cooperation that is applicable to heterogeneous systems. This contribution also relates to the previous contribution.
- We conceptualise physical mobile agents, termed gypsy agents. Gypsy agents provide a new means of transferring system components. This contribution is a result of the combined features of mobile systems interaction in dynamic communities. The contribution is appropriate to all scenarios displaying these attributes.
- We develop models on the movements of gypsy agents. These models demonstrate the usefulness of gypsy agents as a software transfer mechanism. In doing so, we will develop a relationship to epidemic models, and provided new ways in which such analysis can be shown. This contribution is appropriate to all gypsy agent techniques. Moreover, the development of the models on the movement of gypsy agents clearly shows the diversity of the contributions of the entire thesis.

1.2 Thesis Overview

The structure of the thesis is as follows:

Chapter 2 explores dynamic systems. We initially detail the general approaches to dynamic system updates, we then discuss those that related to the Java Virtual Machine (JVM) and mobile system. During this analysis, we provide our rationale for why the DUPE framework is designed for Java applications.

Chapter 3 discusses distributed cooperation. During this discussion, we focus on discovery based distribution techniques, assessing their advantages from the perspective of remote heterogeneous systems. In this chapter, we determine the most appropriate distribution technique for the underlining communication structure for the DUPE framework.

Chapter 4 combines the conclusions from Chapters 2 and 3 and introduces the concept of dynamic communities. This chapter outlines a core contribution of this thesis. We discuss how dynamic communities are established and what systems can achieve as a result of dynamic community interaction. This discussion allows

us to determine the benefits that a dynamic community will provide to its interacting members. Furthermore, in this chapter, we conceptualise the movements of adaptable mobile systems as beneficial, and identify them as physical mobile agents, which we term gypsy agents. In the following chapters this concept will be seen to be a main contribution of this thesis

Chapter 5 introduces our design of the DUPE framework. In the chapter we provide the internal construction of the framework's design. This includes the framework's association with a target application and JVM. The design is provided in enough detail for implementation. In this chapter, we also identify a dynamic community, resulting from the DUPE framework, as a DUPE community. This term and concept is important for the remainder of the thesis.

Chapter 6 details the framework's DUPE community interaction design, including the aspects relevant to gypsy agents. However, as several sections of community interaction are also sections of DUPE's internal construction, elements of DUPE community interaction are described in Chapter 5. These elements of the design are referred to where necessary. The design of the framework is concluded by defining the requirements of a compatible DUPE implementation. To finalise this and the previous chapters, we identify and discuss several contributions to the field of dynamic communities.

Chapter 7 assesses the safety elements of DUPE community interaction. In this chapter, we analyse all the security and adaptation concerns that arise during the discussions of the previous chapters and we provide solutions to the problems that have been identified. These solutions are then included in the DUPE framework specification[2]. This chapter concludes our discussions on the background and the design of the DUPE framework.

Chapters 8, 9 and 10 provide three implementations of the DUPE framework; these are DUPE Lite, DUPE 5.0 and DUPE JPDA respectively. The DUPE Lite implementation is a limited compatible DUPE middleware. This implementation is designed for a standard JVM. As a consequence of this, no runtime dynamic adaptation is available. However, we include this implementation to show the flexibility of the DUPE community language.

[2]The complete DUPE framework specification is provided in Appendix A.

Our second implementation, DUPE 5.0, is a fully compatible DUPE middleware. This implementation is designed using the J2SE 5.0 JVM, as it is capable of achieving runtime dynamic code updates. However, as a result of the limitations of the targeted JVM, there are some limitations on the code components that can be used by this implementation. These limitations, however, are not detrimental to its cooperation within a DUPE community.

Our third implementation, DUPE JPDA, is the most complete of our implementations. It contains advanced security features and flexible code component usage.

All our implementations have a common design structure, particularly in their DUPE community communication elements. These sections of the implementations will be discussed in the early implementation chapters, and will not be re-introduced in the latter chapters.

Chapter 11 provides an evaluation of the DUPE framework based on our DUPE implementations. We present the results of both technical measurement and scenario analysis tests. The tests are designed to determine the usability of DUPE communities and to measure any overhead that interaction in a DUPE community is likely to put on an application. The results provided in this chapter show that the contributions of the thesis, as described in earlier chapters, are relevant.

In Chapter 12 we provide an analysis to determine the benefits of gypsy agent movements. To do so, we make use of mathematical modeling. This is provided to indicate the contributions that this concept makes to the field. The format of this chapter differs from that of our previous chapters, as it is an assessment of a separate conceptual theory. However, as the theory is directly associated to a contribution that derives from the DUPE framework, it is necessary and relevant to the conclusions presented in the thesis.

Chapter 13 provides our final conclusions and proposes new ideas and concepts for future work.

Chapter 2

Dynamic Systems

2.1 Introduction

The focus of this chapter is to provide a background into the concept of dynamic system alteration. Our aim is to analyse dynamic updating techniques in order to determine which techniques may be applicable in general, and in particular, to mobile systems. We detail the different means of enabling dynamic updates within systems as this ability forms the core aspect of the DUPE framework. We will demonstrate that the framework can be achieved through the use of several techniques, enabling us, in turn, to achieve our goals for the framework.

We begin by discussing dynamic updating techniques applied to different programming languages in order to illustrate the many different ways in which dynamic code alteration is achieved. After analysing the different approaches to dynamic updating we discuss their use within mobile systems. Following this discussion we then focus more specifically on Java techniques, analysing the different means that allow dynamic alterations with the Java Virtual Machine (JVM). In the final section of the chapter we conclude from the analysis which techniques are best suited for implementation using the JVM.

2.2 System Updating

Dynamic system updating requires a balance between several factors, such as system execution and type safety. As a result, it is possible to break a system

with incorrect changes, examples include untimely changes and unfinished alteration. Therefore, it is important to utilise appropriate techniques. To evaluate the capabilities of updating techniques we first need to determine the best characteristics of a system update technique. This information then provides the necessary details to evaluate updating techniques.

Firstly, we clarify that dynamic updating is the runtime alteration of a system. However, it is also known by several different terms: runtime manipulation, dynamic code, and system updating; we use these terms interchangeably. There are many different aspects of dynamic updating; for example, the complex areas such as version control, type safety, timing of an update and method of update.

The initial step is to define 'good system updating'. A simple definition could be: a good system update is one which alters the application correctly without changing current execution state. However, we note that this definition is not simple, but simplistic. Apart from this proffered definition, other definitions focus on creating simple generic means of system updating [50] and still others concentrate on type safety during system updating [53]. However, in general these definitions all identify the same characteristics when determining the usefulness of any system updating technique. These characteristics are: *Flexibility*, *Robustness*, *Ease of Use* and *Low Overhead*. Hicks [50] presents an explanation of these characteristics as follows:

- **Flexibility**: the ability to update all parts of the system without causing any runtime pauses or difficulties
- **Robustness**: the ability to minimise system errors during alteration
- **Ease of Use**: simple to use. (This can be seen from either the programmer's or the operator's point of view.)
- **Low Overhead**: the ability to update should give minimal overhead in both memory and processing power.

Any measurement of these characteristics is useful in analysing dynamic updating techniques if the techniques under analysis, in general, relate to the same programming language or languages of a similar nature. For example, techniques seen as flexible for C++ might use dynamic link libraries (`dll`) files, whereas, flexible techniques for Java might be those that make use of Java's interoperability and distributed capabilities. These examples, show that the flexibility of

a dynamic uploading technique can be determined by the core elements of the language's execution environment. Therefore, some measurements will be more adequate for some languages than for others. Moreover, it is definitely the case that for some languages it is much harder to create a dynamic updating technique than for others. We identify three levels of dynamic updating complexity:

1. **Languages which Include Dynamic Updates Capabilities:** A language has dynamic capabilities built into its standard structure, for example, Erlang [4].

2. **Languages where Inclusion of Dynamic Updates is Possible:** A language where the addition of the dynamic capabilities is seen as a moderately simple addition. For example, dynamic capabilities in C++ can be achieved via `dll` files [53, 49]. The research for these languages focuses on the problems associated with dynamic updating, such as type safety [49].

3. **Languages where Inclusion of Dynamic Updates is Difficult:** A language, such as Java [65] which was originally developed where dynamic updating was either not considered, or a decision was made never to provide it. Indicative of this is the list of different techniques that attempt to enable dynamic updating for the the Java language [69].

There is, of course, the existence of self modifying machine code, yet again, this is highly system specific and as such flexibly is inadequate. However, alterations of program flow and semantics can be achieved using `.dll` files or Universal Plug and Play (UPnP) devices [72]. For example, many operating systems constantly reload sections of code using both `.dll` files and UPnP (adding a new printer may cause a UPnP exchange) [72]. Unfortunately, some of these techniques for reloading programs are generally specific for each particular operating system and sometimes require resetting, restarting or pausing of the entire system.

The more consistently well regarded and utilised methods of system updating, and those included in our analysis, are those which incorporate C, C++ and Java languages as these languages are widely adopted and, as is more the case for Java, they maintain specific attributes making them useful in mobile devices. The dynamic aspects of traditional languages such as C, C++ have been covered by many researchers; for example, see the lists provided by either Hicks [49] or Hjálmtysson and Gray [53]. The extensive coverage of this area indicates that the ability to alter the behaviour of a system during runtime is difficult. Many applications that incorporate dynamic updates are designed for a specific purpose; for

example, updating software and versions and remote alterations. Consequently, many techniques are limited in generality.

Furthermore, the language type, and here we deal with interpreted (virtual machine) compiled languages, also provides an indication of not only the nature of system updating techniques but also the methods that can be applied.

An analysis of the techniques for use in analysing the dynamics of these languages is found in Hicks [49][1]. The list developed by Hicks illustrates the rationale for a new dynamic updating procedure targeted at C++ applications. Moreover, Vandewoude and Berbers [116] reiterate many of the observation made by Hicks during their analysis of dynamic updating techniques applicable for component-based embedded devices. We reintroduce the list of techniques by incorporating the more recent dynamic techniques and reintroducing those techniques seen as beneficial to mobile devices and systems. Table 2.1 gives this (brief) up-to-date reference for dynamic system techniques.

The table details the dynamic capabilities of each technique and gives an example of its use. It represents only a small, unique section of the entire research area and already we can see the large extent of the research. Other similar techniques not analysed in Table 2.1 due to lack of suitable qualities, are available from the literature (see [49, 50, 69]). With a few exceptions, the techniques, listed in Table 2.1, are generally Java and C++ based. This is a direct result of two factors:

1. these languages are the most widely adopted and have received the most attention by researchers, and
2. other languages contain dynamic constructs in their basic state, for example, Smalltalk [41].

[1]Hicks and Nettles [51] (to be published) have provided an update of Hick's dynamic updating technique. During their discussions in this paper, they include the further analysis of other dynamic updating techniques. However, as they point out, the new analysis provides no further details on current updating techniques, neither does it note any new techniques for updating systems. They simply refer readers to Hick's PhD thesis [49].

DVM [69]	The Dynamic Virtual Machine (DVM) is a specifically designed Java Virtual machine targeted with additional VM instruction support runtime alteration. The alteration of the code is type safe and the mechanism uses multiple versions.
JDrums [2, 29, 1]	JDrums is a virtual machine targeted at adding maintainability to Java systems. It is essentially an altered version of the Virtual machine much like DVM; however, unlike DVM it uses standard Java byte code.
Dynamic C++ classes [53]	This is the development of a type safe dynamic technique for C++ systems. Tested through the use of ftp and web servers, it has demonstrated the potential of dynamic code updates across a network.
Dynamic Update [49]	Dynamic Update is designed to provide C++ updatable applications with a balance between safety and flexibility.
Erlang [4]	Erlang is a dynamic logic language specifically designed with networking, concurrency and dynamic code techniques. Although not widely adopted Erlang, or the concepts discovered from Erlang, demonstrates that a language can be designed as dynamically updatable.
Tcl/Tk[85, 112]	Tcl is a scripting language which includes constructs for dynamic code updates. Tcl is widely used for both simple and complex applications.
DUCS [12]	DUCS is a Dynamically Updatable Component based System. It is designed to allow distributed systems to atomically substitute new components.
Javaassist [24, 23]	Javaassit is an aspect based Java development tool that incorporates some dynamic updating through reflection packages within the aspects. As Javaassist's primary development aim is Aspect Oriented programming the scope of dynamic changes are limited and used for aspect manipulative purposes.
JMangler [61]	JMangler is a framework for generic interception and transformation of Java programs at load-time [60]. Although JMangler only alters the code at load time this does give a choice as to which classes an application should load and implement, giving it attributes similar to Runtime alteration.
Colomba [11]	The Context-and-Location-based Middleware for Binding Adaptation (Colomba) is a Meta-Data based Framework for the manipulation of system semantics throughout a network. Colomba is designed to allow a degree of code sharing and transportation based on Meta details for each class.
JavaSymphony [39]	JavaSymphony allows a programmer to dynamically alter a distributed architecture.
JOIE [25]	The Java Object Instrumentation Environment (JOIE) is a framework for safe Java byte code transformation.
J2SE 1.4 HotSpot JVM [103]	The HotSpot Virtual Machine extended Sun's Java Virtual machine with the Java Platform Debugger Architecture (JPDA), using the Java Virtual Machine Debug Interface (JVMDI) [103]. It is designed for debugging systems and code tracking. The attributes of the JPDA have now been incorporated into the JDK 5.0 as a standard.
Java 5.0 Standard Edition [108]	The newest version of Sun's Java Virtual Machine has incorporated the concept of runtime code manipulation. With the use of the java.lang.Instrument package Java applications are able to define section class loading techniques that enable a previously specified jar file to manipulate class bytes.

Table 2.1: Dynamic Updating Techniques and Applications

2.3 Dynamic Techniques for Mobile Systems

Research in the area of dynamic systems in mobile devices is noticeably less extensive than in standard systems. We are specifically interested in the techniques most applicable, or foreseeably applicable, to mobile devices and, therefore, we will concentrate on further analysis of the techniques listed in Table 2.1. We will firstly analyse the specific needs, nature and restrictions of a mobile system. This will be used to derive a set of requirements that will in turn be used in a further analysis of dynamic updating techniques to determine their use for mobile systems. However, not all techniques need necessarily be currently available within mobile devices, only perceivably.

2.3.1 Limited Capability Systems

We consider a limited device to be one which has restricted capabilities as a result of a specific attribute. A limited system has restricted capabilities as a result of its execution on a limited device. We feel these systems will benefit significantly with the use of our framework; particularly those which are mobile.

Sensor devices are extremely limited devices. In general, they have limited battery power, processing capabilities and provide very little data storage room. However, they are physically small, and are getting smaller, and therefore can provide valuable data from difficult locations. Current advancements in the technology are providing sensor and embedded devices with communication and cooperation techniques. However, the communication is limited to a single wireless protocol. Although, this provides sensor devices with small-scale interoperability capabilities, the analysis of current developments may provide valuable insight into their future

Although sensor devices are generally developed for a specific use, more general devices are beginning to emerge. These devices consist of adaptable mechanisms to allow them to be applied to several different tasks. CSIRO's Fleck device is one such device [28]. A Fleck device consists of a low power CPU with additional flash memory, a radio transceiver and built in temperature and charge current sensors. It also allows for sensor boards to be added on to further device capabilities. In order for a software application to execute on a limited device, operating systems (OS) with small footprints are required; for example, Fleck devices run TinyOS [52]. The operating system used by such a device may be written specifically or

may be a standard OS adaptable to several devices; the TinyOS is an example of the latter.

Some small operating systems offer advanced flexibility. Of interest are those which are capable of dynamic code updating. The Contiki OS is one such example [35]. Contiki is a lightweight operating system for networked sensor devices. It includes support for dynamic loading and the replacement of programs and services. Contiki's ability to dynamically update its code is advanced for a lightweight OS. For example, although it is theoretically possible to change application code in the TinyOS, using the nesC language [40], it would be restricted to full system alterations. This limitation is a result of nesC's expectation that code is compiled and distributed as a complete program [40]. Contiki on the other hand is capable of reloading code at runtime. Moreover, this code may also be downloaded at runtime from another remote system. Although Contiki systems are able to communicate in such a scenario, they are limited to those using the Contiki OS, which itself is limited to few devices [35]. Interestingly, the Contiki developers note in their reasoning for Contiki that:

> *the possibility to dynamically load individual programs leads to a very flexible architecture* [35];

our work extends this concept by dynamically reloading program segments.

2.3.2 Analysis of Mobile Systems

By mobile systems we mean physical mobile systems not agent systems. Mobile systems present several limitations and unique attributes due solely to the nature of the devices. Mihailescu [75] gives a detailed analysis of mobile systems, therefore, we only need to consider those aspects of a mobile system that determine the limitations or present new capabilities and requirements for dynamic manipulation. The most obvious limitations of a mobile system are those which are directly due to their size, power, I/O peripherals and connectivity. These are shown in Table 2.2.

These attributes can be used to analyse the applicability of each dynamic updating technique to mobile systems. The main attributes that determine if a dynamic updating technique is applicable to mobile systems are:

- **Size**: The technique must be as small as possible in memory footprint.

Size	Memory use, CPU processing requirements, video needs, speed requirements.
Power	Peripheral use, speed, memory and CPU use.
I/O Peripherals	User input needs, display needs, connectivity and interoperability requirements, networking.

Table 2.2: Mobile Device Limitations and Associated System Requirements

- **Memory Use and Speed**: The technique should use as little as possible of the device's time and resources.
- **Flexibility and Portability**: The technique should be as flexible and portable amongst as many devices as possible.
- **Connectivity**: The most advantageous techniques will make use of the extended connectivity offered by Mobile Systems.

The ability to be compatible with the above four attributes makes a dynamic updating technique appropriate for use in mobile systems. The next section will investigate other feasible dynamic techniques in respect to this statement.

2.3.3 Appropriateness of Dynamic Techniques for Mobile Systems

As previously mentioned most dynamic updating techniques are designed for specific reasons (see Section 2.2). JDrums [2, 29, 1], for example, was designed to understand the implications of dynamic alterations of an object-oriented program using the Java Virtual Machine (JVM) as its test case. As JDrums was not designed with mobile systems in mind it contains many aspects which make it unusable in such devices. Most detrimental to techniques such as JDrums is that it is a modified JVM. This is fine for standard personal computers, however, mobile devices are made using specific hardware and operate using specifically designed operating systems (OS), they, therefore, require their own specialised JVM. Consequently, most other modified versions of virtual machines are incompatible with the JVMs of many mobile devices. Therefore, this technique is generally not suitable for mobile systems.

The exception is when modifications are applied with a mobile device in mind. However, our research has yet to find such a case. Nevertheless, it is possible,

and probable, that, in the future, some of the techniques we have discussed will be deployable on mobile devices. Responding to such a probability, the following analysis provides an indication of the likelihood of each techniques distribution within future mobile devices.

A measure of the appropriateness of each of the techniques given in Table 2.1, is now presented in Table 2.3. This Table indicates that some techniques are significantly more appropriate for mobile devices than others. In particular, we see that techniques of dynamic alteration using Virtual Machines, specifically the Java Virtual Machine, provide greater flexibility. This point, in a similar fashion to the questions raised by Vandewoude et al. [116] in respect to embedded systems, raises a new, important question:

Is a targeted language more appropriate for mobile systems than a specific dynamic technique?

To address this, we will split the techniques into several language groups according to the levels of dynamic updating complexity developed in Section 2.2. These groups are:

1. **Languages which Include Dynamic Updates Capabilities.**
2. **Languages where Inclusion of Dynamic Updates is Possible.**
3. **Languages where Inclusion of Dynamic Updates is Difficult.**

Furthermore, as discussed earlier, it is likely that Java will be the most appropriate language. Therefore, we will present a more extensive analysis of Java in the Difficult Inclusion analysis, Sections 2.6, 2.7 & 2.8.

2.4 Languages which Include Dynamic Update Capabilities

Some languages inherently allow applications to dynamically update their runtime execution. Prolog, a logical programming language, is an example of such a language. The extensibility of Prolog allows developers to use internal commands and techniques to assert and retract code (multiple ways in which the same code section can change) [8, 73], through the use of the specialised commands: `retract` and `assert`. Such commands allow the dynamic alteration of Prolog

Technique	*Current Situation*	*Future Deployment*
DVM	Allows for runtime updating. Some hard coded updating methods are required to be included.	As a JVM implementation the DVM provides the possibility of designing an implementation specifically for mobile devices.
JDrums	Development is progressive with this JVM and has seen the arrival of JPatch [46] which uses the JPDA structure [99].	The progressing developments of JDrums are positive for its future use within mobile devices.
Dynamic C++ classes	Advanced updating is allowed with this implementation. However, applications that use this technique must be programmed with updating in mind.	This technique is only deployable within specialised devices that it have been designed for. This lack of interoperability deems it unlikely to be deployed on mobile devices.
Dynamic Updating	This technique has been shown to work using web based triggered updates. It is a safe mechanism for C++ updates. However, it is constrained by its need for hard coded updating knowledge.	For much the same reasons as Dynamic C++ classes, it is unlikely that Dynamic Updates will be distributed among mobile devices.
Erlang	Erlang has been specifically designed to be dynamically updatable from the beginning.	Erlang was developed for the telecommunications industry and has been applied within several mobile devices.
Tcl/Tk	Tcl is a scripting language which includes constructs for dynamic code updates.	Tcl has been used on mobile devices for the purpose of supporting other languages.
DUCS	DUCS is a component based updatable framework. Its work has progressed in term of its distributed applicability.	As a component based architecture the extension to mobile devices is limited, however, its distributed concepts are an interesting feature.
Javaassist	Java assist allows byte code manipulation during runtime. Code can be written, using the javaassit package, that changes the class structure.	If the required packages are available within a mobile device with enough processing power this technique is deployable.
JMangler	JMangler is a manipulation technique that alters code during runtime. Run time support is minimal.	Due to its updating nature this technique is not seen as applicable to mobile devices.
Colomba	Columba is more than an updating technique it is a distributed means of achieving updates. Moreover it is designed for mobile devices.	Although, Colomba is implemented on mobile devices, it is not flexible in terms of extension. Its structure is designed for a particular purpose and would not be applicable for any separate techniques.
JavaSymphony	The dynamic JavaSymphony uses distributed architecture to alter runtime behavour of distributed nodes. As such, it has been included as part of a distributed performance measurement tool.	This technique works for cooperating multiple systems, such as Grid systems, however, within an individual mobile system the package is not effective.
J2SE 1.4 HotSpot JVM	The usefulness of this technique is seen in it inclusion in many debugging systems, including JDrums new extension JPatch.	The JPDA structure requires a large amount of system resources, however, as mobile devices grow this implementation would be possible to be included.
Java 5.0 Standard Edition	Allows for restricted class updates during runtime, via direct JNI modification code.	We view this technique as the most likely to be deployed on a wide range of mobile devices.

Table 2.3: The Appropriateness of Dynamic Techniques in Mobile Systems

code. However, due to their high flexibility, they are deemed to be 'dangerous'. Despite this, several other conventions that are necessary for our work, such as discovery (see Chapter 4 and 5), are non existent.

Languages such as Erlang [4], Tcl/Tk [112] and Smalltalk [41] give strong and specific means of dynamic alteration. Other languages, such as Prolog, Perl, Shell and Lisp, also include, or have been extended to include, dynamic redefinition capabilities. Although, many of these are limited in functionality, of specific interest are Erlang and Tcl/Tk. Both these languages inherently support several features required for our framework, such as advanced dynamic updating capabilities and distributed communication. They will now be discussed individually.

2.4.1 Erlang

Erlang is a concurrent programming language designed for distributed soft real-time applications [4]. Erlang was initially deployed by Ericsson as a language to develop large-scale telecommunication systems which required soft real-time applications [4, 38]. The language has since been released as open source and is now used in the development of a number of applications; for example Cellpoint use Erlang applications within mobile services [21].

Erlang is required to provide continuous operation to applications. To achieve this, runtime system upgrades are necessary. This particular attribute gives Erlang its fault-tolerant attribute. To allow for runtime upgrades Erlang can hold duplicate instances of a single function module. This is possible as Erlang's language constructs limit the visibility of functions to associated modules. This modular encapsulation is such that a function can only be accessed from outside the module if it is exported to do so. A function can then be called using the module and function names.

Understanding the modularity of the language provides an insight into how it achieves dynamic updating. However, to completely understand Erlang's dynamic updating capabilities knowledge of its use of process identification is required. A process is a key concept in Erlang and therefore all processes are identifiable. A process is referred to by its process identifier. Process identification is used to establish Erlang's easy-use distributed methodologies. For example, any external resource, such as files and network connections, are identified as a process by Erlang; although understandably, such processes do have certain

restrictions [18]. In combination, Erlang's process identification and modularity allow it to trace all processes executing any specified module.

Therefore, the redirection of new process calls to a specified module can be achieved without interfering in current execution. This is achieved by using multiple versions of the module. An Erlang system is able to redirect all new module calls to a new module version while all currently executing processes continue using the old version. The entire old module can be replaced once all the old executing processes are finalised. Moreover, Erlang provides the necessary primitives to gain access to the execution status of each module version [48] allowing the programmer to fine-tune system updating.

Another interesting aspect of Erlang, in respect to our framework, is its distributed functionality. Erlang was developed to run in a distributed environment [3]. The internal structure of Erlang provides simple techniques for distributed communication allowing a system to treat a remote process as another node of processing. Furthermore, all distributed mechanisms work synchronically with both the concurrent and dynamic updating capabilities. Therefore, it is plausible that Erlang may be applied to generate dynamic updates via distributed triggers and code sharing. As indicated in Chapter 1 this attribute is a requirement of our work.

However, although the development of Erlang has far surpassed its initial expectations, and although it is a useful tool for mobile applications [70], some methodologies which we require for our work, such as service discovery, are not currently available.

2.4.2 Tcl/Tk

Tcl/Tk is the combination of the Tool Command Language (Tcl) and the Tk GUI development tool kit. Tcl is an interpreted scripting language which is powerful and flexible enough for the development of both desktop applications and networking systems [112]. Much like Erlang, Tcl is another example of a language with inherent dynamic update capabilities. The language provides constructs for renaming and deleting procedures, and for loading new procedures via specified files. These aspects of the language are in built. Furthermore, they are designed to give a programmer control over the internals of Tcl [85]. By using a combination of these language constructs it may be possible to achieve triggered dynamic

updating. Ousterhout [85] details how an application can rename a procedure, for example change the procedure name myProc to myProcOld, and add a new version of the old procedure, in our example a new myProc version. In addition, it would be a simple step to provide the new procedure details using a file which could be possibly downloaded from a remote server. This degree of flexibility could prove useful for our framework.

Of further interest in this language is the Safe-Tcl sub-language [16, 86]. Safe-Tcl is a restricted language. It has limited file access and it restricts the execution of many system commands. Safe-Tcl code is executed using two separate interpreters, a safe interpreter and a master interpreter. The safe interpreter will execute all code from an untrusted source, however, if the code is known to have come from a trusted source it may be given extended capabilities by executing it using the master interpreter. Both interpreters are used together via an alias mechanism from the safe interpreter to the master. Therefore, code from an application can be interpreted using either interpreter depending on execution circumstances; the selection process is generally automated [85]. This essentially provides the language with an adaptable sandbox style execution for downloaded code.

We can determine that Tcl/Tk, or specifically Safe-Tcl, provides the safe distributed technologies which are required for our framework, and we also note that Tcl has clean, easy integration with other programming languages, especially Java; for example, Jacl [113] . However, Tcl/Tk does not currently support interoperability among systems and does not provide high level mechanisms such as service discovery.

2.5 Languages where Inclusion of Dynamic Updates is Possible

C++ programming has several flexible features that aid development of dynamic uploading. Pointers and dynamic memory techniques facilitate the creation and removal of objects and variables and allow programmers to manipulate different versions of a single object [50, 53]. Common issues associated with dynamic updating are present in C++. These include problems such as type safety and agility [50]. Irrespective of this, C++ has been used for such purposes. For example, Hicks [50, 49] used C++ to construct a dynamically changeable web server,

FlashEd. FlashEd allows users to upload pre-tested code (compiled on a testing station) over the web to dynamically alter its execution. This illustrates that, as a result of its constructs, C++, and C to an extent, can achieve dynamic alteration of a system if implemented correctly, and in a safe manner. Even though mobile device usage of C++ is broad, it remains platform dependent due to its compilation to specific machine code. The system dependent nature of the languages C and C++ restricts its implementation for a mobile environment. This is a result of the instruction sets.

C# is immensely similar to Java. It has little access to pointers and consequently is less flexible than C++ or C for dynamic update manipulation. Nevertheless, its access to memory spaces and objects is achieved more freely than in Java [74]. Moreover, as C# runs within a Virtual Machine it provides a level of system interoperability. Despite this, currently its inclusion within mobile devices is limited. However, as C# is a relatively new language there is little research in the area of runtime dynamic alteration. Moreover, as a result of its large footprint size [74], its deployment within mobile devices is rare and likely to remain so for some time.

2.6 Languages where Inclusion of Dynamic Updates is Difficult

As previously mentioned we focus this analysis specifically on the Java language. Java has a unique ability to provide uniformity to multiple operating systems [65], a feature that is useful for mobile devices. This uniformity is possible as Java is an interpreted (VM) based language that allows programs to (theoretically) be written once for use on multiple systems[2]. As an interpreted language, the internal construction of any JVM must hold to a particular set of rules and guidelines. If a VM can correctly execute Java byte code instructions written to a particular level (the level is defined by Java configuration) then that VM can be specified as a conforming JVM, or at the least a restricted JVM. For example, the KVM [98] and CDLC HotSpot [107] JVMs are both designed for limited resource devices. However, a conforming JVM may also provide extra capabilities via additional byte instructions. As a result, although the VM conforms to standard

[2]In practice applications are at times written for specific operating systems due to constraints, system format and use of the Java Native Interface [63].

JVM requirements, its extended capabilities deem it a non-standard JVM.

Section 2.2 indicated, in Table 2.1, that there are non-standard JVM implementations that are specifically designed for dynamic system alterations. It is necessary to analyse their techniques closely to determine if Java is appropriate for our purposes. Initially, we find that these techniques manipulate Java in different ways: some involve redesigning the lookup and linking of Java class files, while others allow direct inclusion of system code. For example, the use of specialised class loaders within the JVM's class loading structure provides a system with the necessary flexibility to access the JVM's internal workings [65]. As a result, Java class files may be found and loaded in any specific manner.

To understand how Java dynamic updating techniques work, a simple understanding of a JVM's structure and code manipulation is necessary.

2.6.1 The Purpose of Java

Java was developed with ideals of portability and mobility in mind, and, although its growth has led it into more diverse areas these main objectives still remain [65]. The JVM is designed in accordance to its original specifications. The most important is that the JVM dynamically loads Java classes during runtime and develops class instance objects as required using class loaders. This allows the JVM to load a class once and reuse (referencing) the same class object each time an instance of it is created [65]. For class loading to occur, the specifics of the Java class file must be adhered to.

2.6.2 The Java Class File

As the Java class file format is specific, it includes mechanisms that allow it to be identified as a Java class file, the most direct is the magic number [65]. The magic number is a secret hash that corresponds to the compiler, target virtual machine and class. Further details on the constructs of the actual class bytes, for our purposes, are not important. What is important is how the class bytes are verified and how they are used during runtime.

2.6.3 Class Verification and Distribution

A JVM uses three steps to further ensure that the class bytes gathered from a class file are correct and pass certain verification tests. These three steps remain consistent, irrespective of how the class bytes are obtained, and are designed to check class details before it is initialised as a Class object within the JVM. If any of these steps fails, or if the magic number does not a match, a class will not be loaded by the JVM. The steps are:

- *Class Loading*: Loading is the process of finding the binary representation of a class or interface. Each Java class that is to be used by an application will have its byte code representation loaded by the JVM when required. This representation is then used to create a Class instance and generate all object instances of that class [65]. Moreover, a class is never initialised and distributed throughout the JVM's Runtime Constant Pool [65] until linking time.
- *Class Linking*: Linking is the process of taking a class or interface and combining it into the runtime state of the JVM so that it is ready for execution [65]. The time it takes to link a class is equal to the time it takes for the JVM to create a corresponding Class object.
- *Class Initialisation*: Initialisation consists of the execution of a class or interface. Class Initialisation is initiated via the `<clint>` method [65].

After the final steps of initialisation are completed, a reference to a class instance (object) is achieved by using a pointer to an object handler that is itself a pair of pointers. Of the pointers within the object handler, one points to the section of a table that contains the methods of the object, which itself includes a pointer to the representing Class object for correct method execution. The other pointer indicates the actual memory space that is allocated from the VM's heap for the object data details [65]. This pointer structure is shown in Figure 2.1. (Further information on the Java class bytes and details of the JVM's structure are outside the scope for this work and is available in Lindholm and Yellin [65].).

2.6.4 Java Class loading

Java's dynamic class loading mechanism builds the data structure representation of classes in the JVM. It determines the way the JVM manages class objects and their instances. We have already mentioned that class loaders control all loading

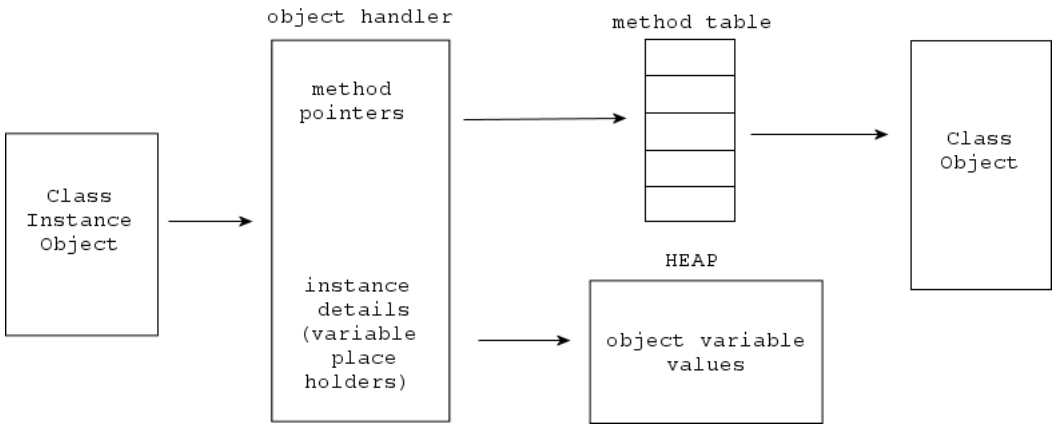


Figure 2.1: Class Instance Referencing in the JVM

of class files for an application. And, although loading, linking and initialisation is controlled by the JVM, the class loader(s) always initialise the loading sequence [65]. Java's class loading structure includes the `java.lang.ClassLoader` class. Subclasses of this can be used by developers to dictate the manner in which the class files are found and loaded. New class loaders are loaded by existing class loaders which results in a tree structure for all class loaders. For example, Java applets manipulate the class loading structure to safely download files from a web location.

Java's class loading is optimally designed for safe dynamic linking and loading of class files. Furthermore, whilst it allows files to be located, loaded, and linked during runtime the class loader also aids in the segregation and distribution of byte code within the JVM [65]. The dynamic nature of Java's class loading and its close connection with byte code allows designers to use specialised class loaders to manipulate class files in unspecified ways. Moreover, the inheritance structure that is produced when classes are loaded into a JVM allows a class loader to define the entire loading aspects of an application. In this respect, as classes are loaded via a class loader, any subsequent classes, or classes instantiated via an already loaded class, will be loaded via the same class loader [65]. In practice this allows program designers to load an initial class (generally a start up class containing the `main()` method or `init()` method) through a specialised class loader resulting in the entire system being loaded through this class loader (a subclass of the `java.lang.ClassLoader` class). Furthermore, if the same system then loads another different class loader, say for a specific application section, the additional class loader will also be loaded through the first class loader. And, if,

for any reason, the second class loader then fails to find a class, all class processing will be delegated to its parent class loader (the first class loader) and so forth [65].

2.7 Dynamic Updating Techniques for Java

We previously mentioned that the ability to update Java application is difficult due to the restrictions of the JVM. Its inability to unload and reload classes results from the bootstrap class loader disallowing such unloading for type safety reasons. Officially the Java Specification book states that:

> *A class or interface may only be unloaded if and only if its class loader is unreachable. The bootstrap class loader is always reachable; as a result, system classes may never be unloaded* [65].

From this statement, it would be reasonable to assume that a class can *never* be unloaded from the runtime state of a JVM. However, Liang and Bracha [63] identify a type safety problem with the JVM's class loading which is related to dynamic class updating. They describe how the combination of class loaders and the Java Core Reflection API, or a predefined interface, can be used to manipulate classes. The technique exploits the fact that a class loader enables the unloading of classes that are no longer referenced, or indirectly referenced, by any existing instance objects [117]. This is a result of the name space identification of classes by the JVM using the full class name, including packages, and the identifier of its class loader; for example, `java.lang.String` loaded by class loader `classLoaderA` would be identified as `classLoaderA.java.lang.String`. Therefore, if the `java.lang.String` is the sole loaded class of classLoaderA, and if there are no instances of String remaining, then the String Class object will be unloaded. Liang and Bracha's technique loads a class using a predefined interface and never directly refers to the class within the application. An example of their technique is shown in Figure 2.2.

The example code uses an Interface when referring to a class that is targeted for alteration (`Service`). It also uses a temporary class loader (`MyClassLoader`) to load the class. Therefore, according to our analysis in Section 2.6.4, two versions of a class can exist at the same time; for example, `tmpClassLoaderA.MyClass` and `tmpClassLoaderB.MyClass`. And, as each of these class versions can be referred to via the predefined interface (`ServiceInterface`), a new version of the

```
class Server{
            private ServiceInterface service;
            public void updateService(String location) {
                        MyClassLoader cl = new MyClassLoader(location);
                        Class c = cl.loadClass(``Service'');
                        service = (ServiceInterface) c.newInstance();
            }
            public void processRequest(...)
                    service.run(...);
            }
}
```

Figure 2.2: Exploiting Java's Class Loading

class can be defined. However, the new version of the class will only be used for new instances. The old class version instances will remain the same until their use is complete. Once all old class instances are complete and unloaded via garbage collection, the old class version will have no more objects and it too will be unloaded.

Although it is possible to write a complete application referring to classes via interfaces and use a single class loader for each class, it is not a standard procedure. Theoretically however, the technique does indicate that it is possible to unload a class. However, the interface restrictions also indicate that the technique, up to the point to which Liang and Bracha had researched, will not allow a complete reload of the current instances of the class.

The authors of JMangler [61] indicate that the addition of classes and interfaces, and the alteration of signatures and code structure are aims of their research. These aims coincide with those generally set out for runtime alteration techniques. However, all that has been achieved thus far in this area is outside the scope of runtime dynamic updating.

Even though dynamically unloading and overloading class details and their related instances is extremely limited in standard JVMs, and the JVM's controlled structure seemingly inhibits the deletion or replacement of any running classes, there is research being accomplished to change this. Malabarba, Pandey, Gragg, Barr and Barnes [69] identified three techniques that are generally used to allow dynamic updates within Java:

1. *Virtual Machine Extension.*

2. *Library Based Solution.*

3. *Language Creation.*

A further technique for Java class alteration was introduced by Bellavista, Corradi, Montanari and Stefanelli [11]. They successfully established a *middleware based* means of altering the structure of Java class bytes during runtime. This makes a total of four vastly different techniques. We will analyse each of the four techniques and indicate any perceived benefits or implications of their implementation within mobile systems, current or futuristic.

2.7.1 Virtual Machine Extension

Virtual machine extension is the creation of a new non-standard JVM that is based on a targeted JVM specification. It is designed to adhere to all aspects of the originating JVM and incorporate the ability to dynamically update an executing program [29, 69]. This technique is the most common of the four techniques. It creates a fast and unique solution which is always open to further development. Examples of such JVM's include, DVM [69] and JDrums [29]. All subsequent JVM's using this technique must be able to execute all Java programs (some are dependent on a specific API). However, as many ubiquitous and pervasive systems rely on interoperability, the simpler it is for a system to be adapted to a device the more likely it can be adopted. This technique requires each device to contain the extended JVM, therefore, limiting the scope of applicability.

As previously mentioned, JDrums [2] was initially developed for the simple dynamic alterations of Java programs however, the scope of the project has now grown. JDrums is an altered JVM and associated toolkit which enables the updating of Java class objects and their referencing instances. This was originally designed for mobile devices, however, there were problems due to memory management. This change in direction of the scope is a result of its use of both an OLD class object and a NEW class object for each overwritten class. This characteristic means it must maintain storage space for two classes of the same type. Further to this is the use of a toolkit which increased the footprint of JDrums. JDrums may not currently be the best fit solution for dynamic mobile systems, however, it does detail the implications and problems that exist when dynamically updating Java applications in this manner. Moreover, continuing research

is leading to a more efficient means of dynamic updating using JDrums[3].

Other extensions of the JVM have led to the development of design specific virtual machines. Such JVMs enable new operations to take place or to be achieved more easily. Many virtual machines have been for developed limited devices, for example Lego Mindstorms [118, 114]. However, unfortunately, these JVM's limit the user and it is impossible to execute a standard Java application.

2.7.2 Library Based Solution

Library packages may be designed to give a program the ability to update its structure. As long as their constructs are followed, the library classes give a program the ability to dynamically update its structure. In many cases, specified conditions must be satisfied in order for application updating compatibility. This technique, unfortunately, leaves the programmer to determine what sections should, or should not, be flexible in terms of dynamic updating. Library based solutions give similar computational results to that of a JVM execution. However, there is an associated loss of speed and a likely need for program statement inclusion that may be seen as a deterrent by prospective developers.

One major benefit of this technique is that it is more flexible than an extended JVM as it does not require changes to the JVM. Logically, this seems to be the best and most flexible solution for mobile systems. However, a need for programmers to design specifically around the library, or at least include it within their system, is a hindrance to applications [1] in that a system is only dynamically updatable if it is designed to be. Furthermore, program inclusions may require interface implementation, specific code inclusion or technique alteration, all of which could limit the functionality and flexibility of a system.

2.7.3 Creation of a New Language

Although this technique does not allow an established language to become dynamically adaptable, a new language is generally based on an existing language and mostly adheres to its specifications, thus giving it the same programming 'feel', 'look' and 'use' [4]. The creation of a new language is seen to be the best

[3]JDrums has recently been extended into JPatch, however, as at the time of this publication there are no publications of subsequent findings, only web details available at the home page [46].

solution for dynamic updating. From initial stages, anew language would adhere to all necessary requirements and constructs. It would incorporate all aspects of type safety, concurrency, communication and speed that are otherwise seen as potentially dangerous for dynamic altering systems.

However, the creation of a new language results in just that, a new language. Very few devices would be capable of implementation; it is likely that little work would be achieved within it and consequently, various bugs will be present. This is not to say that eventually a new language could not be widely adopted, as in the case of Java, yet, for the purpose of a mobile system the use of an established language is advantageous. This is due to existing reliability, multiple platform compliance, population conformance and resource availability.

2.7.4 Middleware Application

Middleware application is a technique which takes a separation of concerns approach to system dynamic updating. By separating a target system from its dynamic updating framework, a middleware framework is able to change a system's execution via introspection or manipulation [71]. In terms of system updating, the middleware framework is the most recent. This technique is particularly important as it incorporates the flexibility of a library technique, yet, much like some altered JVMs [69], it excludes the need to program code in a specific manner or use a specific API. This technique may rely on other techniques [11]. For example, a middleware may be able to dynamically update any system by using a specific library solution or targeting a specific VM.

Middleware architecture implementations are commonly used within distributed systems [79]. They have led to the development of several separate techniques that extend distributed interaction through the implementation of an application [11]. Bellavista, Corradi, Montanari and Stefanelli [11] approach dynamic manipulation from a distributed systems point of view. We discuss their technique in full later in Section 4.4. This particular method of updating combined with distributed activities will be closely analysed throughout this entire thesis as it is the core of the framework.

2.8 Standardised Dynamic Virtual Machine

Developments in standard JVMs, in particular from version 1.2 to 1.4 to 1.5, reflect the progress in dynamic updating. Most notable, for this research, are advancements in Java's standard API [108].

Since Sun sets the standard for Java, they are able to evolve the JVM standard, unlike non-standard JVMs such as DVM [69] and JDrums [29]. This means that any new standard JVM creates a flexible and interoperable means of dynamic updating. Moreover, if combined with a middleware solution, as described in Section 2.7.4, it is possible that any system implementing on any device could be manipulated via the middleware to become dynamically updatable. This is a situation which previously had not been achievable.

There are two main developments in standardised dynamic updating of virtual machines: the *Java 1.4 HotSpot Virtual Machine* and the *Java 5.0 (1.5) Standard Virtual*. We will now briefly discuss these developments as they are discussed in detail in Sections 10 and 9, respectively.

2.8.1 J2SE HotSpot Virtual Machine

The Java HotSpot Virtual Machine, available in J2SE version 1.2.2 to 1.4, allows dynamic class manipulation. This feature is provided through an extension to the Java Platform Debugger Architecture (JPDA), or more specifically, the Java Virtual Machine Debug Interface (JVMDI) available through the Java Debug Wire Protocol (JDWP) and the Java Debug Interface (JDI) [103]. It is these sections of the JVM which give it the ability to substitute class code in an executing application. The JPDA structure is designed to use a monitoring section (front-end) and an implementing section (back-end). The front-end is able to monitor the back-end as a JVM object executing a target application. This allows the front-end to watch and control the progression of the targeted program.

We provide a technical analysis of this JVM is Section 10.2.

2.8.2 J2SE 5.0 Instrumentation

Even more recently, the J2SE 5.0 [6, 108, 104] has incorporated dynamic class updating as a functional feature of the standard JVM [5]. This is achieved through the inclusion of a new system package `java.lang.instrument`. When included, this package gives a program deeper capabilities than standard introspection. The `java.lang.instrumentation` enables applications to access their own class structure and alter running class procedures. The technique is aimed to aid in the creation of debugging and analysis tools [104]. However, we maintain that it can be used for other purposes of system manipulation. This would be the most desirable technique if it becomes available in small machines.

We provide a technical analysis of this JVM is Section 9.2.

2.9 Dynamic Code in Mobile Systems

As previously explored, limited memory devices, such as PDA's, mobile phones, embedded systems, and wearable devices, represent a technology inhibited by their platform specific software and hardware restrictions. However these are trade-offs for portability. Mobile devices contain software that is designed for use on its hardware platform and operating system. For example, many devices contain a JVM that has been specifically designed. The individualistic nature of these system designs limit the methods that may be used for dynamic system alteration.

Throughout this chapter, we have continually indicated that there are limitations in the application of dynamic updates in mobile systems. However, for now, let us pretend that this is not the case. This will allow us to analyse the extra capabilities dynamic procedures can give to mobile systems.

The usefulness of dynamic updating within a mobile system differs from that in a standard system. Firstly, as with standard computing and server systems, software updates, bug fixes, and maintenance are still beneficial to a mobile system. However, the unique nature of a mobile system allows the reason for change to be different. The novel applications that can be built using this capability are explored in Section 6.6. The differences in the use of dynamic updating for mobile systems are due to one specific quality: portability, which itself leads to

cooperation and interoperability. Furthermore, all systems on mobile devices can be split into two categories: those that interact by any means with other systems within the network, for example browser, email client application, and those that are device bounded and have no, or very little, interaction with the network at all, for example, word processing, calculator and a calendar.

Recently, system behaviour alteration has been used to change the state of mobile devices. The use of these techniques in current systems is such that some degree of consideration must be applied when coding an updating system [11]. This shows that there is a need for a dynamic updating technique that has no coding requirements and that makes use of the attributes of mobile systems. This technique would be designed to exploit many of the aspects of mobile devices that we have previously discussed, and, in particular, target their portability and network diversity. The design of the technique is such that most systems can utilise its capabilities for their own benefit. We explore this in Chapter 4.

2.10 Problems Associated with Dynamic Alteration

We differentiate between dynamic alteration and dynamic updating. The problems with dynamic alteration are: updating continuous looping sections of code, changing class instance to a Global instance, and those associated directly with the updating of an existing class object, for example, alteration of private variables, reverse referencing from within a new object from a new class (passing in a current object), and changing all same type objects into a single global object [2]. We do not attempt to solve the problems associates with dynamic alteration we are concerned with aspects of dynamic updating.

During our discussions in this chapter, we have noted the problems which are seen to be detrimental to mobile systems, however, we have not taken into account the problems associated directly to a system updating technique implementation. We have determined that it will be such complications that will determine the use of any framework implementation. Therefore, what we do assess in this thesis is the use and reason for dynamic updating. This is covered in the respective implementation chapters (Chapters 10 and 9) and evaluation (Chapter 11).

2.11 Summary

This section has provided a background on dynamic updating and the many different techniques used to allow this. We have targeted our discussion to dynamic techniques within mobile systems. It is also suggested that although not all techniques are appropriate for mobile systems many are, and others are likely to be in the future. It can, therefore, be argued that our framework can specify that a system must have the ability to dynamically update, however, it is not necessary for the framework to specifically designate the method used to achieve the update. We will present two different mechanisms for use in two of our framework implementations, (see Chapters 9 and 10).

The chapter also suggests that for mobile applications the restriction of any dynamic technique is best limited to a single language. We have also argued that the most appropriate language for mobile systems is Java and, therefore, the framework will specify that all target applications will be written in that language. In the next chapter we explore the benefits of discovering the code remotely and analyse it in terms of dynamic updating.

Chapter 3

Distributed Cooperation

3.1 Introduction

In the previous chapter we analysed the dynamic updating capabilities of different languages and assessed their use within mobile systems. The focus of this chapter is the analysis of techniques for distributed system cooperation. We analyse the techniques in order to establish a means of remote code loading for our framework. However, to take into account the requirements of mobile systems we must initially detail the communication capabilities of mobile devices. This information is provided to demonstrate how mobile systems communicate and to introduce the physical aspects that all distributed cooperation techniques are built on. Following this, we analyse the different discovery techniques used for remote activities, such as client server and peer-to-peer applications. In addition, we differentiate between their use as standard networked environments and advanced system cooperation.

The chapter will detail the different capabilities each distributed technique contributes to system cooperation paying particularly attention to their use within mobile systems. A primary aim of this chapter is to analyse and determine an appropriate distributed technique for the cooperation and remote code transfer aspects of our framework. This will provide the basis for community communication and information transfer[1].

[1]Further reasons for the choice of the distributed technique for DUPE will be presented in the chapters on framework design (Chapter 5) and community cooperation design (Chapter 6).

3.2 Advancements in Mobile Technologies

There are many, new technology advancements that are changing the capabilities of mobile systems. Among these, the most influential lie in the areas of wireless communication speed and range, personal identification, and device connectivity [83]. Speedy communication allows system interaction to be smooth, and any extension to network transmission range gives users larger areas to work (play) within. The impact of these advancements can be seen in the progress of mobile phones. For example, originally these devices were used solely for telecommunication, however, with advancements in the related technologies, they now provide capabilities such as video phone (3G network) and image transfer.

Transmission speed, however, is not the only factor involved in creating greater functionality in mobile and wireless devices. Other factors include: storage techniques, minute processing capabilities and communication technologies. These factors are generally device specific. However, in all devices, greater processing, storage capacity and the downsizing of components would facilitate several new areas of technology that would govern the evolution of wireless networks.

3.2.1 Related Advancements in System Cooperation

Within wireless networks and mobile environments, announcements, postings and acknowledgments of available services allow devices to discover the capabilities of any network services they have encountered. Current discovery protocols, such as Jini [79, 36, 56, 45, 47] and UPnP [72], along with developing techniques, such as PdP [20] and JXTA [26, 17], allow for the dynamic discovery of services within a network. Some discovery techniques provide communication between ad-hoc devices within the network while others allow interconnection through use of Peer-to-Peer (P2P) [17] ideals. However, there will always remain various restrictions within portable technologies. These restrictions are a result of the lack of storage space and processing speed that continue to exist.

Along with these changes in the mobile industry there have also been several advancements in discovery technologies. The main aim of these advancements is to allow mobile systems to cooperate in a similar nature to standard systems. For example, JMatos [20, 87] is designed to allow restricted mobile devices to utilise Jini networks. There are different versions of JMatos, some developed

for Java applications and others for C++ applications; however, they all give complete use of Jini techniques to their respective systems by placing a compact lookup service on a mobile device. Developments like this are enabling mobile devices to interact with other systems and, therefore, it is no longer necessary to consider them separately. In terms of system connection and discovery, the new distributed developments provide an interoperability among differing types of systems that previously did not exist.

3.2.2 Mobile Device Connectivity

Some ubiquitous devices, embedded systems and other portable devices interact and alter states without need or want of user interaction; for example, autonomous applications for decentralised communication [88]. Moreover, there are other embedded applications which alter their execution in accordance with their environment [125]. In these cases, the devices are connected to others so that they can all work together according to changing conditions. Whilst this type of interaction is a fully automated and independent pervasive system, there is, however, no true relationship between the embedded systems and their environment; there is just controlled reaction to a predicted situation. Some portable and embedded systems can change code according to different situations [64]. This capability provides them with real time data, however, their limited communication and restricted hardware limit their flexibility and overall network cooperation. For example, car systems consisting of multiple embedded devices, each controlling a separate section, are unable to cooperate during unpredicted circumstances [77, 125]. This is a result of each embedded device working separately or with a few selected devices.

3.3 Wireless Communication Techniques

Network communication is achieved using an interactive link: wire, radio waves, power lines and infra-red, to name a few. Within wireless networks, Bluetooth [14, 13] and the 802.11 family [55] have emerged as the most useful and adaptable. In assessing the advantages of any communication technique, the cost, range, bandwidth reliability, speed and adaptability are all major factors. Distributed system cooperation requires reliable, accessible and standardised communication. Consequently, these are also measured as essential requirements for our frameworks communication standard. Only Bluetooth and the 802.11 provide these

requirements within mobile devices. We will now present an overview of each of these communication standards.

3.3.1 802.11 family

The 802.11 family, sometimes known as WiFi, has become the common standard communication for wireless networks [55]. According to the 802.11 implementation, there are different degrees of connection range and speed applied within a device. The 802.11 family definitions, according to [55], are shown in Table 3.1.

Standard	**Transmission Rate**	**Communication Band**	**Communication Method***
802.11	1 or 2 Mbps	2.4 GHz	FHSS* or DSSS**
802.11a	54 Mbps (max)	5 GHz	OFDM***
802.11b	11 Mbps (max)	2.4 GHz	DSSS
802.11g	20+ Mbps 54 Mbps (max)	2.4 GHz	OFDM or DSSS
802.11i#	Added security to 802.11 standard		
802.11e#	802.11a,b with QoS		

**FHSS - frequency hopping spread spectrum **DSSS - direct sequence spread spectrum ***OFDM - orthogonal frequency division multiplexing. # was scheduled for approval in 2004*

Table 3.1: 802.11 Standards

The specification of the 802.11 communication standard is continually evolving. However, as Table 3.1 indicates, each version, other than 802.11a, uses the 2.4 GHz communication band providing backwards compatibility. This level of compatibility is important as it allows devices that are only capable of using the earlier versions to communicate. Of course, unless upgrades are available, each device will only ever communicate at the rate applicable to the standard to which their hardware is designed.

802.11 standards which use 2.4 GHz maintain a communication range of approximately 100 metres although range is highly susceptible to interference from obstructing objects, such as walls. This standard is commonly used when creating a wireless local area network (WLAN), as it has similar attributes to a wired network, yet, allows wireless devices to connect to the network. However, the

speed is slower in comparison to wired networks. WiFi was initially expensive to produce and the hardware was large in size compared to other communication techniques. As a result, its merger into mobile devices, particularly into smaller sized devices such as mobile phones, was slow. However, it is now becoming a standard component of mobile devices, such as iPaq PDAs.

3.3.2 Bluetooth

Bluetooth was initially developed for short range communication. It is able to wirelessly connect devices within a spherical range of approximately 10 metres, on the 2.4GHz ISM band. However, much like 802.11, it is susceptible to interference [14].

In a Bluetooth network all devices are identified via the use of a Bluetooth Identification and Stack [13]. The Bluetooth Stack holds, and periodically updates, references to Bluetooth enabled devices, using their Bluetooth identification. Bluetooth communication devices are generally cheap and relatively small. Their low price and small size have extended their availability, and as a result, the standard has seen enormous growth in the mobile communications market.

Bluetooth is seen as the best communication option for short-range areas. Moreover, its small size has seen it included in many mobile devices. However, Bluetooth is generally not used for large scale networks. As a result, its use for advanced distributed activities is not prevalent throughout the field. This is not to say that it can not be used for such purposes, only that, in general, other communication techniques prove more simple and more convenient to adopt.

3.3.3 Emerging Standards

Wireless communication standards are continually evolving. New standards emerge continually and probably will do so for the foreseeable future. A current example of an emerging standard is the 802.15.4 (ZigBee) communication technique [126]. ZigBee uses much less power than both Bluetooth and 802.11b, however it has a slower transfer rate of 250 Kbps [126]. Like Zigbee, a new standard will offer specific capabilities not provided by other techniques. However, not all developments are of technological capabilities. Standardisation and device compliance is also driving the technologies, particularly as more mobile devices are developed. This trend indicates that communication standards are not exempt from

the current interoperability and compatibility evolution which is seeing the development of programming languages such as those within the .Net framework [74]. Within communication standards, ZigBee is an example of a major development; for example its many different contributing parties include competing technology companies such as Phillips, Motorola and Siemens [126].

3.4 Distributed Cooperation

Distributed techniques make use of remote files, objects or procedures to connect systems. This means that it is important that the means of invocation between systems is well established and clearly defined. Similarly, a clearly defined distribution discovery technique is equally important to our framework. However, before we can determine what systems can achieve using distributed cooperation, we will assess the best means of cooperation. Therefore, we will now look at some standard techniques and determine their benefits. This analysis will also clarify the difference between remote discovery and remote invocation. Table 3.2 shows the areas of these techniques that we will be analysing.

	No Discovery Mechanism	**Discovery Mechanism**
Local Invocation	Invoke a class locally using details from a known location.	Invoke a class locally using details from an unknown discovered location. (***required capability***)
Remote Invocation	Invoke a class remotely using details from a known location.	Invoke a class remotely using details from an unknown discovered location.

Table 3.2: Techniques of Distribution and Invocation

The table also indicates the area of discovery and invocation that we require for our framework: a discovery based mechanism that allows local invocation. The aim of our discussions is to determine if a current technique exists that provides this functionality. We will analyse each of the appropriate techniques according to the categories established in Table 3.2.

3.4.1 No Discovery, Local Invocation

This level of distributed cooperation allows local invocation of an object that is based on the details from a known remote server. Included in this category are methodologies such as Java Applets and Web Start.

This technique does not use any discovery for code transfer, instead it accesses remote code via a URL reference[2]. The use of URL addresses gives implementations of this technique the ability to load program code that does not exist on the client's system. And, unlike object based distributed techniques, they are able to load class files without a representing remote object.

Although this technique does not use remote objects, it must still generate instances of classes that are not available on a client system. Java Applets, for example, use a specific class loader (`sun.applet.AppletClassLoader`) to direct clients back to a server (usually the originating HTTP server of the applet) to gather code. This is illustrated in Figure 3.1.

The client's class loader will download class files as required to create local objects based on the remote byte code [97]. This allows clients to run applications that do not exist on their systems.

A side benefit of this approach is that it gives vendors control over versioning, security specifications, personalised attributes and upgrades. However, there is a lack of dynamism and anonymity which may disadvantage the client as it must know the correct server location to locate class files [102].

[2]Some service discovery techniques, such as Jini, make use of URL references to gather required information from a specified code base, however, this is used in combination with remote services. Therefore, we do not classify them as web based discovery techniques.

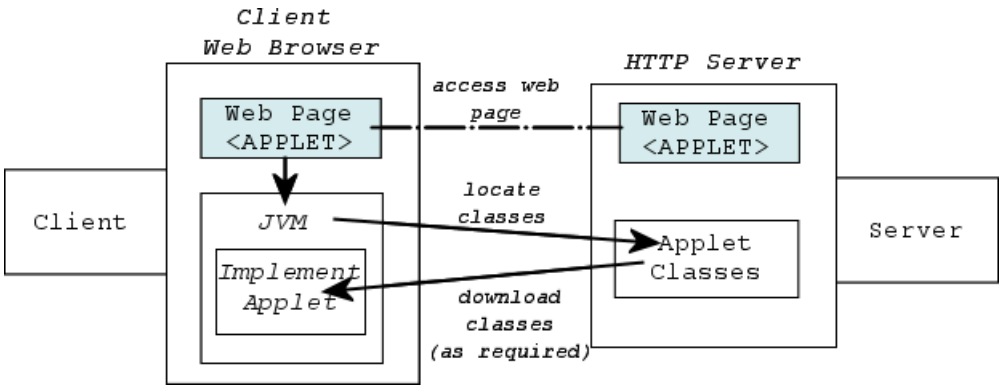


Figure 3.1: Description of a Web Class Loading - Applets

Much like Applets, Java's Web Start gives a client the ability to generate a program via web based links [109]. However, Web Start can run with or without a web browser and is able to generate applications from scratch [109]. Web Start was initially created to remove the confusion that surrounds the installation of multiple applications and their version control. Although, it is similar to Applets it is unique, and there are some distinct differences:

- Web Start handles much larger applications and is not specifically confined to the browser.
- Web Start uses caching to allow the user to run applications when not connected to a network.
- Web Start, when compared to Applets, gives a greater flexibility to its programs allowing for greater control and creativity.
- Web Start needs to be installed on a user's machine before it can be used.

Yet, as with Applets, Web Start still has the restriction of a URL. Although it can cache previously run applications [109], initially, it still must know exactly where to get the application from.

Unfortunately, this technique provides no anonymity to services. Consequently, a remote class cannot be located without the knowledge of its location; a major requirement of our framework.

3.4.2 No Discovery, Remote Invocation

This level of distributed cooperation allows the remote invocation of an object that is based on a reference on a known remote server. The main technique in this category is Web Servers.

Web Services allow system to cooperate through the use of the already established HTTP internet protocol. A web service is the implementation of server applications that can be invoked using SOAP calls over HTTP. The W3C have standardised Web Services and the syntactic elements of SOAP [122]. They define Web Services as:

> *A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web Service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.*

As this definition indicates, the description of Web Services is achieved using a XML based description language, specifically the Web Services Description Language (WSDL). The WSDL provides a similar service to that of CORBA's IDL and RMI's Remote Interface (discussed in the next section) in that it provides the necessary details to describe what the remote service looks like. Although, by using a discovery technique such as UDDI, web services are able to provide a discovery mechanism, in general, they are not applied. This is a consequence of the complexity, poor searching capabilities and lack of complete standardisation of their application to web services [80]. This being the case, for our purposes, we consider web services to have no discovery capabilities. Therefore, the location of the service must be known.

Notably, Web Services have, at times, been seen as the most appropriate distributed technique for common systems. This view derives from the fact that its underlying infrastructure, HTTP, already exists. However, Web Services also rely heavily on their XML descriptions, have a need for server address knowledge, are at times unreliable, and are represented as a stateless service. Again, this technique does not incorporate the necessary attributes for our framework.

3.4.3 Discovery, Remote Invocation

This level of distributed cooperation provides a local reference to an existing remote service. Included in this category are the techniques: Corba, RMI and Jini.

The discovery of remote objects throughout a network has become a standard technique for distributed interaction. CORBA [82], RMI [105], Web Services [122] and .NET [74] are the most widely used forms of object based system cooperation. Remote objects provide a link between a client and a server allowing them to interact through remote invocation. An example of distributed cooperation using remote objects is provided in Figure 3.2.

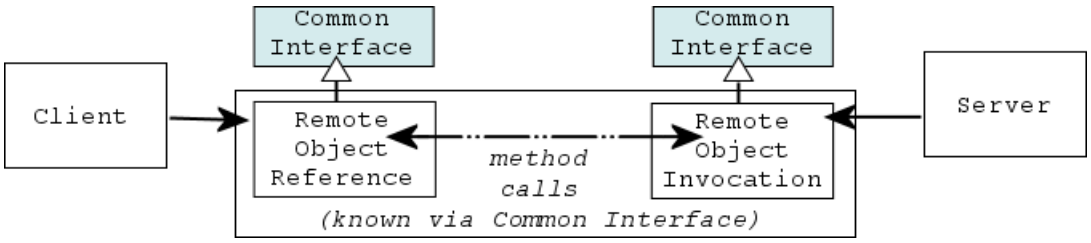


Figure 3.2: Description of Remote Objects

CORBA and RMI make use of similar remote object manipulation to achieve distributed communication[3]. They use naming based discovery of remote objects to link a client to a server. CORBA uses an IDL description class to identify procedures that may be called remotely [82], and RMI uses a common subinterface of the `java.rmi.Remote` interface for the same purpose [105]. CORBA is generally seen as a difficult technique. However, its complex structure is an overhead of its ability to connect multi-language services. RMI on the other hand is simpler to understand but confined to Java systems.

Fortunately, as both RMI and CORBA use object definitions to connect a service and client, techniques that allow them to work together have been developed. For example, RMI over IIOP allows the creation of Java RMI services that are understood by Java CORBA clients [100].

[3]Although, CORBA clients supply the proxy for the server [82], and RMI servers supply the proxy for the client [105], in essence, they achieve a similar task.

Jini [36, 56, 79] is a similar concept to distributed remote objects. However, it uses remote objects as conceptually defined remote services where each object represents an available service or proxy to a service. Jini is an object orientated method of discovering services and objects within a network that aims to eliminate the need for direct knowledge the physical or logical address of any server/service [79]. Therefore, it enables the discovery of services without any knowledge of their whereabouts or the location of the associated registry. This is known as anonymous service discovery or location independence.

Jini's anonymous dynamic discovery of services is extremely useful within ad-hoc connected networks such as mobile wireless networks. For example, it enables a client to discover a system with a randomly assigned address; without the need for systems to know the actual address of a service. Furthermore, as mobile systems are likely to be used within several different networks, Jini's discovery flexibility allows them to access all the networks resources using a single application.

The discovery of services and objects, no matter what technique is involved, requires matching a request to an available remote object or service. This matching process may be achieved using direct matching, such as names and namespaces (CORBA, RMI), or, it may be achieved more flexibly through descriptions of system requirements (JINI). As mentioned, wireless networks have a need for a flexible cooperation. Consequently, the same remains for any discovery technique that is applied.

Discovery techniques that find services using name matching techniques provide searching systems with a perfect match, similar to the matching of a google search string. However, without the knowledge of the exact lookup name a service can not be located. Therefore, although this technique is exact, inherently it gives no leeway for simple searching errors or a lack of knowledge of the service being searched for. This restriction is most limiting if a client has a need to search for a 'type of' service where the exact service details are unknown.

The discovery of services based on a list of needs provides flexible lookup [36]. This type of discovery technique allows a searching client to find a 'type of' service by specifying required methods. Jini is one distributed technique which makes use of this type of service discovery. Therefore, a wireless network is likely to benefit from Jini discovery techniques as it is able to handle ad-hoc movements. To further aid the use of location and physical specific services, such as a toaster,

printer or coffee machine, details of their physical location can be included as part of their searching information.

Although, most existing distributed techniques seem to be beneficial to all system cooperation, they are constrained by design. For example, within Jini and RMI, an object, class or proxy service that is advertised for remote access must implement the `java.rmi.Remote` or `java.io.Serializable` interfaces somewhere within its structure [79]. Furthermore, a subinterface of this interface must be known to both the server and the client [79]. Only then is a remote object able to be discovered and manipulated by each client.

Remote objects and services are ideal for remotely invoking sections of a server that exist for the use of clients, and the sharing of information. Moreover, by specifying the methods of a remote object, using its common interface, the processing that may be performed by a client can be established.

However, there are several areas that these techniques do not cover. For example, a client may wish to run a process using a remote object yet wants the computation to be performed without the knowledge of a server. This case represents a situation where the client creates an object that has no reference to an object on the remote server, even though the associated byte code is discovered from that server.

We can see why this limitation is present by looking into the way remote objects and services are transferred. Several distributed systems use marshalling for object transfer, and consequently, there is no passing of byte code unless it is associated with a remote object. In this case, when a client discovers a remote object it is in the form of a marshalled object. This then points the client to a server for further required details [36]. This technique is, therefore, unable to create an instance of a class without having a reference to at least one existing object on the server side.

The ability for remote object applications to create instances of remotely based classes is a result of stub files. However, the creation of the remote class instance is still achieved using a class loader [65]. In the case of RMI and Jini the `java.rmi.RMIClassloader` is used. The RMIClassloader is designed to load classes using the codebase URL reference that is stored within a discovered marshalled object. It uses this information to find the class bytes of a remote object

and generate a class instance. The class instance is then merged with the serialised object information from within the marshalled object to establish the remote object representation [36, 97].

Other than limiting the applications to a single language, this limitation of marshalled objects and discovery services is generally not noted as a disadvantage. For example, Jini applications are usually designed to make use of this unique distributed flexibility and discovery technique, but not to manipulate the loading of remote class files. Unfortunately, for our framework, this technique does not provide the necessary local invocation. However, as we will now discuss, in combination with proxy techniques, this technique may allow the discovery and local invocation of objects.

3.4.4 Discovery, Local Invocation

This level of distributed cooperation allows a local service to be invoked on discovered remote information. This is the level of cooperation we require for our framework. Unfortunately, no direct technologies exist for this level of cooperation. Therefore, we now suggest the base component of a solution and extend on this in Section 3.6.

The previous section discussed discovering remote object and invoking remote objects. However, what we have not yet discussed is the concept of using remote objects as proxy services. A proxy service is a remote object that allows a client access to server side processing. When using a proxy service, application processing can be achieved on the server and results can be passed back to the client. To achieve this, a proxy service object must implement a specified common interface(s), and once again that interface must be known to both server and client. Figure 3.3 shows how remote objects can be used as a proxy to a server.

We believe that using this type of structure may allow the discovery of class details that can be locally invoked. Before we detail how this can be achieved, we will continue our analysis of distributed methodologies.

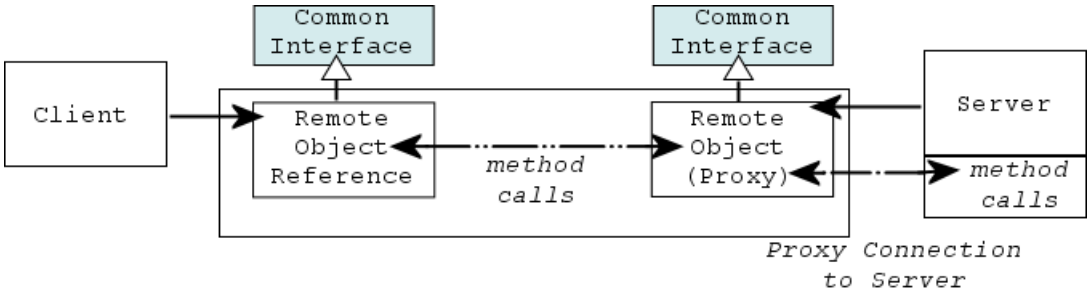


Figure 3.3: Description of a Proxy Service using Remote Objects

3.5 Mobile Agents

Prior to describing our solution to distributed cooperation using discovery mechanisms for locally invoked objects we wish to discuss Mobile Agents as a unique technique of distributed cooperation. We discuss these to ensure that our analysis is comprehensive. Mobile Agents were developed to enable mobile applications to gather details from a remote server by dispatching an entire program; for example, Java Mobile Agents using Aglets [62]. When using Mobile Agents, both the client device and the server device must provide room for agents to be executed: the mobile agent environment. Therefore, most mobile agent applications are written for interpreted languages such as Java byte code. However, this does provide benefits such as heterogeneity and the use of sandbox security [22].

In terms of distributed cooperation, mobile agents are designed to give a new capability to a system. They aim to provide the system with the ability to execute applications on other devices that they may not be able to process themselves. This gives mobile agents two unique abilities not present in other distributed methodologies:

1. A client is able to run an application (mobile agent) on a server, or other client (in a peer-to-peer situation) and obtain all responses.
2. A client is able to distribute a mobile agent and then leave the network. They can do this as a result of their ability to return to the network later to gather the responses from their agent's execution.

This level of code exchange is of particular interest for our work as it enables mobile devices to run applications that they could not previously run. However, in terms of distributed cooperation, the technique possesses no real advantages

in comparison to other distributed techniques [22]. Moreover, our studies determined that, as a result of the disconnected distributed connectivity that the technique uses, a server, at times, may lose valuable processing as a result of agent use (although the agent environment setup should disallow such a situation). For example, Figure 3.4 depicts an agent being implemented within the servers agent environment.

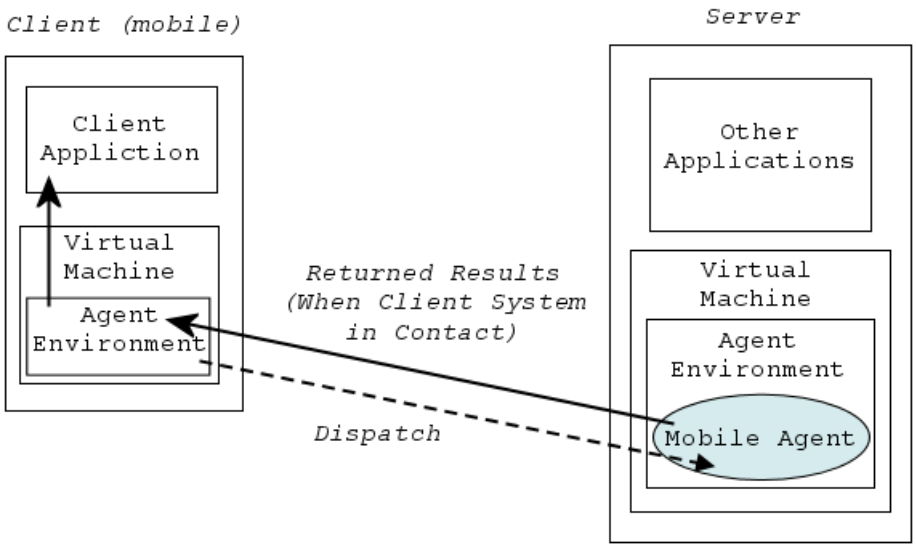


Figure 3.4: Mobile Agent Execution

During this implementation, the client is processing very few, if any, tasks that are related to the agent. This means that the server is doing all the client's work. In a resource sharing network, if an advertiser of a resource essentially processes all accessing clients, then the advantages of resource use may seem to be outweighed by the negative aspects of network interaction. Consequently, the server (advertising system) may choose not to participate.

Although, this is not always seen as a disconcerting attribute, and has actually been used advantageously in context awareness, for example, REDMAN [9]. We must still reject its use for community cooperation as it does not provide the two-way resource access that is required.

3.6 Remote Anonymous Class Discovery

This chapter has so far discussed the methods that can be used for a program to be initiated and executed via distributed techniques. There are many standard uses for distributed techniques, such as client server applications, however, there are many other more advanced applications of its use. For example, OSGi [84] systems give devices, mainly home based devices, the opportunity to access information from within their home network. The connection of systems within the network may enable them to configure their internal structure via other systems. The information gathered from the network can then be used to alter their behaviour. For these changes to occur, details in the form of files or system attributes must be located in an accessible section of a task based services [84]. Typically, these are: database information, or at times information on context such as temperature. We pose the question:

What are the benefits if these resources are system byte code components?

We have seen so far, that, remotely accessible services enable systems to gather information, generate programs, access server data and cooperate throughout networked environments. However, we have also found that there is no current technique which allows localised invocation of objects based on discovered information. This is imperative as our framework requires a class loading structure that expands the reach of a JVM beyond that of its host system, yet, does not restrict itself to knowledge of addresses.

In the previous chapter, we discussed how Java class bytes are normally loaded into a JVM through the use of a class loader. In this chapter, we discussed this further indicating that the class loading can be achieved via Java's standard bootstrap loader or a specialised loader such as URLClassLoader, AppletClassLoader(s) or RMIClassLoader. Many of these specialised class loaders instruct a program to gather Java class bytes from specified places or in a particular fashion; for example, we discussed the attributes of the AppletClassLoader. However, our previous question of *whether or not class bytes can be discovered anonymously via a distributed connectivity, and initialised locally* remains unsolved.

To determine which of the system cooperation techniques may answer this question, we can categorize them into two distinct types:

- Direct Remote Access (Web based techniques).
- Indirect Remote Access (Remote Objects and Agents).

We determined that each of these methods is developed for specific reasons and use: indirect remote access provides a flexible means of finding a service, whereas direct remote access is designed to enable thin, minimally connected clients. Unfortunately, none of these techniques are able to construct an application, or a section of an application, from class files that the application does not have, nor knows it will need to discover.

The anonymity essential for our remote class discovery eliminates all direct remote access techniques as they require direct URL referencing knowledge. Regardless of this, these techniques remain the only methods that exchange code for direct implementation of a class. In order to hide this technique directly we suggest burying them in a local class loader. This class loader will be able to contact a remote class loader to gather class details and avoid the need for remote objects combining both discovery and local invocation. This could be achieved by using the remote proxy service structure we described in Section 3.4.3.

Adding to this, as we indicated in the previous chapter, Java is the most common language for mobile devices, and therefore, a distributed technique designed for this language is most appropriate. Therefore, we see the discovery technique Jini as the best fit for our solution. We will now demonstrate how this technique can anonymously discover class details, and we will discuss its use in our framework in Chapter 6.

Figure 3.5 illustrates how a Jini service may be used as a proxy to a remote class loader. The common interface of the Jini service will determine the degree of access a distributed system has to other systems. (This will be demonstrated throughout Chapters 5 and 6.). The interface would provide the service with access to a procedure which can pass the byte code of a specified class. This byte code can then be used by the discovering system to invoke a local object in a similar manner, for example, Applets. The exact implementation of this is discussed in Chapter 5. At this point, we only wish to describe the concept.

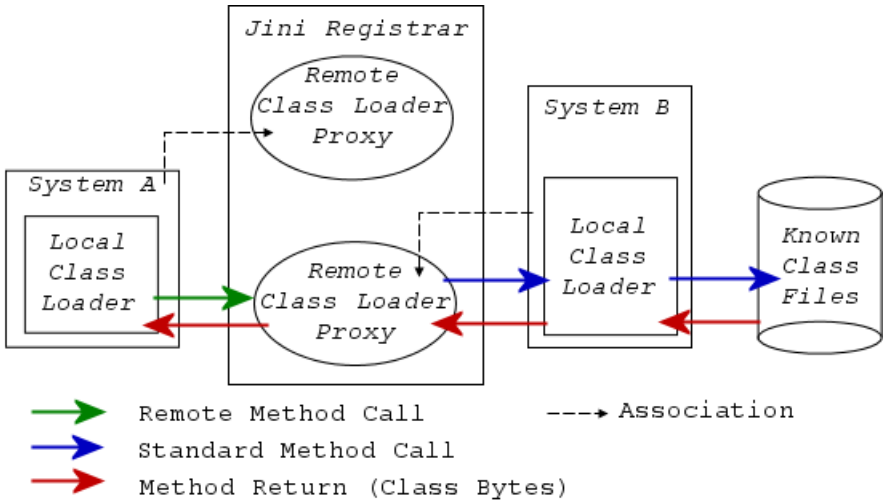


Figure 3.5: Discovery of Class Details Using Jini Proxy Services

3.7 Summary

This chapter has discussed many different techniques of distributed cooperation. It suggests that all techniques have one core component in common: they all allow a client to access information on, or from, a server. For our framework we used this analysis to determine if there is a specific technique that allows remote anonymous code sharing. However, our research indicates that no exact method is specifically designed for code transfer in a sharing manner. We therefore suggested a solution.

We demonstrated that some discovery techniques, such as Jini, provide anonymous interaction but directly lack code transferring flexibility, while other discovery techniques allow for code transfer but are not anonymous. However, we suggest that a class loading structure based on Jini proxy service could be used as a means of two-way code transfer. How our solution relates to the current distributed cooperation techniques, as found during our discussion, is shown in Table 3.3.

In the previous chapter, we concluded that there is such a large range of dynamic updating techniques designed for Java applications that most systems have the ability to include one of these techniques if so wished. These two conclusions

	No Discovery Mechanism	**Discovery Mechanism**
Local Invocation	Java Applets Web Services	**Suggested Solution:** using discovery techniques, such as Jini, to create remote class loader proxies.
Remote Invocation	Web Services	RMI CORBA Jini

Table 3.3: Distributed Cooperation Techniques

raise a further question:

What can be achieved when dynamic updatable systems have the ability to transfer and gather code remotely and anonymously?

This question is directly related to our second question in this chapter (Section 3.6):

What are the benefits if these resources are system byte code components?

The following chapter will discuss these two questions using concepts in localised dynamic communities and adaptable systems.

Chapter 4

Dynamic Communities

4.1 Introduction

The focus of this chapter is to analyse the concept of a dynamic community. In doing so, we will also determine the attributes of the systems acting as community members. Essentially, the chapter derives from the concatenation of the analytical conclusions from the previous two chapters concerning the use of dynamic updates and distributed cooperation for our work. Consequently, during this chapter we will return to several previously discussed areas and discuss them from a different viewpoint and within a different context, examining them from the perspective of dynamic communities. To aid the understanding of the concepts that arise during our discussion, we include an analysis of other research areas that are of a similar nature, or that we perceive as applicable.

Any particular definition of a community is difficult to grasp within the full context of system interaction. Therefore, initially, we clearly formalise the meaning of a community from our point of view.

In sum, this chapter will provide the final conceptual details for the DUPE framework and suggest a technique to achieve dynamic communities and adaptable systems.

4.2 Defining a Community

Protocols, such as TCP/IP, describe to systems how to send data, but they do not define what to send or why it is sent. These aspects of transfer are achieved by communication protocols. However, our research in Chapter 3 into such technologies, for example distribution techniques, indicates that system communication is never straightforward.

We demonstrated Chapter 3, that a system can download information, or make use of remote objects, to establish a meaningful relationship. The most prevalent of these techniques is a client-server relationship. This relationship allows cooperation by enabling systems to share information, objects and services across networks. To do this some protocol is required. Consequently, the normal function of a service is constrained by this protocol, and as a result, all such communication is restricted in type and content. This can be seen as *controlled* system interaction. We want to examine a broader concept of system cooperation; one in which components have individual roles and the result of any communication is incidental of these roles. To achieve this, we suggest community cooperation.

A community is a flexible interaction between different entities; each entity is a member of the community. The present definition of a community provided in wikipedia.com [121] is:

> *A community is a loosely coupled collection of individuals where the interaction is informal and spontaneous rather than procedurally formalized, an end in itself rather than goal-oriented.*

Within a community, the reactions and interactions of each member are a result of what information, or capabilities they may acquire from the other members. For example, in a home community the fridge may be able to synchronize viewing stations with the TV as a result of details found in a new DVD player. In this case, the TV asks the DVD player what it can do. The DVD player replies that, among other things, it can synchronize with the TV. The fridge likes that, so asks how it is done and then applies the technique. Some other more complex examples may include: the learning capabilities of artificial intelligence (AI) techniques, or the reaction capabilities of spawned agents. In each of these examples, the members grow, or learn, concurrently with the combined knowledge of the community. It has been demonstrated that the dynamic cooperation established

among these techniques can be used for knowledge collaboration [123].

We have explained in Chapter 3 that, within a client server situation, the communication is precise and meaningful; for example, a statement such as: *'Run this method on your object',* or, a question such as: *'Do you contain this particular service?'* This type of communication protocol directs the response to be specifically tailored. In the previous statement, the response to the statement will always be the designated return type, and the only answer to the question must be a boolean response. We conceive that the communication within a community is an open ended query(s) that does not require a precise response. Some examples include: *'What did you just do?'*, or, '*What do you know how to do?'* This type of communication has unknown response ramifications. For example, a response may be an array of object types, or, as is the case of the second question example, the return of a specific object type that can consist of information with unknown implications. Using this protocol, a community establishes communication based on random and unknown responses, rather than a set of common questions and answers.

The previous discussion of community broaches the topic of 'community'. The research area of the behaviour of emergent systems has defined 'environment' in terms similar to what we are considering for a community [34]. However, their definition is complex, as its intelligence is based on its encompassing sub-systems; the primary task of these systems involves interaction, whereas, in our concept, a community is separated from the tasks of the system. Even so, it seems that, within both definitions, communication involves uncontrolled system interaction, or more specifically, uncontrolled system *reaction*. This unique attribute has yet to be harnessed as a system cooperation technique without associating it to the task of the systems.

We see that it is therefore evident that a community is a gathering within a network where the *controlling entities* are the *community members* and not the underlying network protocols.

4.2.1 Identifying Dynamic Communities

The view of a community that we have now established comprises two factors: *unknown members* and *unknown messages and responses*. However, this is only

a theoretical view of a community and in reality only the use of Artificial Intelligence (AI) techniques, such as those applied within multi-agent systems [119], would allow systems to interact with no previous knowledge about the other members. In fact, without the knowledge of what another member may look like, there is an extremely low possibility of a system joining a community at all. Therefore, a system which wants to become a community member must have two things in common with the other members of the community:

- the system must be able to be seen as a community member, or possible community member and
- the system must be able to understand the community language of communication.

This is not to say that the communication within a community needs to be stringent and highly specified. It indicates only that a member must be able to talk via the same means as the other members in order to successfully communicate within the community; the communication content remains variable. We believe that it is this communication flexibility that will enable dynamic systems to establish a specialised type of community. As each system in the community represents a community member, and, as each member has equal opportunities within the community, each member has the ability to affect the overall status of the community. Moreover, systems are free to move in and out of the communities, adapting as they go.

This level of a community produces a dynamic community.

Furthermore, as each member is a dynamic system (one able to alter its runtime state, as discussed in Chapter 2), the state of a community will continue to alter as a result of any change within a member. This is a direct result of dynamic members altering behavior based on other members. Considering the continual movements of mobile systems through communities, such as that illustrated in a park analogy that we are about to introduce (see Figure 4.1), community states will alter continually as new members enter and old members leave or change state. Together with the findings from Chapters 2 and 3 this provokes the question:

Will a gathering of systems, all with the capability of sharing structure (code) resources and dynamically altering their runtime structure, generate, in a section

of a network, a dynamic community?

To respond to this question, an understanding of a community's communication language and the attributes of its members is required. To begin, we develop a detailed view of a dynamic community.

4.2.2 Painting a Picture of a Dynamic Community

To further understand the concept of dynamic communities we now provide an analogy as shown in Figure 4.1.

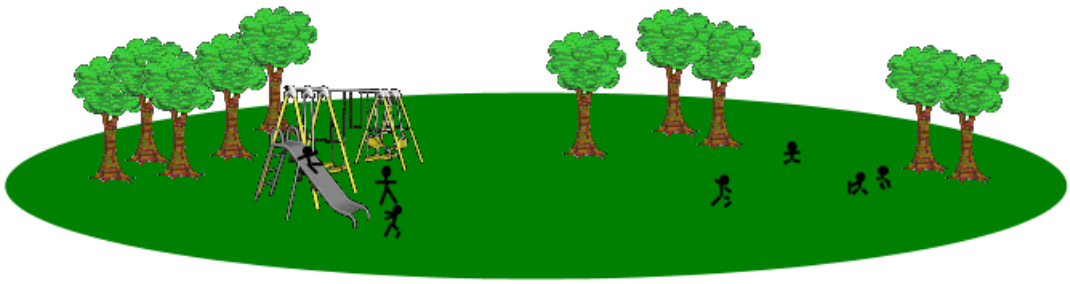


Figure 4.1: Community within a Recreational Park

In this analogy the physical area of the park is the community structure and the people within the park are the community members. There are several important patterns of behaviour that arise from the member interactions within this situation:

- the members are in no way confined to the community;
- each members has the ability to alter their behaviour based on the state of the community and the state of other members. For example, if it begins to rain a member is able to find shelter under a tree, or, if a member encounters another unknown member they are able to choose whether or not to communicate. In both of these cases the reaction is a new behaviour. That is, before entering the park the member did not know to run to a tree if it rained, this knowledge and the resultant action was obtained from community interaction;

- other than in circumstances where an individuals behaviour threatens another member, a community in no way restricts the behaviour of members;
- members are able to communicate and interact freely within the community;
- there is no requirement for the community to monitor, limit or define communication requirements. That is, the people may communicate in any manner they please within the park. Of course, whether or not the behaviour is irresponsible is assessed on an individual basis.

The behaviour patterns present in this scenario are features that can be identified in all communities. Such communities are established in true life and created through a gathering of people. The communication within the community can be seen as the process which enables members to inform and learn from each other during conversation(s). Moreover, as the community develops over time, each member will understand more of the information that exists within the entire community. This learning process is a communal learning process. A communal learning process is one in which each member continues to either add information to the community, and/or learn information from the community. Such is the interaction within dynamic communities that the shared information is not necessarily just facts. The information may be behavioural (how to achieve something: shelter under a tree), community specific (how the community achieves a task: directions within the park) or even an update on past knowledge or behaviour (how it now works: how to use the new equipment within the park).

It may be derived from this analogy that the communication language within a community is relatively unspecified. However, for a dynamic community consisting of computational applications using tightly prescribed protocols, this situation is never likely to occur. Therefore, we will now detail how an appropriate level of community communication can be established.

4.3 Dynamic Community Communication

The nature of community communication lies within open ended query, and this comprises a combination of common communication protocols. Subsequently, any definition of the level of interaction required for dynamic communities to exist will form from a simplistic communication base and build to a non-complex subsection of a complex communication protocol. That is, without the underlying

structure of a simple protocol, and the discovery capabilities of a complex distributed technique, dynamic community communication is difficult to establish.

We have determined that, community communication requires a simple commonality protocol for interaction. This is similar to how TCP provides a common definition to establish a means of data transfer. However, unlike such low level communication protocols, which only describe how to send information, dynamic communities must also know how to understand the information. Moreover, irrespective of member discovery and community resource usage, the communication protocol must provide a level of commonality among all members.

If we return to the home example from Section 4.2, we can visualise this level of communication. Initially, each individual appliance has a specific task; the fridge keeps food cold and the oven cooks food. There is no communication. The introduction of a simple communication to this scenario allows the appliances to communicate, however, there is no meaning to the communication transfer. Currently, the next step is complex communication; for example client server protocol. In this case, the appliances share information via task oriented queries; for example, the fridge will send cooking instructions to the oven when asked. What is required in a community is a communication that sits in-between simple and complex. Although, less intricate, it needs to remain descriptive enough for system cooperation. For example, an oven can be taught by the fridge to understand complete recipes instead of the fridge continually sending individual instructions. Therefore, if the home establishes a community, the communication among appliances will be according to what they wish to achieve.

Interestingly, for system connectivity, this level of communication relies on a complex communication technique, such as distributed cooperation. We identify that community communication is a new communication level that lies between simple and complex communication. The communication levels as we have established are presented in Table 4.1.

The creation of the community communication level is, however, an intricate task. There must be a level of simplicity combined with a complex overhead layer. We will now discuss how the complex layer of distributed cooperation can be combined with a simple single question that induces the response of a usable code resource, enabling the establishment of community communication.

Communication Type	**Structure**	**Examples**
Simple Transfer	*No Structure*: this level only defines the appropriate manner to send data	TCP/IP. UDP
Community Communication	*Exploitive Structure*: this level is a general question level. Although the number of questions is limited, the replies are varied.	(Example queries) What can you do? What do you know? What do you eat?
Distributed Cooperation	*Task Structure*: this level provides a structure to be able to ask specific questions with specific replies. It is a very tight structure.	(Example queries) What is the time? Where are we? Do you like Pizza?

Table 4.1: Levels of Communication

4.3.1 Generic Communication Response

The specification of community communication states that it does not ascertain a precise response. However, for system cooperation to be maintained the response must be understandable by all members within the community. Therefore, the problem is:

How is it possible to establish community responses when a level of commonality within the communication must exist for it to be legible?

In answering this question, although the uniformity of the responses required for community communication is difficult to establish, there are methods that may allow it to occur. However, before determining the best response we suggest that a single communication question should be established. We have already suggested questions such as 'What did you just do?', or, 'What do you know how to do?' The concern is that there are many different types of communities based on the types of questions that can be asked. For example, asking 'What do you know how to do?' leads to communities such as typical discovery communities (for example, Jini) where discovery is followed by task interaction. Instead, we

ask a different question, leading to a different type of community. Our question is *'What can you teach me?'* or, *'What can I borrow?'* This will lead to a particular type of community. And, we further this to suggest that the answer for system based communities be in the form of code components. Moreover, in contrast to the semantic annotations of methods, we deliberately do not identify *exactly* what the code will do. For example, the code may achieve a general task but is created according to its current context. This statement holds true through the rest of the thesis and we will make reference to it when appropriate. We will expand on the full details of this type of community in Section 4.5, however, before we do so we provide general discussions on community members in Section 4.4.

This communication suggestion is indicative of the required aspects of a dynamic community that we established at the beginning of this chapter. Moreover, as can be derived from Chapter 2 and 3, it demonstrates that a standard language is the most suitable means of establishing the system commonality and interoperability in dynamic communities. For example, a community based on a communication transferring standardised Java class details, such as the common standard J2SE 1.4, can be utilised by a variety of systems. This includes, mobile systems, server applications, desktop systems and possibly some embedded systems. Moreover, this is a system based community and as a result it maintains different characteristics from those found within a human community. One main difference is understanding knowledge. In a human community learning is a process in itself; it takes time and practice to know how to do a task. However, as system tasks are defined by code, knowledge is instantaneous; as soon as a system is given knowledge it can use it.

This level of system interaction is a technique yet to be explored in the field and is the principal topic of this thesis.

4.4 Community Members

A simplified version of our established definition of dynamic communities is: *dynamic communities are a convenient means of dynamic system cooperation at a particular point in time.* More importantly, we can determine that their usefulness is dependent on its members. Consequently, we suggest that a productive community is one where members frequently gain from their interaction. In order for this to occur, each, or most, members must continually change state. We

use the term system adaptation to identify this change. If members frequently join and leave, then a community must be extremely flexible in terms of member handling and control. Furthermore, as the members determine when they enter or exit, their adaptation capabilities determine the flexibility of the community. Here again it is evident that members themselves are governing the community.

4.4.1 Mobile Community Members

To extend this discussion, we move our attention to the devices containing the member systems, as member movements throughout communities are essentially controlled by the devices in which they are being executed. The movement of mobile members is likely to be continuous and random, as a result of human movements; consequently, it is necessary that each system has individual control over its community entry and exiting reactions. For example, if a member leaves the community they may, or may not, take away the new information that they have gathered from the community. Moreover, their departure should have little, if any, impact on the other members of the community. There are several different movements of systems through a community. These are presented in Figure 4.2.

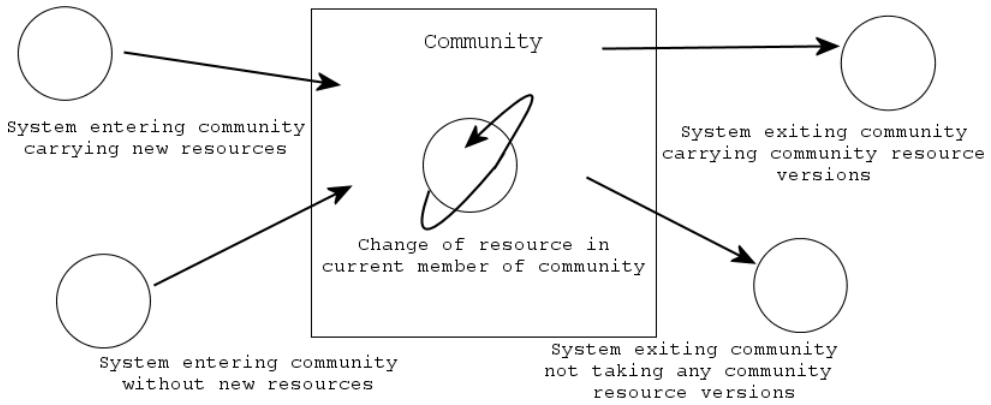


Figure 4.2: Movements of Systems throughout Communities

The movements are the causes of systems' states change through system adaptation. However, they do not determine how adaptation is achieved. For example, at times, members may be stopped during communication, or left looking for another system from which to gather resources due to the departure of a community

member. In such cases, the community as a whole, should continue as normal as long as some members still exist. In the same sense, the entry of a new member will have little impact on the community members. The entry of a new member could in fact be a positive gain for the community. The new member may bring new information with them that they are willing to contribute to the community. Here, not only will the current community members gain from the involvement of the new member, but the community as a whole.

The movements of members define several key factors that facilitate the generation and stabilisation of dynamic communities. The following is a list of attributes that each member within any community must be able to carry out without hindering communication and the cooperation process:

- discover other community members
- access the resources offered by other community members
- be seen as a community member
- offer their resources as part of the community.

With these attributes, members contain all the required qualities that allow them to establish dynamic community communication. These attributes enable members to *enter* and *establish* communities, *learn from* their interaction within the community, *contribute* to the community, and *exit* safely from the community. These attributes in terms of systems interacting within communities are:

- systems establish, discover and join dynamic communities;
- community communication and information transfer between members, must induce the adaptation of dynamic systems;
- members can leave a community without interrupting the execution of other members;
- dynamic systems are capable of generating communities wherever they see it is necessary or beneficial.

This list of attributes comprises the specifications that each member of a dynamic community should hold. These attributes are most important for the mobile members of communities. We apply them to the DUPE framework.

4.4.2 Movement of System States

A dynamic community should allow systems to move and adapt as they wish. A community has access to shared resources, such as printers or code libraries. An entering system can share these resources and also bring in its own resources to be shared by others. In particular, code can be shared. Furthermore, in any community there may exist portable devices which a user is likely to continually carry with them, such as a PDA, mobile phone or laptop. The probability of this type of a system discovering many different communities is high.

As previously mentioned, to get the most from a community a system should be able to adapt to the current status of the community. This is achieved by ensuring that as a system encounters a community it is able to access all the available resources. Furthermore, it is also possible that an entering system will carry into the community a more apt version of a specific resource; for example a system upgrade. In such a case, the already current members of the community are likely to adapt to the system state the new system has brought along. If this occurs, the entire community state may be altered. Furthermore, the movements of systems throughout communities may be unpredictable. As a result, over time a mobile adaptable system might travel through all communities, potentially spreading common resources.

This is achieved by ensuring that when a system encounters a community the possibility exists that they will be able to access the same resources that every other member uses. In turn, members can alter sections of their code that depend on a commonality among the other systems within the community; for example, communication protocol files, location specific code or enhanced versions of current system code.

This attribute is unique to the mobile members of dynamic communities and is the basis for the concept of a physical mobile agent, or a *gypsy agent*, which will be discussed further in Section 4.4.3.

As some members will maintain the resource state of their past communities as they move throughout other communities, a single state is likely to spread through many different communities. Currently, this concept is only used by viruses and file sharing systems. We have a wider more positive view. To identify the requirements of systems to become gypsy agents, we will detail some of their

simple attributes then describe the techniques which enable such attributes to be present. We will continue our analysis of gypsy agents from a wider perspective in Chapter 12.

4.4.3 Gypsy Agents

As a result of their direct relationship with human movements, gypsy agents move in much the same way as a human gypsy. During their travels, gypsies gather and give information. For example, the human gypsy learns how to achieve tasks such as cooking particular foods or understanding information communicated in a previously unknown dialect according to the standards of each community they encounter. The transfer of structure that gypsy agents achieve is also related to other human examples including the transfer of trust from friend to friend and business to business trust transfer relationships.

4.4.4 Standardising Gypsy Agents

To determine how gypsy agents might be established we will now look the nature of mobile systems. Specifically, we take into consideration the interoperability of heterogeneous mobile system.

Most mobile devices provide connectivity via technologies such as Bluetooth, GPRS, WiFi, WAP or some other medium. These communication capabilities, as we have already discussed in Section 3.3, should, theoretically, enhance device usability and interconnectivity. However, we have continually pointed out that the interoperability of systems is flawed in that it restricts the degree of system interaction allowed. For example, mobile phones are simply able to communicate with each other via telecommunication, however to exchange information, or play a multi-player game, there is either a specific need (as a generalisation) for all devices to be developed by the same manufacturer (in many cases the same model). This restriction in interoperability is a direct contradiction to the needs of dynamic communities.

Fortunately, this flaw has not gone completely unnoticed by some developers. Techniques aimed at providing device flexibility, or countering a lack thereof, have been supported by manufactures and software developers. They are generally achieved using three main techniques:

- **Standardised Operating System**
- **Standardised Programming Environment**
- **Common Communication Protocol**

We will discuss each of these in detail in the following sections.

4.4.5 Standardised Operating System

Operating systems, such as LinuxOS, VxWorks, QNX, MSDOS and Symbian are designed for embedded systems, and accordingly, have specific characteristics. Symbian [111], a standardised limited operating system (OS), is able to be implemented as the OS for several mobile devices, including mobile phones, PDAs and PDA-phone devices. Symbian's communication and interoperability techniques adhere to standardised formats, and therefore, the installation of Symbian as the operating system of a device is beneficial. Symbian allows for connection between separate devices, including those from different manufactures, including different models by the simplification of the transfer of information, voice and data.

Currently, Symbian is installed within a number of devices (see the Symbian home web page [111] for a full list). This list is growing, and once it is applied to more devices, mobile interconnectivity may be simplified. However, due to the competitive and lucrative nature of the embedded OS industry, a single OS implementation is unlikely, as is any significant co-operation between existing OS designs.

Other techniques of interoperability and adaptation have been established. For example, many operating systems achieve updates trigged at startup. Microsoft and Linux operating system upgrades require hard-coded addresses of update hosts. It is not in their nature to allow a discovery based mechanism. Furthermore, Version control techniques for file management, such as RCS [115] and SCCS [91], represent a dynamic control mechanism. Although none are specific to automated evolution of a system, they do encompass the ability to maintain versions of a component. Mobile device upgrades represent most closely the changes that can occur with a mobile device. For example, a firmware upgrade to a mobile phone can enable it to achieve faster computations, yet still below all restrictions set by the hardware. However, adaptation techniques for

both operating systems and version control are specific methodologies and are not representative of the attributes required for dynamic systems, therefore, we will not provide an extensive analysis.

4.4.6 Standardised Programming Environment

A common communication language will establish system standardisation. Most commonly this is achieved through the use of a standardised Virtual Machine (VM), such as the JVM. The Java programming environment standardises communication by equipping devices with a common programming and communication platform. Although, there are several differing versions of the Java specification (as can be defined by Virtual Machines and corresponding APIs) the underlying goal still remains the same: commonality and adaptability [43, 65]. We have already established in Chapter 2, that Java's machine independence gives it platform independence. This independence is necessary for the dynamic community. This can be seen through the deployment of distributed systems running over multiple platforms, for example a Jini application [79].

We have already discussed the interoperable aspects of the Java language in both Chapter 2 and 3, therefore, we will not extend this discussion further.

4.4.7 Common Communication Protocol

Other means of communication standardisation include the creation of a communication protocol such as the XML Meta Data [11]. Common protocols are used as communication standards to produce a separation of concerns approach to the system and the communication of a mobile application. However, a necessary requirement is that device-side code be implemented through whatever means possible. Many different languages are able to understand particular communication standards, and are therefore standardised. In these cases, program semantics are not precisely detailed and the communication details any information for the systems to interpret. This allows information to be transported from device to device; however, it does not specify exactly how it should be handled. Specific formats may incorporate specific programming techniques, as is the case with [11] XML format. However, there are no specifications to say that we cannot find the XML data and interpret it in a separate manner.

An example of such an implementation was given by Bellavista, Corradial, Montanari and Stefanelli [11] who created standardisation through middleware and a clearly defined Meta-Data (XML), termed *Columbia*. Flexible in its approach, Columbia is otherwise restricted by its own methodology and use of context variables. Concerned with security and plausible reasons for dynamic change, a Columbia enabled system must detail a specification for update, which in turn, represents a need for programmer interaction. The restriction shown by Columbia is similar to that of context aware systems in that they require a system to know how to react before encountering context variables.

We see that system adaptation (the ability to incorporate any program into the middleware) should not be limited to the constraint of the system but should be based on the flexibility of all systems forming the environment. The importance of simplicity when programming the target system is essential in this concept, in that the less a programmer has to consider about the updating requirements of a system, the more efficiently they can generate the program as a whole.

4.5 Middleware based Community Architecture

We suggest that it is evident that due to the dynamic updating nature of dynamic community members, the use of a standard language would prove more adaptable for system cooperation. We also acknowledge that the use of a common protocol would enable each system to utilise the communication as they wish. However, our definition of a community system interaction comprises, or is the majoring factor, in the act of generating a community. Therefore, a well established method of system interaction and adaptation generates a well established community. Moreover, the flexibility of system interaction and co-operation determines the use and adaptable properties of the community. The degree of flexibility within a community must not be such that the community becomes unstable, yet system interaction must still be kept at a level at which tasks are achievable, and the zero configuration of devices is possible. The separation of these two properties of community flexibility is difficult and dependent on the methodologies applied to the concepts of the system interaction.

Middleware has been applied to establish adaptable software [71]. We see that a dynamic community can also be generated through the use of system side version of a software middleware. The members within a community consist of systems

running within a compatible middleware. The communication between the members is irrelevant to the running of the actual system. All information gathered by, and from a system by a middleware is based on the progress and needs of the system itself, and in accordance with the community. Code segments available from the community will not be gathered by a middleware if they are not appropriate for the system. The appropriateness is determined by class structure dynamic settings.

As already discussed, all possible members of a dynamic community exhibit several characteristic:

- ability to discover communities
- dynamic adaptability according to the community
- ability to share their resources (code) throughout the community.

The middleware structure provides these aspects to systems. Furthermore, when designed as a framework, any compatible middleware (a middleware which adheres to the specified design of the framework) can provide the community attributes to a system.

We established earlier that community communication includes a discovery component. Each member can discover community members and regularly check for needed code or replacement code. In doing this, members access an up-to-date reference to another member, not the member itself. This reduces latency and overuse of bandwidth by connecting to members only when necessary. Furthermore, each member continually maintains their own reference within the community, allowing other members to listen for update events to occur and check the new details. Consequently, all members can maintain appropriate code within their systems without having to contact other members until download of code is necessary.

The cooperation of the members is controlled by the discovery technique. What each member decides to do with the community resource is an individual choice. However, to generate a dynamic community, it is specified that each member have the minimal capability to dynamically re-define their class structure. This enables them to contribute and ameliorate in accordance to the current community. Figure 4.3 shows that each system may be different in nature, yet, still able

to cooperate and gain from a community structure.

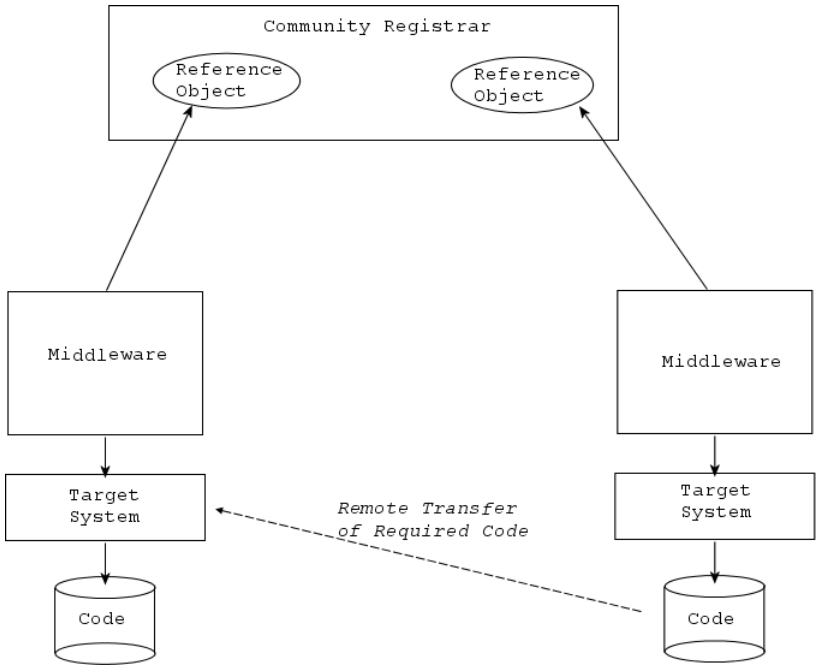


Figure 4.3: Transfer of Code Resources Using a Middleware

In the figure, each member (consisting of the middleware and the target system) can transfer code (dotted line) while maintaining their reference object. Remembering that each member is able to search through reference objects without affecting any other members, the diagram shows the interconnection and cooperation of the members within the community. Moreover, as the communication protocol is simple, being a single question, all reference objects easily supply all necessary information to questioning members.

4.5.1 Discovery and Member Management

We have now described how a member of a community can be a system executing within a middleware that is capable of interacting with other members of the community, where the middleware includes: the ability to advertise a remote representation of the system details; a proxy to middleware resources and code gathering algorithms; and, the discovery of other members. Figure 4.4 shows the link between the middleware, the target system, the remote representation object, the proxy to the middleware and the code source files of the target application.

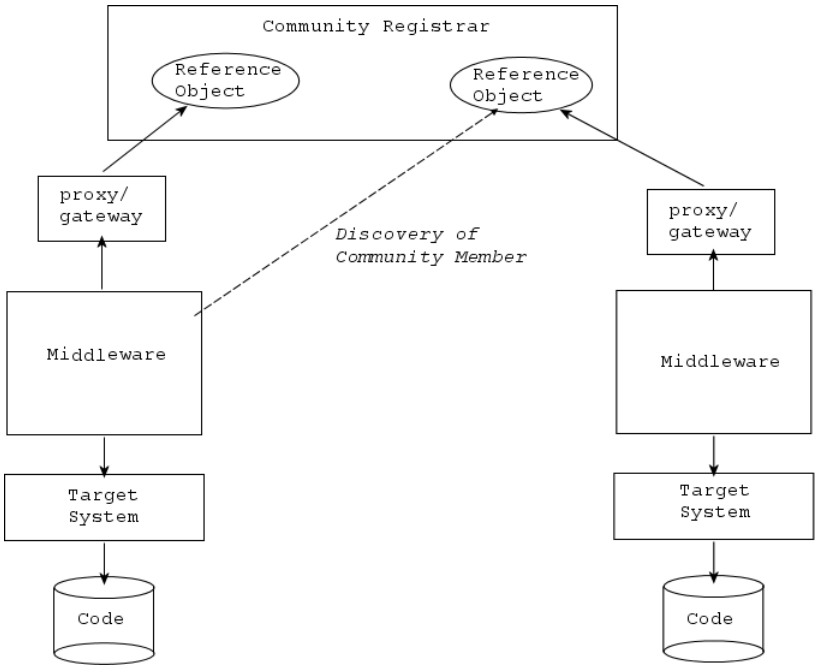


Figure 4.4: Discovering other Community Members

The figure depicts the representation object and the proxy link as two separate entities. However, they should be considered closely related and essential to each other. There are two main reasons for the link between the remote object and the proxy:

1. to update the information within the representation object in order to efficiently reflect the status of the target systems
2. to upload and download code details during interaction within the community.

As previously noted, when the connection between the middleware and the representation object is used to update current status details, a community event is generated. Such events are created and handled by the discovery technique and thus have no, or very little effect, on the inducing member.

4.5.2 Generating and Stabilising Communities

In an effort to minimise the added weight to the bandwidth of the communities discovery mechanisms, all members make use of in-built events mechanisms and

advertising proxies. Only when a member discovers a wanted code component is the message size and frequency raised. During this process several main transfer messages are sent, a corresponding discovery notice to the searching member, a forwarding request for the code details to the source member, and finally, the transfer of the code details from the source member to the searching member. During this process, the searching member's system may incur slight delays, particularly whilst redefining its structure. Furthermore, the source member's system might have a very slight delay during the code uploading process. However, this delay time period is minimal and should only be noticeable on systems with limited computation abilities.

As a community is based on discovery techniques, it does not rely on physical address knowledge. Therefore, a community can be established by any system within any network. The flexibility of this aids the establishment of the main idea behind the dynamic community: the adaptation of remote systems to their surrounding environments, communities and other systems.

To show how the techniques we discussed can be use create dynamic communities we will now analyse other similar or relevant areas of research.

4.6 Possible Methods for Developing Communities

Thus far we have outlined much of the theory behind the main contributions of this thesis. Specifically, we have discussed:

- using dynamic updating techniques and distributed activity to establish adaptive systems;
- creating a middleware framework, using dynamic systems and distributed activity, for Java applications providing interoperable adaptive systems;
- defining the attributes of a dynamic community;
- using adaptive systems to generate dynamic communities and
- analysing the effects of mobile adaptive systems within dynamic communities, gypsy agents, and their ability to spread the state of system components.

We will now discuss the techniques that we see as applicable to creating a dynamic community. However, we state that the concept and definition of a community that we have established thus far is unique to our research; no other research has discussed the dynamic cooperation of mobile systems in the same manner, nor the specific cooperation techniques for use within a community.

4.6.1 Required Attributes of Dynamic Communities

In analysing the techniques that may be applicable to dynamic communities, firstly, we point out that system adaptation is more advantageous than a simple system updating technique. There is a greater flexibility given to a system which can react to its current community as it is allows the system to change its implementation to be more appropriate at a given time. Moreover, the alteration of a system appropriate to the community is also beneficial to the community by generating community states that are consistent throughout all members. To gain such a reaction to the community, a system must have access to variables to react to.

There are several different areas of research which could be used to establish the community attributes we have described above and at the beginning of the chapter (see, for example, McKinley, Sadjadi, Kasten and Cheng [71]). Although, each technique has more specific research targets, they all display some capability in the creation of communities. Specifically, we identify the research areas of most interest to be subsections of the wider areas of Adaptive Software [71] Autonomic Systems [66, 120], and Context Aware Systems [31].

To determine the usefulness of a concept for community generation, deriving from our description and analysis of dynamic communities, we have found several features which are advantageous:

- **Mobile and Static Interaction**: dynamic communities are sections within a network through which systems may constantly pass on a regular basis. Furthermore, within a community, the ad-hoc movements of mobile systems are unpredictable, whereas, stationary systems provide continuity to the community by ensuring there are always systems present. However, a dynamic community may not necessarily require the stability provided by stationary systems, therefore, a dynamic community may also be established completely by mobile systems. This degree of community establishment may be temporary and eventually disappear.

- **Interchangeability**: there are many different types of devices interacting within any dynamic community. Each device has a specialised operating system and, in many cases, also a specialised JVM. We established in Chapter 2, that the use of a standard language for dynamic updating is more useful than a particular means of dynamic updating. The same can be applied to community interoperability. Therefore, a community establishing technique that contains the ability to adapt to a system is preferred to a technique to which the system must adapt.
- **Interaction and Communication**: a dynamic community may contain multiple methods of communication and different levels of system interaction. By limiting these communication protocols to a single type, we effectively generate a static environment, as discussed in Chapter 3. However, we also discovered that in order to know what a community member looks like there must be a commonality among the systems. Therefore, any technique must have the flexibility to establish community communication whilst also providing a means of community member recognition.

We must also include in our analysis of the three research areas, Adaptive Software, Autonomic Systems and Context Aware Systems, the perspective of mobile systems. We do so, as at times, the change of software states is reliant on some community members acting as gypsy agents. In the beginning of this chapter, we determined the key factors of a dynamic community. We will use these to identify whether a research area shows the potential to allow the establishment of localised dynamic communities. The following is an expansion of the key factors:

- **Dynamic Community**: like any community, a dynamic community is a gathering of members that are able to cooperate within a set proximity (location). A dynamic community is a community where members are able to join and leave and are able to adapt to the current requirements or state of the community. Furthermore, the community may also respond by an adaptation of the entire community. Therefore, a dynamic community is a community where each member is able to benefit from their interaction via adaptation and change as a result to community resources, and where the community is also continually adapting.
- **Community Establishment**: the establishment of a community derives from the ability for a network to allow the discovery of systems to such a degree that all systems within a particular location are able to communicate and share resources at a high and anonymous level. This will automatically enable them to establish a community.

- **Community Evolution**: community evolution is the result of the evolution of the dynamic systems within a community. As such, a community will only evolve at the rate at which its members are evolving. For example, if there is an alteration to all community member protocol details then we can say that the community has evolved to the new protocol details.

Further to these attributes, a localised dynamic community in the simplest form must allow any system, including mobile systems, to act as a member. This requires a high degree of interaction and communication. The level of interaction allowed by a community is determined by the method and technique that has been used to enable member cooperation. However, in researching the relevant areas we have chosen not to include complex areas such as AI and agent swarming as generally they are designed specifically for particular systems, and consequently, they are limited in flexibility and compatibility.

4.6.2 Adaptive Software

Adaptive software is an extremely large area of research which has recently attracted a significant amount of interest. This is a result of the emergence of pervasive computing and advancements in distributed techniques and dynamic systems [71].

McKinley et al. [71] detail two approaches to the implementation of adaptive software: Parameter Adaptation and Compositional Adaptation. However, as they also point out, Parameter Adaptation, which is the modification of program variables that determine system behaviour, is inadequate in that it will not allow the addition of new algorithms or components. This restriction is detrimental to its use for generating dynamic communities, consequently, we will not analyse Parameter Adaptation.

Composition Adaptation [1, 9, 11, 12, 92], is the exchange of algorithms or system components between systems to allow the adaptation of a system to its environment [71]. We also include in this area the techniques of creating dynamic systems that are applicable to mobile devices, such as JDrums [2]. The reason for this is that, with the inclusion of a distributed technique such as Jini [79], more so JMatos [87] for mobile devices, we believe it is highly possible to create a community establishing technique.

McKinley, Sadjadi, Kasten and Cheng [71] presented an analysis of the different techniques in compositional adaptive software. They sought to review how, when and where adaptation should occur. What they discovered was that the answers these questions are dependent on the technique applied for system adaptation. However, they do state that there are several techniques that are generally used to achieve system adaptation:

- **Separation of Concerns**: this is the separation of the functional behaviour of a system from other concerns such as quality of service and fault tolerance. This technique has been used in aspect oriented programming [59], for example AspectJ [58]. Although, this approach provides a means of separating sections of a system for adaptation, it does not provide a true means of adaptation. That is, its dynamic capabilities are limited [71].

- **Computational Reflection**: reflection is the ability for a system to analyse and reveal the details of its current execution. Computational reflection is used for system updating. It is in fact the basis of the manipulation of a JVM, such as J2SE 5.0 to extend the capabilities of its dynamic updating. This was discussed in detail in Chapter 2.

- **Component-based design**: in this technique software components are used by a system for adaptation. This approach can be entwined with late binding machines for dynamic alteration [71]. However, components must be associated with another level of identification in order for them to be used. Moreover, a form of adaptation reasoning must exist for a system to adapt to each component. This research area is seen to be important to adaptive systems [71].

The research into adaptive software by McKinley et al. [71] provides an analysis of adaptive systems and gives an indication as to how they are achieved. Unfortunately, other than indicating that adaptive software gives systems the ability to adapt to their environment, the research does not further the analysis of what is achieved overall as a result of systems adapting to each other. Moreover, in further analysis of the area, it is noted that other than achieving adaptive systems, it does not indicate the advanced implications of a technique, nor do they take into account the heterogeneity of systems prevalent in society. Furthermore, many of the techniques currently being developed lack interoperability [71] and are therefore not currently adequate for dynamic communities. Consequently, we ask a two fold question deriving from adaptive software:

Using dynamic updating within the JVM in combination with anonymous resource sharing, is it possible to create a generic adaptation framework for heterogeneous systems which remain interoperable? And, will this level of interaction be sufficient to establish dynamic communities?

To determine a response to these questions we will now analyse how adaptive systems concepts have been used for similar reasons.

The capability for adaptive software to be used as a community establishing technique and enable the transfer of system states is best demonstrated by the Context and Location based Middleware for Binding Adaptation (Colomba) system [11]. Colomba is middleware based structure which allows system component exchange based on binding strategies. Colomba based mobile systems establish connectivity similar to communities by using their ability to dynamically alter based on software components achieved via logical mobile agents. This is achieved using XML for communication standardisation and a dynamic Java Virtual Machine. It is a separation of concerns concept which uses a middleware architecture for separation.

Colomba represents a relatively new concept in the updating of mobile devices. It is not targeted at just the updating of the system but at furthering its use and adaption to networks and environments. For example, service providers can express binding strategies at a high level of abstraction using unique binding policies. The purpose of the abstraction is to present changes to the system without any intervention into the applications logic [11].

However, Colomba is restricted by its security based properties and use of context variables. Furthermore, it requires XML knowledge which gives it a large overhead. In addition, Colomba maintains that, as binding policies are determined by the providers, there is a combination of context awareness and a middleware system [11]. Moreover, it is the use of binding policies that allows a system to describe to other systems what it is capable of. This description, in the form of metadata (XML), then allows each system to bind to an environment and thus alter execution. However, as a consequence of this, Colomba is reliant on a new set of program information that is not common to many systems. Therefore, it use is limited to applications designed specifically for Colomba.

Apart from this, Colomba does demonstrate that it may be possible to establish dynamic communities using an adaptive middleware.

To support this statement we note that code discovery and consequent system alterations is a low level approach to system adaptation that may be used to manipulate systems semantics [92]. It has been shown that such a procedure is extremely flexible in that it allows systems to alter their behaviour according to the other systems it has encountered by gathering code, remotely. Advanced manipulation techniques, for example, error removal and system upgrade, are achievable using code updates via system alteration. Moreover, when this is combined with community cooperation it may allow system to continually redetermine their execution based on their current situation.

4.6.3 Context Awareness

Context awareness is the concept of a system being in tune with its surroundings by making use of the environmental context that has been made available [31]. Its significance to the field was established with the emergence of the pervasive mobile device, where the context awareness capabilities allowed mobile systems to determine how to react to the variables from different environments[1] [44].

Today, there are many different areas within the Context Awareness sector. The most relevant to community and system state transfer is in the area of middleware based developments [7, 31, 30, 44, 76, 89, 94] and self administering systems based on context [68, 76]. These areas give a commonality to systems by allowing them to gather context via a middleware. By using a middleware to locate context from within the environment, a system is able to react to it according to its own predetermined logic.

Most context aware applications are developed to aid, complement or enhance the usability of a system when it is operated within a particular situation. For example, Dey, Mankoff, Abowd and Carter [32] use a context aware toolkit to provide information to three different applications: a word predictor system, an

[1]The definition and related arguments as to the use and meaning of context are outside our scope of research. We establish that context is a 'something' derived from an environment that gives a system a degree of knowledge into the environment's use. This in turn enables it to be used as a means of system adaptation.

In/Out Board and a reminder tool. These are all applications that were developed to alter as a result of the current environmental context. In this case, the reactions are based on the requirements of the system; for example, the In/Out board can only change a person's status from *in* to *out* and visa versa. It is not possible for the context to alter the system to identify a person as *leaving*.

The applications used by Dey et al. are an example of how a reaction to environmental context can only occur if the system is aware that the context may, at some stage, be encountered. Consequently, context information that is useless for many systems may exist in a particular area. This remains even if the context is gathered by a standardised middleware. This is a result of a context aware system not being able to make use of the unknown context available; if it is not known, the context is ignored [10, 93], This indicates that a context aware system is not adapting to an environment, it is altering the execution of its already known semantics depending on context discovered from within an environment.

Context aware systems are a technique which is able to gain the most information from its environment. This includes the use of sensor equipment and small embedded applications. However, even though context aware systems seem to alter their behaviour based on current situations, they are much like Parameter Adaptation in that they are unable to alter their algorithms. This further indicates that context aware systems only change their execution according to an encountered context. This restriction is an inhibition Schilit, Adams and Want [93] describe as the IF-THEN relationship between the system and the environment.

The IF-THEN limitation of context aware systems does not completely rule out its use as a dynamic cooperation and adaptation technique. There are instances in which a generic context detail may be established [10]. For example, the use of XML type descriptors may allow systems to present contextual to the environment such as that used by Colomba [11]. Although this is not specifically adapting to the context, it is generalising it into non-pure context which may enable adaptation to occur. However, an argument against context aware systems remains as they require preprogrammed changes. We see a need for the influence of context that presents the potential for systems to grow in ways not seen by the programmer, and thus change dynamically according to an environment.

In order for context aware systems to be used as a community establishing technique, we therefore note the need for two concept extensions. Firstly, to create community type cooperation, context aware systems must be able to view and use the context of other systems; currently they, generally, only allow for one-way evolution. Although, this capability exists to an extent in some context aware implementations, the enhancement of this ability would allow a better knowledge of the movement of neighbouring mobile systems. Secondly, for the transfer of system components the ability for systems to present their system code as context, or the response to a context, is required. This attribute could allow the mobile context aware systems to locate new system components based on context awareness, and become dynamic community members. The combination of both these extensions within context aware systems could readily enable them to establish *dynamic communities* and allow for *community establishment* and *community evolution* through systems state transfer. This is an attribute we have not found in any context aware research.

4.6.4 Autonomic Systems

The concept of Autonomic Systems relates to creating high level operations where user intervention is never required; that is, the creation of self-managing system components [120]. The goal of research in the field is to allow autonomic elements to achieve self-healing, self-configuration, self-protection and self-optimisation [120]. Furthermore, that other design techniques will give planning mechanisms to autonomic systems allowing them to decide on how a task is best achieved within an environment [66]. Both these elements of autonomic systems are essential for its use as a technique for establishing dynamic communities.

As autonomic systems have the ability to perform high level tasks without the intervention of a user, they would essentially be able to cooperate easily with other systems. Moreover, autonomic systems that react to the environment are already displaying some attributes of community cooperation. And, essentially the ability for an autonomic component to self-manage is, to a degree, adaptation. Of particular interest is the capabilities of self-protection and self-optimisation within autonomic systems. If alterations to the component are based on the environment, including other systems within the environment, then many autonomic systems may already (or could be simply altered to) enable *community establishment*, *community evolution* and system state transfer and thus allow for *dynamic communities*.

However, autonomic systems are high level applications. Their implementation as a middleware has not been achieved and the research in the area focuses on embedded devices, for example Flecks devices [28].

4.7 Summary

Dynamic communities are simply a location specific network which allows systems to interact on a more intricate level than is generally achievable. However, during this chapter we determined that the creation of a dynamic community is not a simple concept. And, although, we have discovered that some of the main aspects in generating a dynamic community are found in different research areas, not one of these areas contains *all* the required aspects of a community.

We have established that in order to create a means of dynamic community cooperation, any technique must overcome the problems associated with system interoperability and allow for system adaptation and cooperation. Moreover, if a technique is truly designed for full community interaction then only minor limitations should be placed on the community members. Consequently, we suggested that a middleware design is the most appropriate technique for creating dynamic community cooperation between heterogeneous systems. We further suggest that, for interoperability reasons, the middleware make use of Java distributed communication and dynamic updating capabilities. We have discussed throughout Chapters 2, 3 and 4 that this would allow systems to adapt to code resources made available within a proximity based community. This is supported by McKinley et al. [71] as part of their suggestion for future works in adaptive software.

The remainder of this thesis details our solution.

Chapter 5

Architecture and Design

5.1 Introduction

The previous chapters provided an analysis of dynamic systems and distributed cooperation methodologies, and introduced the concept of dynamic communities. During the discussion that followed the analysis, it was determined that a technique does not yet exist which provides mobile and stationary systems with the ability to share system components in the form of discoverable distributed resources, nor adapt to the current resources available within a localised area. We also established that, system cooperation in the form of community resource adaptation could be beneficial within several different operating scenarios: *location specific application states*, *mobile system adaptation*, and *system state transfer via mobile adaptable systems (gypsy agents)*. The remaining chapters of this thesis present a solution to the use of adaptable systems in creating dynamic communities. This is provided in the form of a framework, designed for heterogeneous systems, named the Dynamic Update Projecture Environment (DUPE) framework.

In this chapter we detail all the specifics of the DUPE framework's internal design and architecture. The level of specification provided makes it is possible to create an implementation. The chapter begins by giving an overview of the DUPE framework and its use for community cooperation. Following this, we describe the main sections of the framework:

1. Distributed System Communication and Code Sharing
2. Dynamic Systems Alteration and

3. Community Interaction and Cooperation.

These sections of the framework incorporate the key areas that we established in Section 4.6.1. This has been achieved by controlling class loading within a target JVM, discovering and advertising in a dynamic community, initialising a target application, and managing mobile movements. The three main sections of the framework each comprise several different sections. The relationship among the different sections of the specific areas will be outlined in Section 5.4. Beginning with this section we provide an overview of all the DUPE sections, however, first we introduce some new terminology and provide a general view of the framework. Most of the sections of DUPE are addressed in this chapter, however, several are further discussed in the next chapter, and the complete details of the security and adaptation control aspects of the framework are presented in Chapter 7. The description of the framework will be provided beginning with a top-down approach.

5.2 Terminology

Before we begin our discussion we will describe the new key terms that are used throughout the remainder of the thesis.

DUPE: A framework that gives heterogeneous systems adaptive capabilities.

DUPE Middleware: Any middleware implementation that holds to the standards provided by the DUPE framework.

DUPE Compatible: An application becomes DUPE compatible when it is implementing within any DUPE middleware.

DUPE System: Once an application is DUPE compatible it is referred to as a DUPE System.

DUPE Community: A localised group of DUPE systems that are interacting based on the DUPE resource cooperation specifications. The state of a community is the state of all resources within the community (this is detailed in Chapter 6).

DUPE Member: Any DUPE system participating in a DUPE community.

DUPE Service Object (DSO): The remote service as advertised by a DUPE Member within a DUPE community. The DSO represents the resources that the DUPE member is offering to the DUPE community.

5.3 The DUPE Framework

The first section of our DUPE framework discussion provides a basic structure overview. This overview is designed to give the simplest view of the framework.

5.3.1 Discovering and using Remote Classes

Essentially, the DUPE framework allows an application to redefine its runtime execution using classes obtained from other applications within the same location. The downloading of remote classes[1] is achieved through a remote class loader structure using a distribution technique designed according to our recommendation in Section 3.6. For DUPE we have applied Jini as the distribution technique for reasons discussed in Chapter 3 and 4. Figure 5.1 provides an illustration of how the remote class loading works. The figure shows how the proxy access setup

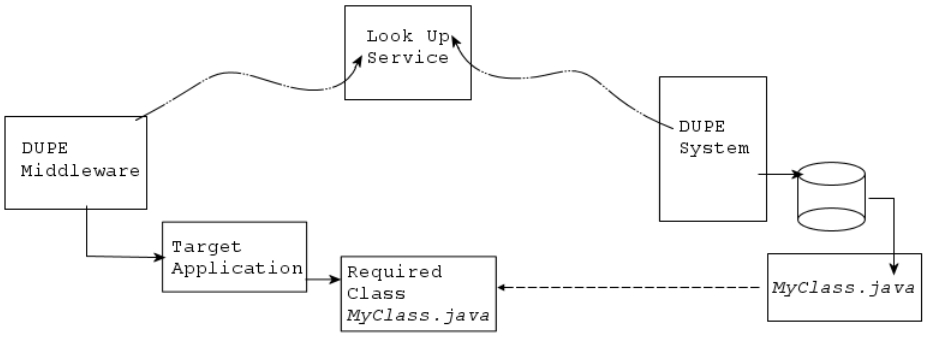


Figure 5.1: Remotely Gathering Class Details

is used for a middleware and target application. The middleware is able to look up other systems which contain requested class files. Then, via their proxies and remote services, the two systems communicate and transfer required class details. Figure 5.2 details specifically how the conclusions from Section 3.6 have been applied to the DUPE framework. This figure depicts the proxy setup clearly and depicts the direct path that remote class transfer takes (shown as a red path). If it is the first time the class has been used by the DUPE system, no redefinition is required and it is instantiated as normal. However, if during execution a new class version becomes available, it can then be used for creating objects. Exactly

[1]It is not appropriate for some Java files to be loaded remotely using the DUPE structure. For obvious security reasons, the following should be excluded: standard J2SE packages, and the group of `sun.*` packages.

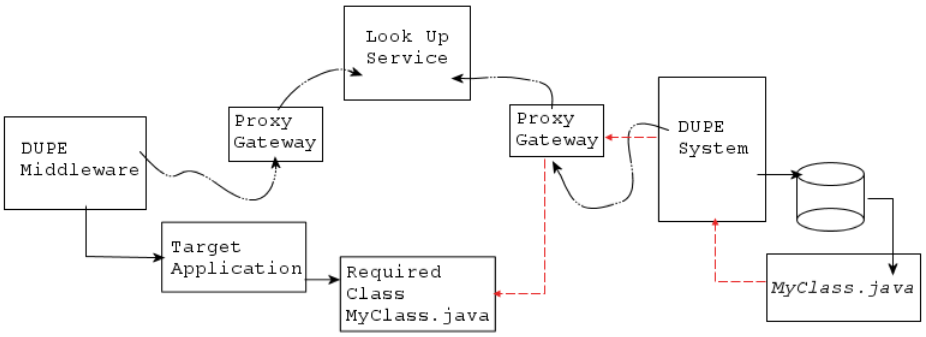


Figure 5.2: Remotely Gathering Class Details using Proxy

how this redefinition is achieved is specific to the dynamic updating technique used for a framework implementation. However, what must be achieved is an alteration in the system based on Java byte code. Furthermore, all byte code is obtained in the form of a complete Java class.

5.3.2 Establishing DUPE Communities

Although the technique we have described for discovering remote classes and distributed resources is simple, it is also very useful for system cooperation. For example, we established in Chapter 4 that the communication in a dynamic community consists of a single query with unknown repercussions. That is, a community maintains a non-task oriented protocol. DUPE's class discovery technique incorporates a communication protocol that, in its simplest form, asks for a class and receives in return Java byte code. The content and exact implications of the byte code are unknown. Furthermore, all remote resources advertised by a DUPE system can be discovered by any other DUPE system. This is achieved using Jini multicast which also restricts it to localised cooperation. According to our definition, this level of cooperation can form a local dynamic community.

We will discuss DUPE communities in detail in Chapter 6. The focus of the remainder of this chapter is to provide a comprehensive understanding of the DUPE framework. This is necessary as a complete understanding of DUPE communities is essential.

5.4 Framework's Internal Structure

DUPE is a middleware framework. It has been designed so that it can be implemented as different compatible middleware applications to allow dynamic communal cooperation between heterogeneous systems. All systems executed within any DUPE middleware are DUPE compatible. This provides the application with the ability to participate in DUPE communities irrespective of its intended use.

The following is a description of the core components for the framework. It is suggested that the reader view this list in conjunction with Table 5.1 and Figure 5.3.

Distributed Class Sharing: the DUPE framework uses the discovery of remote resources that are advertised by members within the DUPE community. Links to these resources are contained within the DSOs that represent each of the community members. During Chapter 3, we established that Jini provided the necessary anonymous discovery attributes, and, that when implemented as a proxy, Jini service can act as a gateway to the resources located on a member's system. Moreover, Jini services provide a means of maintaining DUPE communities via its event mechanisms for entering, exiting and adapting members.

Dynamic Systems: all members of a DUPE community should be dynamically adaptable. That is, all DUPE compatible systems must have the ability to *continually* adapt to the state of their current community. In Chapter 2 we established that there are many different possible methods for creating dynamic updating within a JVM; for example, J2SE 5.0 `java.lang.instrument` package [108]. The technique to be applied needs to be chosen according to each separate DUPE implementation. Moreover, the DUPE specification states that a DUPE implementation that can communicate within a DUPE community, yet can not achieve runtime dynamic updates may be known as a limited DUPE middleware. This flexibility is significant to DUPE in that it demonstrates that the framework can be heterogeneous.

Adaptation and Security Controls: as a result of the dynamic interaction of systems within DUPE communities, including the sharing of code, it is possible that any discovered code may be illegitimate, and therefore, all system adaptation must be tightly controlled. In order to maintain the

DUPE Sector	**Description**	**Section**
Distributed System Communication and Code Sharing		5.6
Gateway Proxy	Provides the link from a DUPE service object to the advertising system.	5.6.1
DUPE Class Loading Structure (DCLS)	The overall controller of the class loading for the target system. It includes DUPE's unique class loader: Remote Upload Class Loader (RUCL), and is generally coupled with functions of several other components.	5.6.2
DUPE Service Object (DSO)	The remote representation of the DUPE compatable system.	5.6.3 & 6.3
DynamicClassEntry	An external representation of a class resource available within the DUPE system.	5.6.3
Dynamic Systems Alteration		5.7
Target Application	The system which has been elected to become DUPE compatable. This can be any Java program.	5.5 & 5.6.2
Dynamic Instrumentation Layer	This area of the framework is dependent on the chosen JVM and specified dynamic updating technique. It is discussed in during discussion on the JVM and implementation	Ch. 2 & 5.7
Java Virtual Machine	The specific JVM used for each DUPE framework implimentation and is not detailed within the framework.	Ch. 2 & 5.7
Dynamic Class Manager (DCM)	Manages all dynamic manipulation of the target system and its JVM during adaptation. This section is specific for each DUPE framework implimentation as it is linked to each specific JVM.	5.7.1
DynamicClass	An internal representation of a handled class.	5.7.3
Class List Manager	The controller of a dynamic list of all class details instantiated by the target application. Class details are stored as a DynamicClass format.	5.7.2
Community Interaction and Cooperation		5.8
Discovery Manager	Handles all communication within the community. Contains the key sub-sectors: Community Observer and Event Generator.	5.8.1
Community Observer	Listens for alteration within the community. Main duty is listening for community events.	5.8.2
Event Generator	Is able to trigger events within the community based on changes within the target system. This is achieved via the DUPE service object	5.8.3
Security Control	Ensures all adaptation and resource gathering is acheived safely. A combination of security and trust measures.	7.2
Adaptation Control	Manages the reasoning for adaptation of the target system.	7.3

Table 5.1: Section Reference for DUPE Framework Discussion

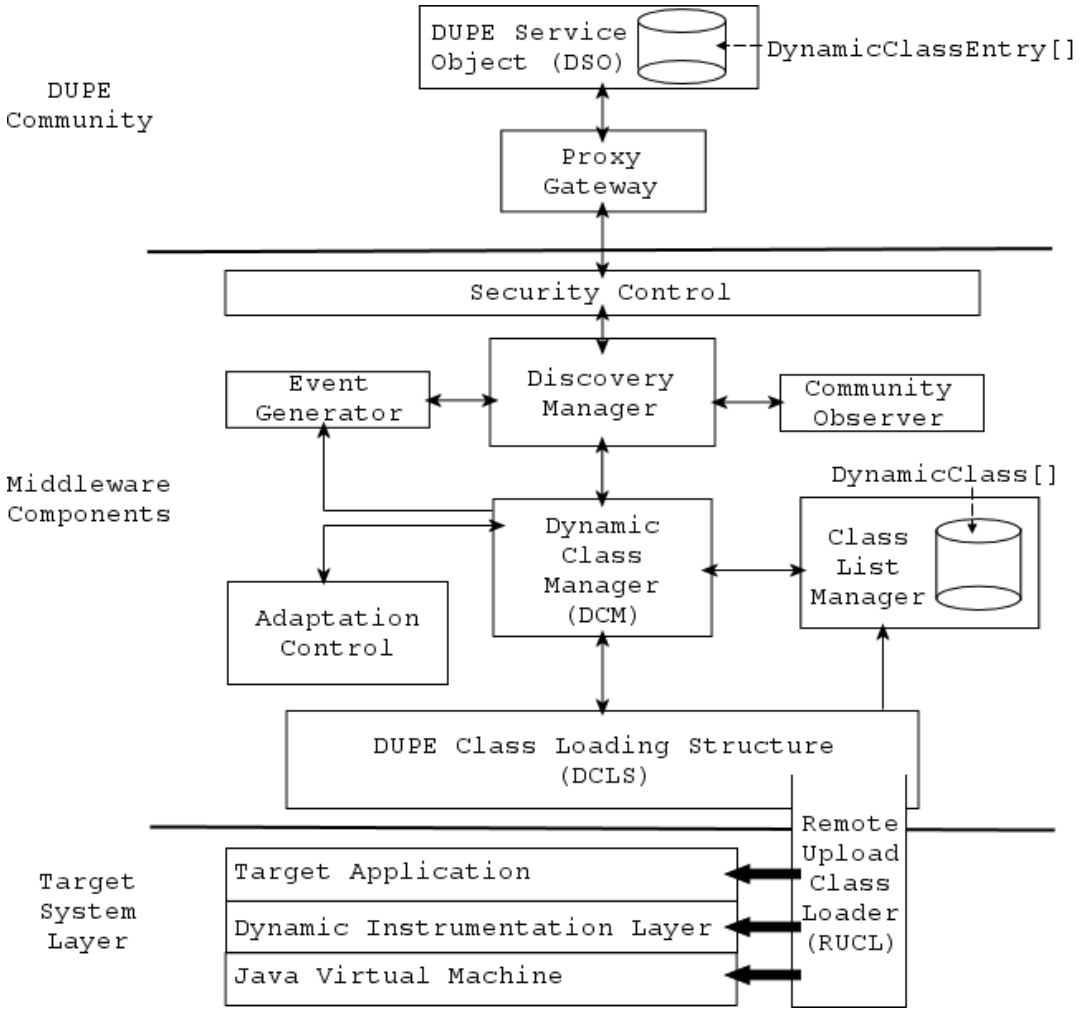


Figure 5.3: DUPE Framework Architecture

safety and reliability of all interacting systems, the framework stipulates several security and adaptation controlling mechanisms.

Framework Consistency: all the aforementioned aspects of the framework must be integrated in the correct manner. A generalization of the formulation and the interconnection of each sector is shown in Figure 5.3. Together with the main components, those stipulated in the previous points, several other key linking sections are required within the framework. However, as long as a DUPE middleware implementation can cooperate with other

members of a DUPE community, via DSO discovery and resource gathering, and achieve dynamic adaptability, the middleware can be termed a full DUPE compatible implementation.

Table 5.1 describes the sections of the framework and provides a reference to their discussions throughout this work. The internal associations and processing links of these sections is shown in Figure 5.3. The pathways that represent internal communication flow are indicated in the figure by directional arrows. These internal links allow the framework to control class initiation and redefinition.

Many areas of the DUPE framework are designed to ensure that an implementation can share files, data, and design structures and maintain the correct execution of the target application. Most important of these is the DUPE Class Loading Structure (DCLS). The DCLS provides the framework with a class loading mechanism to control the initialisation of all target system components. Working together with the DCLS is the Dynamic Class Manager (DCM). The DCM controls the dynamic adaptation aspects of the target application by maintaining a list of current resources (class files). As previously mentioned, the extra flexibility, in terms of dynamic class adaptation, available to DUPE compatible systems, makes them vulnerable to greater risk of unwanted intrusion. The framework counters such threats by using security measures. The framework is structured so that all external communication, such as resource gathering, is filtered through security and adaptation controls. This is discussed in detail in Chapter 7.

DUPE maintains a link between itself and the bootstrap class loader of a targeted system using a combination of the DCM and DCLS. Here, the DCLS overrides all the core class loading methods of the JVM controlling all the class loading of the target application. The use of the DCM and the DCLS for this task is indicative to each implementation and dependent on the dynamic updating technique applied[2]. There are no constraints or obligations that inhibit this association between the DUPE middleware and a target application. Moreover, we do not specify any particular policy for updates. Some examples will be given in Chapter 7.

[2]Depending on the framework implementation, multiple JVMs may be required to separate the DUPE middleware from the target application. We provide an implementation using both a single JVM (Chapters 8 and 9) multiple JVMs (Chapter 10).

5.5 Target Application Handling

We discussed in Chapter 2 that the class loader mechanism of the JVM controls the loading and initialisation of all classes associated with a Java program. We also detailed how a program's class loading procedure can be manipulated by defining a subclass of the `java.lang.ClassLoader` class. The DUPE framework makes use of this technique to control the target application's execution. The DCLS is a personalised class loading structure that includes a redefined class loader: `dupe.discovery.RemoteUploadClassLoader` (RUCL). The RUCL is used by the DCLS to control the underlying loading mechanism for any Java based program. Where this relationship sits within the DUPE framework is shown in Figure 5.4.

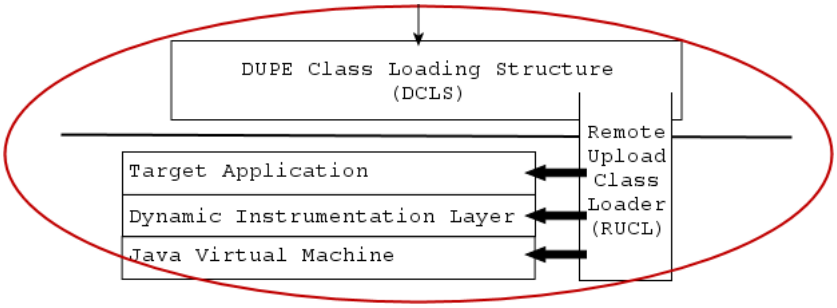


Figure 5.4: Target Application Section of DUPE Architecture

The RUCL must be implemented in such a way that it is able to execute and control, correctly, any Java application. It is the job of the DCLS to make sure that all attributes, such as environment variables and command line arguments, are accessible to the target application in the same manner in which they would normally acquire them. As a result of this, any DUPE implementation is able to load the starting class of a program in the exact same manner as a JVM would normally achieve. Moreover, it should also be designed to search the targeted start class for the `main()` method, and if found invoke it through the use, generally, of reflection techniques[3]. Once the initial class has been loaded by DUPE, all the subsequent classes, as initialised by the target system, will be loaded via the DCLS's class loading mechanism according to the JVM's class loader hierarchy [65]. This means that all classes instantiated by any DUPE system will also be loaded using the DCLS.

[3]For further information on reflection see Lindholm et al. [65].

However, this does not prevent a developer from producing their unique class loader for use within an application that may be used with a DUPE middleware. If an application using its own class loader is executed within a DUPE middleware, Java's hierarchical class loading architecture determines that, even though a system is loaded via DUPE, it will still be able to use its own class loader for specific class loading requirements. However, the unique class loader used by the application will always revert to the DCLS if, or when, it fails [65]. Therefore, the RUCL will still be used as a class loading backbone for the application.

The DCLS is essential to the DUPE framework. For example, within the framework there is a maintained reference within the Dynamic Class Manager (Section 5.7.1) to all classes loaded by the target application. The details provided by this dynamic information are used to determine adaptation requirements and to inform community members of resources that the system has to offer. The relationship between the DCLS and other sectors of the framework will be addressed in the relevant sections of Chapters 5, 6 and 7.

As we indicated earlier, the target application of the DUPE system must still execute in the same way as it would outside a DUPE middleware. It is therefore important that all system commands, JVM commands and special, or unique settings, for example, codebase indications, be handed to the JVM as normal. And, that all arguments for the target application are also handed on. This should all be achieved without interfering with DUPE's communication mechanisms.

An application that would run normally (within Windows XP command prompt), as shown in Figure 5.5, could be a DUPE system using the DUPE start up shown in Figure 5.6[4]. It can be seen from the command arguments of both figures that an IP address (`192.168.0.100`) is handed through to the target application `coffeeMachine.CoffeeMachine`[5]. It should be noted that this argument is for the use of the target application and not the DUPE structure, and that all arguments will be used by the target application in the exact manner that it would normally use them.

[4]This example is a specific implementation of DUPE (DUPE 5.0, discussed in Section 9) and, therefore, the information labeled as DUPE start up commands may be ignored at this stage.

[5]The full details of the argument use are also not necessary at this point in the discussion.

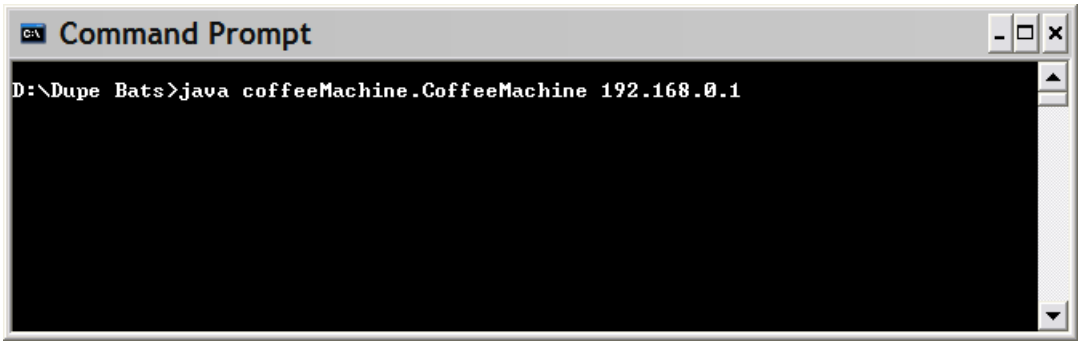


Figure 5.5: Standard Execution of Target Application

Figure 5.6: DUPE Execution of Target Application

5.5.1 Standard Class Access

DUPE aids systems by providing them with the ability to discover and use classes from other systems. However, it must sustain the ability of systems to discover local classes. And, although it is unlikely that any system adaptation would be based on local classes, the framework provides the flexibility to allow internal discovery to occur. Therefore, there are two aspects to the classes of a DUPE system: advertising and discovery.

All DUPE systems advertise their local classes within the community. The remote classes are known as resources. The advertised resource represents a class that the DUPE system has loaded during standard execution. These resources only include those accessed via the JVM class structure, classpath components[6],

[6]Class components should be held in a 'trusted' directory for security purposes. This will be further discussed in Chapter 7.

and other areas that are specifically assigned with the system, for example, codebase servers. It also includes classes loaded via remote discovery.

The discovery of resources may allow an application to use remote classes as well as local classes. This provides the DUPE systems with access to classes beyond their own local device. In fact, it may be possible to execute an application using only remote class details.

5.6 Distributed Systems and Code Sharing

DUPE uses concepts derived from both remote objects and web based class loading to load classes via a remotely accessible class loader proxy DSO.This enables systems to share resources, and allows them to find and use all shared resources from within the DUPE community without prior knowledge of their whereabouts.

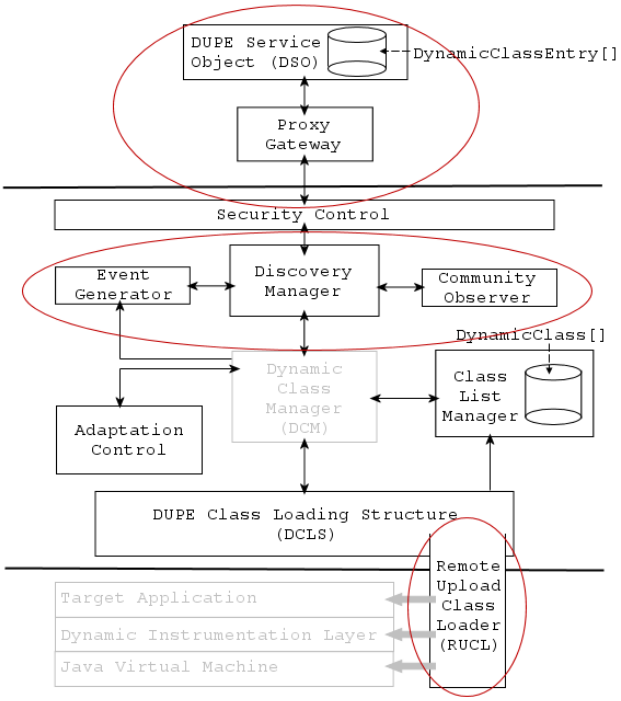


Figure 5.7: DUPE Community Interaction Design

The remote components also provide DUPE systems with the required information to adapt to the community according to their needs and the needs of the

entire community. The distributed aspects of the framework incorporate several sections of the architecture. This is illustrated in Figure 5.7.

In the figure, the insignificant sections are greyed out and the most significant sections circled. Within the framework, there are several interface definitions that determine how the core aspects of class and community control are developed. During the analysis of each section of the framework the relevant interfaces will be defined and their place within the structure of the framework will be identified.

5.6.1 Accessing System Resources

As explained at the beginning of this chapter, a proxy and gateway set up is used by the DUPE framework for discovery and communication purposes. This set up facilitates the connection among DUPE systems, via a DSO, for community searching. The searching mechanisms are controlled using the accessible methods of the DSO linking it to the host system.

DUPE systems are able to discover others within the community and advertise their own DSO representation. Each community member will advertise itself as an available service and continually discover other services (community members). This scenario is shown in Figure 5.8.

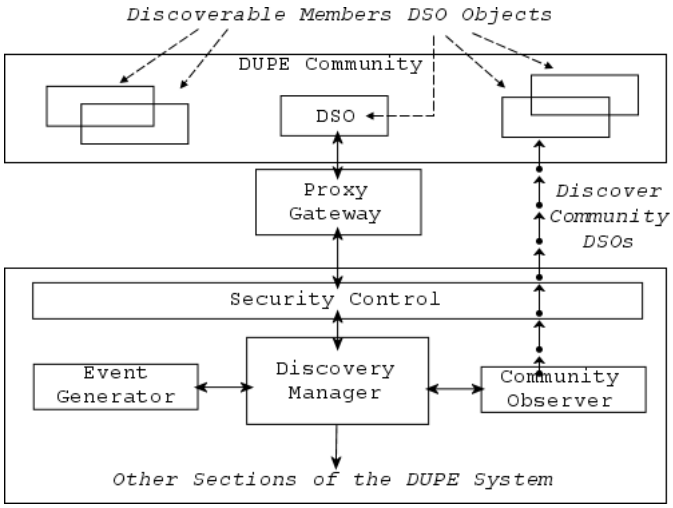


Figure 5.8: Discovery of DUPE Community Member DSOs

The two-way sharing of all members of a community provides, within the community, a web of accessible DSOs that are available to all members.

However, the framework discovery and searching specifications remain mildly flexible (this is discussed in detail in the next chapter). They do not disallow the use of a *discover only* or *advertise only* implementation. The specification remains this way to allow an implementation designed for extremely resource constrained devices, such as embedded systems, to cooperate within the DUPE community. However, as systems of this nature may be noticeably interrupted by uploading resource details, they may wish to restrict their community interaction.

Within a DUPE community the advertisement of a DSO is achieved using Jini proxy services. Any Jini registrar can act as the community resource centre where DUPE members share their DSOs. The community resource centre may be located anywhere within the community, however, if available, it is likely that the it will be located within a stationary device of significant capabilities; for example, a web server. Moreover, for the community to remain active the registrar host system's power should remain on, consistently. A DUPE system advertises its remote service representation within all located registries. Furthermore, within a single location, should several DUPE communities exist, a DUPE system will register itself using a single identifier, ServiceID [79]. The implications of this concepts are beyond this thesis and are part of our future work suggestions in Section 13.1. However, we have noted that internal searching within a system has the potential to create a continuous searching loop. The framework accounts for this using the proxy and the individual ServiceID. Figure 5.8 depicts the advertising of services and illustrates how they use their proxy gateways to gain access to the internal classes of the host system.

5.6.2 DUPE Class Loading Structure

To understand the fundamentals of DUPE based system cooperation a clear definition of the DCLS will now be provided. As already mentioned, the DUPE Class Loading Structure (DCLS) is the personalised class loader structure used within the DUPE framework. It achieves several of the key class and resource manipulation aspects of the structure, many of which are achieved using its Remote Upload Class Loader. (RUCL). Furthermore, there is a close relationship among discovered resource use, dynamic instrumentation of the target system and target JVM, and the attributes of DCLS as the key class loading structure for the

framework. Consequently, the DCLS must supply other framework sections with detailed information concerning class manipulation. The full list of the DCLS tasks are as follows:

- downloading resources from within the community. This does not include determining is they should be used
- controlling the loading of the target application classes
- providing the information for the storage of all class details to the Dynamic Class Manager (DCM)
- providing the initial class loading link into the JVM for dynamic manipulation and
- providing the new resource and class information for the final stage of dynamic instrumentation[7].

The DCLS contains a subclass of `java.lang.ClassLoader` that overrides the core class loading methods.

Within the DCLS the interface `dupe.discovery.RemoteClassLoader` (shown in Figure 5.9) determines the procedures that are used to link the DCLS with other sections of the framework, and, within its own class structure.

The main class loader within the DCLS should implement the methods from this interface. This class loader, the `dupe.discovery.RemoteUploadClassLoader`, is the RUCL we have previously discussed. The specific methods of the interface are used to maintain the separation between the normal operation of a class loader and that which can be invoked remotely. However, the entire structure of this section is not imperative to the DUPE framework and parts may be altered according to the requirements of an individual implementation. The flexible sections of the interface will become obvious as other sections of the framework are discussed.

[7]Although the dynamic instrumentation is specific for each implementation, generally the final state(s) of class instrumentation are, or can be achieved via the DCLS.

```
public interface RemoteClassLoader extends DynamicRemoteAccess {
   public Class loadClassBytes(byte[] classData, String className) throws RemoteException;
    //override core method
   public Class findClass(String className) throws ClassNotFoundException;
    //override core method
   public ServiceID getServiceID(); //access current ServiceID
   public void setServiceID(ServiceID sid); //setting the ServiceID
   public void startClassSearch(int delay); //optional timer for community resource search
   public void addRemoteLoader(ServiceID serviceID, RemoteClassLocator service);
    //store reference to a service
   public void removeRemoteLoader(ServiceID serviceID); //remove a reference to a service
   public LoaderType getLoaderType() throws UnknownHostException;
    //access service information
   public DynamicClass getLoadedClass(String name); //get a loaded class from within system
   public void setDiscoveryManager(DiscoveryManager discoveryManager);
    //set the discovery manager of the system
}
```

Figure 5.9: `RemoteClassLoader.java`

5.6.3 DUPE Service Object

Earlier, we introduced the DUPE Service Object (DSO) as the remote representation of a DUPE system. The DSO provides sufficient information to discovering systems to enable them to decide if there are resources on the advertising system that are useful.

The accessible specifications of the DSO are contained within two interfaces and two class files. The DSO is the most specific element of the framework. All DSO implementations must implement the interface `dupe.discovery.RemoteClassLocator` (Figure 5.10), which is itself a sub-interface of the interface `dupe.discovery.DynamicRemoteAccess` (Figure 5.11).

```
public interface RemoteClassLocator extends DynamicRemoteAccess {
  public Class loadClassBytes(byte[] classData, String name) throws java.rmi.RemoteException;
}
```

Figure 5.10: `RemoteClassLocator.java`

The DSO, as a Jini service, can also contain a collection of `net.jini.core.entry.Entry` entries [79, 36]. The framework uses this to maintain the current

```
public interface DynamicRemoteAccess extends java.io.Serializable, RemoteEventListener{
  public DynamicClass generateDynamicClass(String className)
        throws ClassNotFoundException, RemoteException;
  public void notify(RemoteEvent evt) throws RemoteExceotion, UnknownException;
}
```

Figure 5.11: `DynamicRemoteAccess.java`

details of all available classes on a system. The DSO has an associated instance of `dupe.discovery.LoaderType.java` (Figure 5.12) which is an implementation of the Jini service interface.

```
public class LoaderType implements net.jini.core.entry.Entry{
  public String hostName; //host name (optional)
  public Boolean allowInternalSearch; //internal search switch
  public DynamicClassEntry[] classes; //array or DynamicClassEntrys


  /*Constructors */
  public String toString(){ ... }
}
```

Figure 5.12: `LoaderType.java`

Each LoaderType entry contains an array of `dupe.discovery.DynamicClassEntry` objects. A DynamicClassEntry is a remote representation of a class resource that is available on the system associated with the DSO. The information of the DynamicClassEntry objects is the core information provided by the DSO. Together, they provide enough details for a discovering DUPE member to determine the systems usefulness. Each DynamicClassEntry within the DSO is analysed by the discovering system to check that the service is useful for adaptation needs. If a system decides to load a complete resource that is represented by the DynamicClassEntry it is able to call it via a call to the service using the method `loadClassBytes(...)`, specified by the RemoteClassLocator interface. The complete class structure of the DSO and its related classes is provided in Figure 5.13, the structure of a DSO as a Jini service is shown in Figure 5.14 and the contents of a DynamicClassEntry are given in Table 5.2.

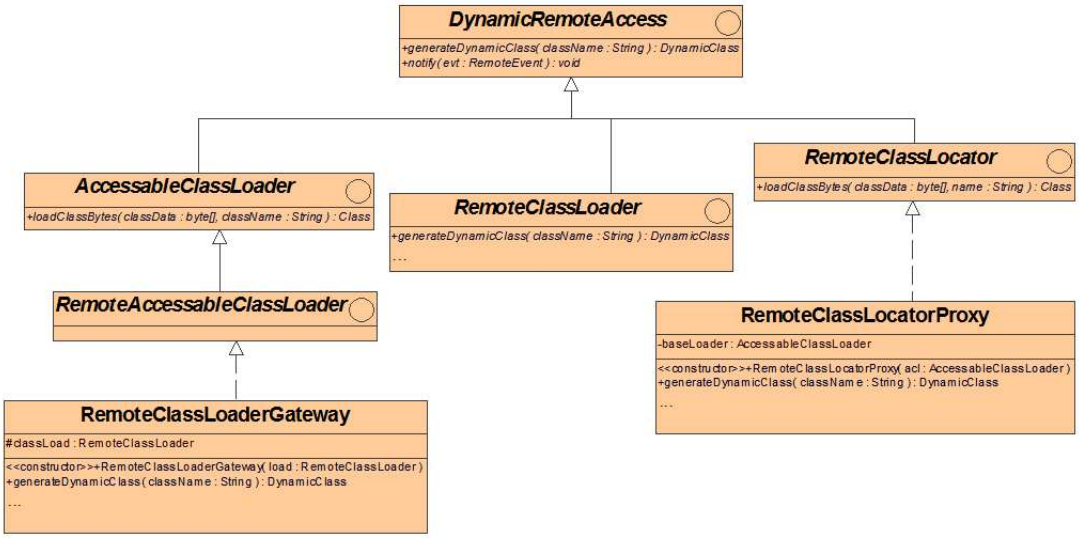


Figure 5.13: Class Diagram for DSO Structure

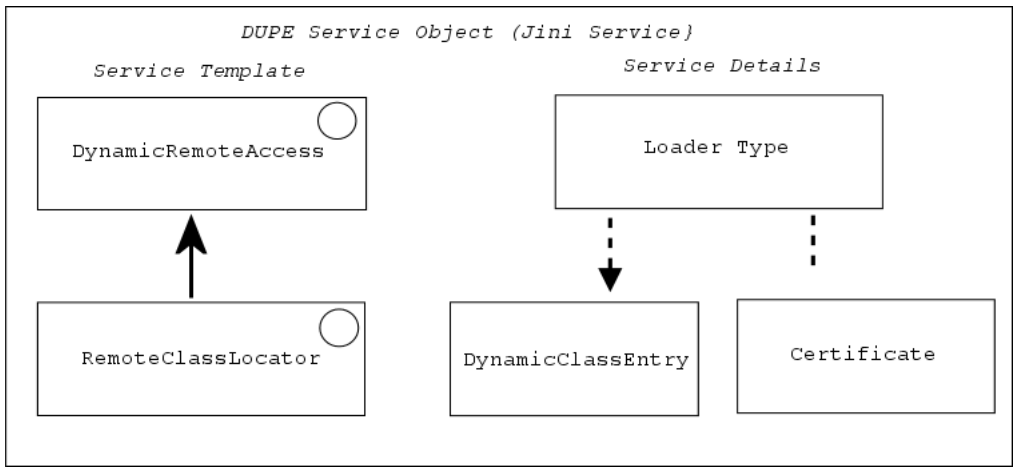


Figure 5.14: DSO Jini Structure

Name	Type	Description
`className`	`String`	The reference name for the class file resource.
`attributes`	`String[ ][ ]`	The details of all attributes that have been assigned to the class resource. (For further details see Chapter 7).
`certificates`	`Certificate[ ]`	An array of certificates that have been assigned to this class resource. (For further detail see Chapter 7).

Table 5.2: DynamicClassEntry Contents

5.6.4 Remote Manipulation

By defining several interfaces for remote access to DUPE systems, the framework provides community members *restricted* access to its class loading capabilities. These methods provide the link between the DCLS's gateway and its DSO which enables access to the DUPE systems resources. The link then allows a community member to check for classes from within the DUPE service, loaded or unloaded, through the DSO to the DCLS. The methods also allow any accessible resources (essentially anything within reach of the DCLS) to be downloaded as required. The discovered DSO will only send the resource class bytes.

The DCLS is able to look within its designated classpath for files (or further if internally specified) and within its JVM to find any Java classes that are requested via resource acquisition. The class bytes are then transferred to the system that has requested them. A DUPE system will never pass any byte code through its own JVM until required by the target applications normal operation, and this should not be achieved until the defineClass() method (a bootstrap class loader method) is instantiated within its own structure [65].

A key to the remote resource loading within the DUPE structure is the dynamic discovery of DSOs. As previously discussed, this gives applications restricted access to the actual inner sections of a remote JVM providing the structure with two of its crucial aspects:

1. the ad-hoc discovery of DUPE systems. There is no need for a system to know an actual address of the system that are going to use to gather resource details.

2. the transfer of byte code through any DUPE compatible system. Every class that is searched for by a DUPE member does not necessarily have to come from the same member.

By specifying the methods of the DSO, the restrictions placed on the processing performed by a searching member can be established. The discovery techniques that the DCLS uses, eliminate the need for classes to exist on the same machine as the host program. A discovery based class loading structure provides access to a remote DSO which can be discovered and used as proxy objects to download Java class files. This then gives access to Java class files that may be downloaded via an *unknown* system, files that have, or have not been instantiated by the discovered service object (depending on user preferences).

The manner in which the DSO and the DCLS are structured eliminates the common limiting aspects of other remote class loading techniques: such as web based class loading or distributed objects. The reason for this is that the information gained from the resource is not specifically contained within the service object; the object is simply a means for discovery. This, therefore, eliminates the problems, as discussed in Chapter 3, associated with:

- the use of a common interface, and
- the requirement of direct knowledge of the hosts URL reference.

Consequently, DUPE systems can run entire programs on their host device even though no files are stored directly within its memory. This particular attribute may prove extremely useful for limited memory devices and embedded systems.

5.7 Dynamic System Alteration

The dynamic adaptation attributes of an implementation of the framework are determined by the updating technique that has been applied. The technique applied will evidently determine the level of adaptability available within to target application, and provide its individualisation.

We discussed the reasons for using Java as the base language and detailed many of the techniques applicable to Java systems in Chapter 2. In that discussion it was noted that each technique has its own specific method for applying updates to the JVM. For example, J2SE 5.0 provides a package (along with some specific commands) [108] whilst the J2SE 1.4 HotSpot JVM requires the use of an internal JVM reference: a JVM within a JVM [103].

The following sub-sections provide the guidelines and specification that are required for the implementation of the dynamic aspects of the DUPE framework. However, due to the individual nature of each dynamic alteration technique the specification may be altered as required during the actual framework implementation. Yet, as long as the DUPE community cooperation specifications, the DSO contents, are maintained, and the information provided by the system holds true according to Java class bytes verification and DUPE security measures (Chapter 7), the individual method in which an implementation maintains its dynamics is not detrimental to the cooperation of the DUPE system within the DUPE community. In fact, the individual nature of each implementation provides a means of interoperability between a larger variety of systems, and is a key contribution of this work. This section of the framework is shown in Figure 5.15.

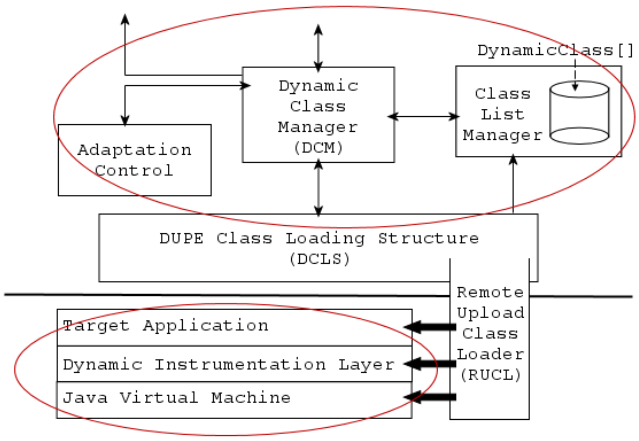


Figure 5.15: Dynamic Instrumentation Aspects of the Framework

5.7.1 Dynamic Class Manager

The Dynamic Class Manager (DCM) is the hub of the dynamic adaptation of the framework. The DCM contains a direct link to the DCLS, as can be seen in the

illustration in Figure 5.15. This ensures that the dynamic interchange of classes is synchronised with the general loading of classes. Moreover, at times, depending on the DUPE implementation, the separation of the DCM and the DCLS may be difficult to obtain. For example, if using J2SE 5.0 it is best, and easiest, to incorporate the instrumentation algorithms within a class loader implementation [6]. Therefore, any dynamic altering method, such as `reloadClass(...)`, may be included within the implementation of, or an extension to, the RemoteClassLoader interface or subsequent RCLS.

The DCM, along with the implementation of the dynamic updating, and as a result of its relationship with the DCLS, should be passed a reference of all classes as they are loaded by the JVM (this is achieved using a Class List Manager (CLM) (see Section 5.7.2)). By maintaining a reference to all the loaded classes, the DCM is able to easily check the class details for the adaptation reasoning process prior to adaptation.

The DCM is also closely linked to the community event gathering mechanism of the framework. The DCM must provide a means to react to events that are discovered from within the community. These events include those generated as a result of the discovery of a new DSO, or a change in a current community DSO. These reactions, controlled by the Community Observer (see Section 5.8.2), allow the system to gather all the necessary information for adaptation reasoning and for the internal structure to gather required information for any subsequent transformation.

There are, therefore, four main components within the DCM. These components are presented in Table 5.3.

5.7.2 Class List Manager

The purpose of the Class List Manager (CLM) is to control the restricted access to all the class reference objects. The class references are stored in the form of `dupe.discovery.DynamicClass` instances created during the target application's execution. This class is designed as a wrapper class for byte code. The CLM contains a Map of DynamicClasses, using the full class names as keys.

The CLM must provide two main public methods: a method to find the details of a class, for example, `public DynamicClass findHandledClass(String`

Component	Purpose
Instrumentation Implementation	The main task of the DCM: the implementation of the instrumentation technique chosen for a specific DUPE middleware.
DynamicClass	An internal representation of a Class which has been instantiated by the JVM.
Class List Manager	A controlling section for the list, or array, of all Classes instantiated by the JVM. Each Class is stored as an instance of DynamicClass.
Restricted Dynamic Class Access	The provision of methods that give other sectors within the DUPE framework access to Class information and instrumentation. This includes the community observation, security and adaptation sectors

Table 5.3: Components of the Dynamic Class Manager

`className)`, and a method that allows it to alter the contents of the list by adding and replacing the contents of the Map, such as, overriding the `public Object put(Object key, Object value)` within the `java.util.HashMap` class. These constructs allow the CLM to act as a centralised database for management of the class details. It can then be used by the other sectors of the framework for class detail evaluation.

5.7.3 DynamicClass

The `dupe.discovery.DynamicClass` is the DUPE framework's representation of a Java class that has been instantiated by a DUPE system's target application. The DynamicClass objects are used within several different sectors of the DUPE framework principally as a means of storage, and for determining the progress of resource updating. The DynamicClass objects are also used for creating the DynamicClassEntry instances for the DSO and achieving security checks. Therefore, there are several attributes within the DynamicClass class which contains key elements of DUPE information. This is shown in Table 5.4.

Name	Type	Description
`classBytes`	*private* `byte[]`	The class bytes of the Java class which the objects represents.
`name`	*private* `String`	The name of the class Java class which the objects represents.
`attributes`	*private* `String[][]` (see Section 7.3)	Attributes, as assigned at compilation, of the class.
`certificates`	*private* `java.security.cert.Certificate[]`	Certificates that have been assigned to the class.

Table 5.4: DynamicClass Contents

The use of the DynamicClass within the different sections of the framework is detailed within respective discussions throughout the rest of this chapter, and in Chapters 6 and 7.

5.8 Community Interaction and Cooperation

The ability for DUPE systems to generate and cooperate within a DUPE community is derived from the use of Jini services. As a Jini service, the DSO exhibits several specific capabilities which play a significant role in the cooperation of systems and the gathering of resources. Within the DUPE framework, there exists a Discovery Manager which is designed to maintain the discovery of other DUPE members by discovering DSO resources and controlling the registration of its own DSO representation.

The framework reacts to the state of the DUPE community by using the DSOs found within the community registry. A DUPE system will use its own DSO to inform the community of changes in its own structure, which in turn will cause community events to be generated. Moreover, DSOs within the community can be used to analyse the contents of their representing systems. The analysis of the DSO contents should be based on the event generation mechanisms that exist as a result of member movements. The community events are listened for by the Community Observer (which may also be known as a Registrar Observer) and generated via the Event Handler sectors of the framework.

All sections of community interaction and observation are grouped together via the Discovery Manager. A trace of community events is shown in Figure 5.16.

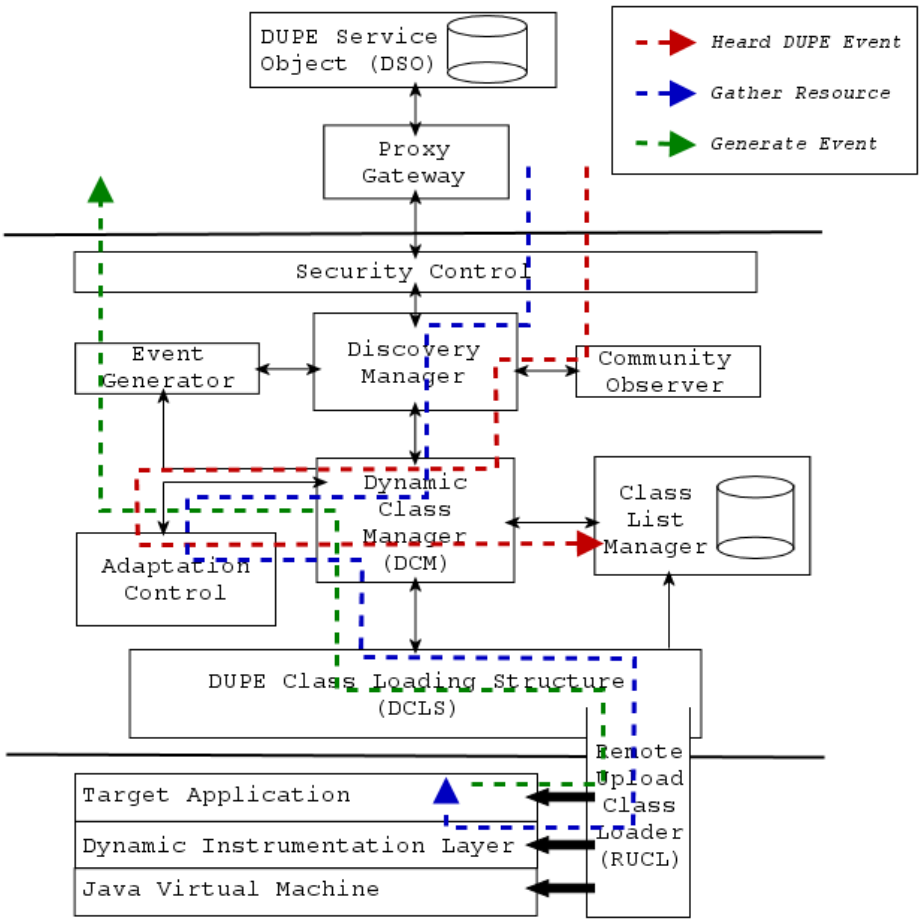


Figure 5.16: Community Event Trace through DUPE Architecture

5.8.1 Discovery Manager

The Discovery Manager is the controlling center of the community cooperation sector of the framework. It has several main tasks. These are as follows:

- discovering the community registrar and other DUPE members
- advertising the DSO within the community registrar

- registering a Community Observer in the community as an event listener
- sending the events created by the Event Generator to the community, and
- controlling the access to both the Community Observer and the Event Generator.

As the center for community interaction, the Discovery Manager is linked to the DCLS and has the ability to initiate the adaptation capabilities of the DCM. This link is important as the Discovery Manager contains both the Community Observer and the Event Generator. Specifically, the Community Observer will inform the Discovery Manager of community events, or those it is set to listen for. The information that caused the event is passed through to the DCM for analysis. We will now outline both the Community Observer and the Event Generator. Further details of their contributions to DUPE Communities are provided in Section 6.4.1.

5.8.2 Community Observer

The Community Observer watches all the aspects of a DUPE systems community interaction by listening for community events. There are several community events that will be heard during community activity[8]. These events are:

- new members joining the community
- members leaving the community
- a change of state in a current member of the community.

There are potentially a large number of events that a DUPE community may generate. This being the case, the Community Observer must run initial tests on each event to check its validity according to security constraints and individual adaptation settings (see Chapter 7).

[8]DUPE community activity and how community events are generated are discussed in Chapter 6.

Insignificant Events

Community events are generated regularly as a direct result of the continual movements of community members and the normal progression of system execution. Therefore, there are many instances when an event should, or can be ignored by a community member. For example, an event may be generated when any community member alters its structure, including, loading or reloading class details. This event generation allows systems running inappropriate versions of the classes to be immediately informed of new versions that appear within the community. However, there will be cases when an event contains no useful information for a listening system. In such cases, the event is heard by the Community Observer and analysed by looking at the reason for the event, and resource contents. If the event is determined to be insignificant, all processing of the resource is discontinued and the event is discarded. The usefulness of an event is determined by using, first, the security measures as defined within DUPE middleware and, secondly, the DUPE system's adaptation settings.

5.8.3 Event Generator

The event generator has a single task: the creation of community events. These events derive from a change in the state of the target system, such as the instantiation of a new class. Therefore, all events are a response to procedures that have been run within the DLCS, and as a result, the Event Generator is informed of this aspect of the DLCS via the Discovery Control, which has a connection either to the DCM or directly to the DLCS.

The contents of the event change are in the form of a DynamicClassEntry. The new DynamicClassEntry needs to be added to the advertised DSO representing the DUPE system. This is achieved by calling the updating method from the Discovery Manager on the DSO using the method `modifyAttributes(Entry[] attrSetTemplates, Entry[] attrSets)` on all the Service(s)[9] that are known by the DiscoveryManager.

[9]Each service implements `net.jini.core.lookup.ServiceRegistration` which provides the `modifyAttributes()` method.

5.9 Implementation Design

We will now provide some general notes for consideration when implementing the framework.

5.9.1 Middleware Architecture

Dupe communities are based on the principle that each member can be any system running within a middleware, and that the middleware is DUPE compliant. The combination of context awareness and a middleware system has been implemented to create an association between system behaviour and encountered environments. For a middleware to gain access to the internal workings of a system there must be a link that provides a gateway into its core. This link is provided through the use of the RUCL. The middleware can then act as a link into the core of a system, particularly from a distributed point of view, as the DSO is then able to gain access to the DCLS via the DCM.

5.9.2 Incorporating a Dynamic Update

In Chapter 2, we discussed the different techniques which expand a system's diversity through the addition of dynamic updating. Some implementations, such as DVM [69] and JDrums [29], rely on the specific use of altered versions of the JVM. These techniques, if applied solely, result in a lack of platform independence. This inflexibility is highly detrimental to small unique devices and their associated operating systems. However, we also decided that although a specialised JVM may be limited in the devices or operating systems that it is able to execute with, it should always remain compliant to Java standards. Therefore, all JVMs used for a DUPE implementation must be able to understand standard Java class bytes (up to the J2SE 1.4 standard), and consequently would be able to correctly execute a standard Java application. As a result of this, any device has the possibility of containing a dynamic JVM of some kind, which in turn, means that a DUPE implementation for that device can be created. And, as long as the JVM is able to understand standard Java code as it should, then any DUPE system using that specific middleware implementation is able to share and gather resources from a DUPE community. However, this does not include limited or specialised JVMs, such as KVM [98] or J2ME [98], as they are unable to execute standard Java byte code.

Limited memory devices, such as PDA's, mobile phones, embedded systems, wearable devices and so on, represent technology inhibited by their inherent platform specific software and environment based interoperability. Obviously, once a device contains a Java Virtual Machine (JVM) there is a high probability that a particular JVM version has been specifically developed for a particular device. Due to compatibility constraints and other complications, it is unlikely, but still possible, that a specialised dynamic JVM such as DVM or JDrums will be incorporated within these devices

The Java standard dynamic techniques, the JPDA HotSpot [103] and Java 5.0 instrumentation [108], are more likely to be incorporated within progressive devices such as PDAs. However, again, the dynamic technique applied to a DUPE implementation is an individual choice. The only requirements for the choice in JVM, or dynamic altering technique, are:

- it must have the ability to dynamically alter, to a usable degree, the runtime code of a system, and
- it must be able to execute a standard Java class file (standard being J2SE 1.4 at the time of this work)[10]

If both of these criteria have been met, then a JVM, or updating technique is appropriate for use as the dynamic alteration method of the DUPE framework. Of course, a limited DUPE implementation can be established if necessary.

5.10 Summary

In this chapter we have discussed the internal structure of the DUPE framework. While many different aspects were discussed, three concepts emerged as the important principles underlying its design: distributed resources, dynamic instrumentation and community awareness.

Distributed resources: provide DUPE systems with the ability to gather code from other systems within the same community. The attributes of the distributed resources determine the cooperation amongst systems, and are

[10] Although Java 5.0 is released at this point it is reasonable to assume backwards compatibility to at least the 1.4 level.

therefore the most specific element of the framework's design. Other sections of the framework can be altered according to implementation and individual requirements; however, the specification of the DSO and its contents must be adhered to, completely. Diversion from the specific structure of the DSO will result in a DUPE middleware becoming unable to cooperate within a DUPE community.

Dynamic instrumentation: is the heart of the individual nature of each implementation of the framework. By controlling the target applications class loading, using the DCLS, and relating it to the DCM, the framework establishes a means of dynamic instrumentation. The flexibility gained by allowing this section of the framework to be individualised provides the unique heterogeneous attributes of DUPE community interoperability. Although the dynamics of implementations are different, they remain able to interconnect as a result of the standardised distributed awareness. Therefore, a system running a specific dynamic JVM, such as JDrums, is able to alter state based on shared resources from a system using, for example, J2SE 5.0 for its instrumentation.

Community awareness: watches the other systems within the community via the community registrar and community events. These observations allow the other aspects of the framework to act according to the state of their current community. The community awareness, contained within the Discovery Manager, allows the framework to act as a community aware framework, and as such, binds the other sections of the framework, and the DUPE system as a whole, to the DUPE community and its DUPE members.

In the next chapter we will further the discussion of the community details of the framework.

Chapter 6

DUPE Cooperation Design

6.1 Introduction

Continuing the theme of the previous chapter, this chapter discusses how the design of the framework facilitates the cooperation of ubiquitous heterogeneous systems and establishes DUPE community communication as a specialised communication technique. During the chapter, we introduce several new concepts that evolved during the development of the DUPE framework. These concepts are:

- **System alteration deposits**: by depositing an updated version of a software resource(s) on any system within a community all other members can adapt.
- **Context (location) specific functioning**: by placing a software resource(s) on a DUPE system in a specific location, for example, a tourist destination, all DUPE members that enter the proximity can adapt to location specific software.
- **Evolution transfer**: the movement of mobile DUPE systems throughout different DUPE communities can establish a new means of transferring system components, and result in system evolution. The term *Gypsy Agent* is used to describe such a mobile DUPE system.

The actions of DUPE systems which result in dynamic community cooperation are common within all of the above situations. The chapter will therefore begin with an in-depth analysis of DUPE communities. Following this, each of the

key elements of the framework communication will be discussed. The details will demonstrate many of the major contributions of this work, and indicate where the DUPE framework fits in terms of our initial discussions on dynamic systems, distributed cooperation, and dynamic communities.

6.2 DUPE Communities

DUPE communities systematically allow the sharing of code structure, specifically they share Java byte code. And, the framework's design stipulates that a system should, where possible, be able to dynamically update. The result of this, is that the state of DUPE communities change in accordance with the current status of their members.

We have discussed how DUPE middleware can be used to execute individual systems and so bring them into DUPE communities. Therefore, as a result of the compatible DUPE middleware, all intercommunication details of the community are understood by all members. This communication level is based on a single common question: *What can you teach me?* In terms of a DUPE community the language of interaction is essentially: *What new, or more advanced, code do you use that, if I use it, would help me improve my community experience?* or, *What code do you have that I can use too?*

In Section 4.2, we described the language of dynamic community communication. If we look at technique applied for communication within a DUPE community, as shown in the previous chapter, it is easy to note that it applies to the constructs of dynamic community communication. Imperative to this is the use of DSOs for discovery of services and transfer of code. A DUPE system can discover a DSO, analyse it for required information and if applicable, call for a resource's Java byte code. The implications, and in fact, the structure of the byte code (other than that it has been verified and that it corresponds to a specified class identification: the class name) are completely unknown to the DUPE system until execution has begun.

DUPE provides systems with greater flexibility and adaptability through the use of the following attributes:

- J2SE 1.4 code sharing

- dynamic code changes within system classes during runtime
- continuous type safety[1]
- extension of the JVM's access, via resource discovery, to class resources, and
- dynamic resource access within any DUPE community.

There are several aspects of a DUPE community that differentiate it from a standard set of client/server applications. For example, in Chapter 4 we established several aspects of dynamic community system adaptation. These aspects directly affected the design of the framework. We also established the requirements for a dynamic community. We will re-introduce these requirements, briefly, as they contribute to DUPE community interaction.

- **Mobile Interaction**:
  Mobile members of the community are the devices within a network which pass through on a regular basis. This gives the community an extremely ad-hoc and dynamic nature.
- **Interchangeable Components**.
  Numerous devices interact within a dynamic community. Each device uses a particular operating system and, in many cases, a specialised JVM. The use of a framework that can be adapted to a system is preferred to a one where the system must adapt.
- **Interaction and Communication**.
  A mobile, wireless, ad-hoc environment may contain multiple methods of communication and different levels of system interaction. As discussed in Chapter 3, the best suited means of communication for a dynamic structure based on location proximity, is wireless.

6.2.1 DUPE Compatibility

A DUPE compatible system runs within any version of a DUPE middleware, the version best suited to their dynamic capabilities. Therefore, the middleware implementation used determines the systems use of the community. For example,

[1]This feature is dependent on the technique applied for dynamic system instrumentation; however, we assume that the chosen technique has proven type safety capabilities.

one may provide all DUPE capabilities, while others may be designed within limitations, such as limited adaptation. However, irrespective of the middleware, once a system becomes fully DUPE compatible it will demonstrate two main characteristics:

1. **Community Interoperability**.
   All systems with a DUPE compatibility are, at the least, able to discover other systems within a DUPE community. Generally, most systems will also advertise themselves within the community. DUPE makes use of Jini for discovery purposes.

2. **Dynamic Alteration**.
   All complete compatible systems are dynamically adaptable. This is essential to a DUPE community as the community is the result of the sharing of resources and the cooperation of all interacting members.

The design sections of DUPE that allow it to interact within communities are depicted in Figure 6.1. How the design of the framework provides systems with

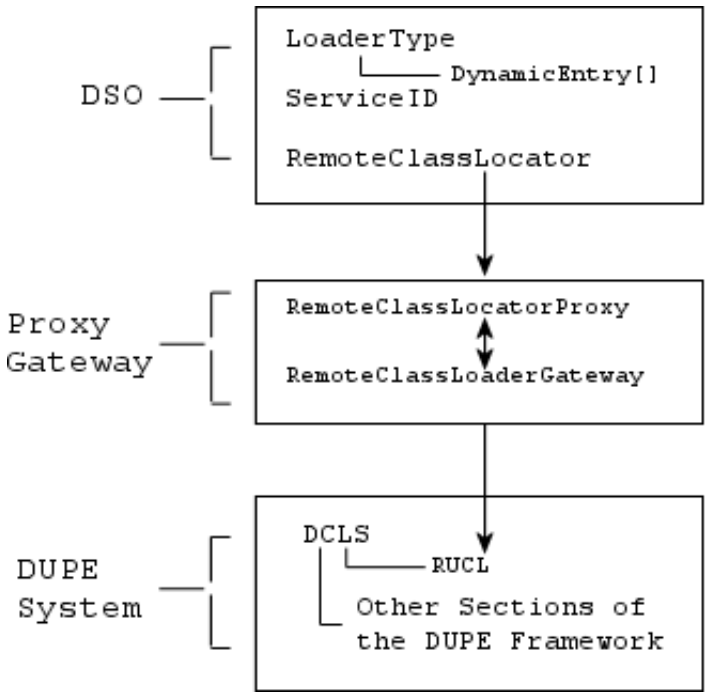


Figure 6.1: DUPE Framework Community Interaction Constructs

access to community resources can be seen in this figure.

6.2.2 Community State

Any updating technique used for an implementation of DUPE will change the system based on events that are generated from the community. And, as a community is governed by its interacting members, all adaptation is generated to some degree from within a system, or multiple systems. This in turn, means that a system will dynamically alter its state according to the other current members of the community.

The combination of its dynamic nature, and the ad-hoc movements of interacting systems, generates an always current community, where new systems seamlessly give or take resources as required for specific operations. We recognise this particular attribute as the community state. In addition, the creation of a (seemingly) stable version of the components within a community; a situation where a majority of members of the community are executing the same component versions, is termed the community state. The alteration of one component within most members of the community initiates a change in the state of the community.

6.2.3 Adaptation

Within a DUPE Community a member will not only alter its behaviour based solely on its need/want for adaptation, but it should also take into consideration the needs and wants of the community as a whole. The level of change is completely determined by a system's adaptation controls.

DUPE's adaptation techniques are based on three main reasons for behavioural adaptation; furthermore, these reasons are all directly determined by the community's event mechanism as follows (Figure 6.2 provides pictorial reference for each situation):

- An entering system can alter its behaviour according to the current community status (Figure 6.2: (a)).
- A current member of the community is alerted by community events that a new resource version is available from the community. This action may result from the entry of a new system as in the previous situation (Figure 6.2: (b)).

- An entering, or current, system can cause the community to generate an event to notify that it contains a new resource version[2] (Figure 6.2: (c)).

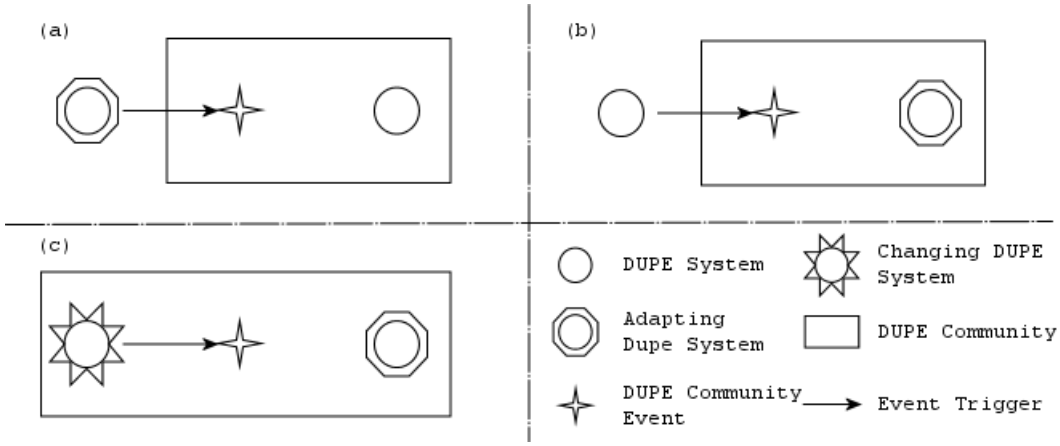


Figure 6.2: DUPE Community Events and Subsequent Member Adaptation

Figure 6.2 shows DUPE members during the different means of adaptation that occur within a DUPE community (these events are a DUPE interpretation of the general community member movements previously shown in Figure 4.2). Again, it is noted from all these situations that, in effect, a DUPE community is governed by its members, and therefore, all adaptation is generated by a member, or multiple members. This in turn allows the community to dynamically alter its state in accordance with the members that are currently stabilising its structure. The event based adaptation allows adaptation to be achieved as a response to the needs of a member.

6.3 Using the DUPE Service object

The DSO contains the necessary information for a searching system to be able to deduce the usefulness of the resources available from the advertising system. Therefore, the usability of a DSO service is defined by the resources (classes) it makes available, and their associated attributes. Consequently, identifying the usability of any resources is an individual choice. A DSO also indicates if its advertising system provides access to local resources that it has not executed itself. However, as the location and uploading of class details in this manner may

[2]If the resource is brand new to the community an event can still be generated.

become time consuming for the advertising system, this feature can be disabled by the user of the advertising system.

The DSO contains compact descriptions of available resources. This aids in maintaining low network traffic within the community. A DUPE member advertises all the resources it has at any period of time within the DSO. Therefore, a searching system needs only to ask the service advertised to check the availability of a class. This means that a DUPE member will not communicate directly with a system until it has located a DSO which contains a class it is looking for.

As indicated in Section 5.6.3, all details within a DSO are contained within a collection of DynamicClassEntry objects. As already detailed, the DynamicClassEntry is a representation of a class resource presented by the DSO. As a service representation, the DynamicClassEntry contains no class bytes or instantiation information, it only gives the class name and the information used for adaptation reasoning and security purposes. (The details of each DynamicClassEntry were listed in Table 5.2).

Moreover, as each DSO is a Jini service advertised by a DUPE member, it is able to negotiate the transfer of component details. Most importantly, this includes the transfer of class bytes. The exact use of the information provided by the DSO is specific to each framework implementation and the adaptation settings applied by the user. The following is a guide to how the information provided by the DSO can be used[3] (although alterations on these may be appropriate depending on implementation requirements).

- The `classname` can be used as an initial search to check to see if a resource is appropriate for use within a system. This use of the resource remains consistent throughout all implementations of the framework.
- The `attributes` are used to determine if a resource, once found to be useful with a system (see above), should be loaded into the system in accordance with the adaptation control setting for the executing system. The use of all `attributes` should remain consistent. Its use in determining adaptation is based on an individual execution.

[3]Attributes and certificates are dealt with in detail in Chapter 7 as they are related to the frameworks security and trust management.

- `Certificates` are used within DUPE to maintain a trusted level of security. Again several settings are automated and several may be refined according to user requirements.
- The `allowInternalSearch` boolean should be used minimally. Searching for undisclosed resources by a remote system will cause noticeable delays for the host system. Particularly, to those executing in mobile devices.

Once a resource is found to be applicable to a system for use during adaptation, the complete class details can be gathered and loaded.

6.3.1 Discovering and Advertising DSO Services

Correct advertisement of a DUPE system's DSO and the knowledge of how to discover another DUPE services DSO is an integral part of DUPE community interaction. Conveniently, it is also relatively simple. This simplicity is a result of the flexibility of the single question protocol used within a DUPE Community.

Advertising a DSO: The DSO is an advertised instance of a Jini service item (`net.jini.core.lookup.ServiceItem`) that contains the required aspects of a DSO as described in Section 5.6.3, which are:

- the DUPE System's ServiceID
- the DUPE Service's RemoteClassLocator instance
- the current DUPE Service's LoaderType object.

Each DSO must contain these required items. Moreover, the `LoaderType` object within the DSO must be kept up-to-date with the current state of its home DUPE Service. The class structure of the DSO was provided is Section 5.6.3.

Discovering a DSO: Each DUPE system understands the RemoteClassLocator interface, as it is necessary for the DSO object. Therefore, using this interface, all DUPE systems are able to identify a Jini service as a DSO. This process is the only way to discover a DSO, it should be used in combination with a `net.jini.core.lookup.ServiceTemplate` object [79]. The process for a client to discover and obtain all the required information from a DSO is as follows, and an example implementation is provided in Figure 6.3:

- upon discovery of a DUPE community registrar, the client system will obtain a reference to all Jini Services which are an instance of RemoteClassLocator interface.
- the client system will check the ServiceID of the DSO for repeated searches
- the client service can then gather the LoaderType object from the DSO and analyse the available DynamicClassEntry objects for adaptation relevance.

```
//create a ServiceTemplate based on the DSO Specifications
Class[] classes = new Class[]{RemoteClassLocator.class};
RemoteClassLocator remoteLocator = null;
Entry[] loaderEntry = new Entry[]{new LoaderType()};
//Service Template Creation
ServiceTemplate template = new ServiceTemplate(null, classes, loaderEntry);

        ... //discover DUPE Community(s)

//look for DSO Sevices using the Service Template
ServiceMatches serviceMatches = registrar.lookup(template, MAX_DSO_MATCHES);
final ServiceItem[] serviceItems = serviceMatches.items;  // all DUPE Members
for (int k = 0; k < serviceMatches.totalMatches; k++) {
  //check to make sure it is not this Service
  if (!serviceItems[k].serviceID.equals(remoteCL.getServiceID())) {
        final RemoteClassLocator foundService = (RemoteClassLocator) serviceItems[k].service;
        addRemoteLoader(serviceItems[k].serviceID, foundService); //add to known members list

        //analyse service attributes using observer object (possible new Thread)
  }
}
```

Figure 6.3: Example Code: Discovering DSO

Although, this is an extremely simple discovery and advertising protocol, it provides the flexibility for the unknown protocol responses required for dynamic community communication (see Chapter 4 for details). The only known parts of the DSO are the interface RemoteClassLocator, which provides the `generateDynamicClass(...)` method that allows direct class byte gathering, and the LoaderType in which there exists an unknown quantity of unknown instances of DynamicClassEntry objects.

We now describe the methodologies that should be applied for gathering all resource information from the DSO. This specification includes the aforementioned `generateDynamicClass(...)` method.

6.3.2 Gathering the Complete Resource

The process of gathering a complete class resource is again relatively simple. There is only one question that a DSO can be asked of that will result in it contacting its host system:

Can you send me a DynamicClass object containing the byte code for (insert class name here)?

As indicated by this question, the return result of the remote method call is an instance of a DynamicClass, and, as we have already discussed, a DynamicClass contains the byte code of the class it is representing.

The method used to gather the class details is part of the `dupe.discovery.DynamicRemoteAccess` interface, which is implemented by the DSO via the RemoteClassLocator interface. Specifically, the method `generateDynamicClass(...)` (shown in Figure 6.4) should be executed.

```
public DynamicClass generateDynamicClass(String className) throws
                              ClassNotFoundException, RemoteException;
```

Figure 6.4: Remote Method Call for Final Resource Gathering

This method should be called by the gathering system using the class name that was found during initial DSO analysis. The returned DynamicClass should be reanalysed to ensure its contents are valid according to the initial DSO information. Once the DynamicClass is found to be trusted, according to security (see Chapter 7), it can then be passed to the Dynamic Manager for initialisation or redefinition.

6.3.3 Sending a Resource

After a DUPE member acting as a client invokes the `generateDynamicClass(...)` on another DUPE member it will wait for the requested resource to be returned. The return of the class is a responsibility of the DUPE member acting as a server. When the method is called with an appropriate class name[4] the server system

[4]The use of a class name gathered from the respective DSO is appropriate. Invoking the generateDynamicClass method using a class name that does not exist on the host DUPE member's DSO is inappropriate and dismissed.

will return a corresponding DynamicClass using information from its Class List Manager.

6.4 DUPE Community Interaction

6.4.1 Community Events

The DUPE community makes use of the Jini architecture as a discovery mechanism, and subsequently, it is able to make use of several Jini events. These Jini events are generated as a result of several different community situations [79, 36]. In terms of DUPE communities these situations are:

1. a change in structure of a community member
2. the entrance of a new member into the community and
3. the exit of a current member of the community.

All events are a result of a change in any community member. However, as a DSO maintains a current representation of a community member it is here where the community events are actually created. Therefore, when a system alters its state, for example, loads a new class or redefines a current class, it generates the community event by altering a DSO. Accordingly, as a Jini service [79] the DSO will then create the community event. This event will be heard by all other community members, and as a result they will react accordingly. All events generated within the community contain enough information regarding the changed conditions to allow all listening members to initially react without contacting the community registrar, the DSO or the initiating system. Moreover, events that are a result of a change in a current DUPE member are a result of an update of the LoaderType object within a DUPE members associated DSO, as shown in Figure 6.5.

The trigger for this type of updating process is a direct result of a change in a DUPE system, and is seen as a subsequent change in its Class List Manager's DynamicClass list. The DUPE Community events form the basis for the adaptation of all DUPE systems.

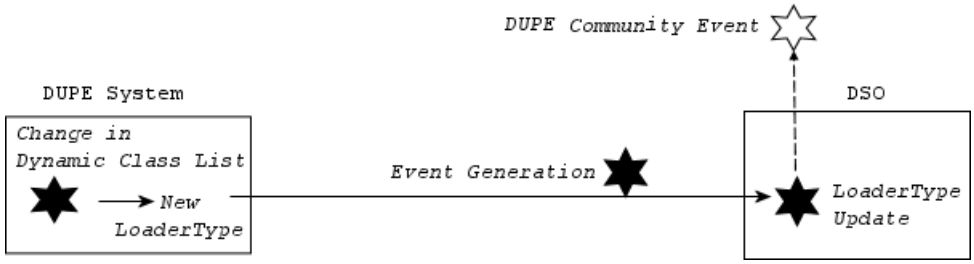


Figure 6.5: DUPE Community Event from DSO Update

6.4.2 Member Cooperation Design Considerations

To join a DUPE community a system must become DUPE compatible. To become DUPE compatible a system is run within a DUPE compatible middleware. The middleware then controls all interaction within the community.

All searching and verifying is achieved within the DUPE middleware section of the DUPE system. And, as each middleware implementation is either developed differently or used differently by the user, all DUPE systems determine their own level of cooperation within the community. However, all DUPE system communication must be achieved via DSO interaction.

For this degree of cooperation, the discovery based nature of the framework allows resource discovery to be dynamic. This is true to the extent that resources may be unknown during initial execution stages, and as required, must be individually discovered from within a current community. This is different to other discovery techniques, such as RMI, where subsequent classes must be locatable via a specified codebase. Using DUPE all Java systems can adapt even though they were not originally designed to. The following is a list of the key attributes of the DUPE framework which allow community cooperation:

- **Remote System Description and Proxy:** there is no need for a DUPE system to create objects for each resource it has to offer. All information regarding all its resources is located within its advertised DSO.
- **Anonymous Distribution:** all discovery, advertising and use of resources within a community are achieved anonymously.
- **Multiple Class Locations:** DUPE members are able to locate their class components from multiple locations. These locations include: local class

files, classes within the community and classes located according to system specific needs.

- **Dynamic Discovery of Members:** all DUPE members can discover all other members of a community. There is no need for a member to know how to directly contact other members. A system need only understand what resources the other members have and how they may obtain them. This is described by the DSO and the DCLS.
- **Community Event Generation:** this is the ability to register a member within a community to listen for the details of all community events. As defined in the previous chapter, these events include:
  1. entry of a new member
  2. change in resource of a current member and
  3. exit of a member.

Along with these attributes, there are also several design considerations that have been applied to ensure safe use of the framework. These are a direct result of the extended system interaction available to DUPE systems. These are:

- **Controlling Dynamic Alteration:** the dynamic alteration is designed to maintain system code. However, without security and adaptation controls any dynamic adaptation would be unlimited and extremely unsafe. The dynamic adaptation is, therefore, controlled by both of these sections. This is discussed further in Chapter 7.
- **Maintaining Security and Adaptation Controls:** DUPE security controls make sure that all gathered remote components are safe. And, its adaptation controls enable the framework to reason any adaptation to a community. These controls should be set as default settings but left as alterable by the user. Examples of adaptation reasoning include number identification based version update and vendor specific alteration. This is discussed further in Chapter 7.
- **Controlling the Target System:** as a middleware, DUPE controls the execution of a target system. In fact, the DUPE framework maintains slightly more control over the target system than is normally applied by a middleware application; for example, DUPE middleware controls the entire execution of an application, whereas other middleware only provide extended capabilities. The framework stipulates that all class loading after verification is managed

6.5 Requirements of DUPE Compatible Middleware

Based on the framework design presented in this chapter and in Chapter 5, we now provide the requirements that must be met for a DUPE implementation to be recognised as DUPE compatible. There are two levels of compatibility for each of these requirements.

Mandatory: a section of the framework which *must* be included for a DUPE compatible middleware.

Standard: a section of the framework which is generally included for a DUPE compatible middleware. If this section is excluded, the DUPE implementation will cooperate within a DUPE community at a limited level. All such DUPE implementation are limited DUPE middleware.

The requirements are provided in Table 6.1.

6.6 New Concepts due to DUPE

Thus far, we have established both the design of the framework (Chapter 5) and, in the early stages of this chapter, its use as a distributed cooperation tool. The remainder of this chapter provides a detailed look at the beneficial contributions that the framework may provide to systems. We determine a beneficial contribution to be a technique that has either been made easier to apply as a result of the framework, or something which only the framework can offer that benefits the use of a system, or the cooperation of systems.

Principally, there are three new concepts that have arisen as a result of the DUPE framework. Although, each of these concepts differs in what it achieves, as mentioned earlier in this chapter, all rely on the unique system cooperation that is enabled within DUPE communities. Two of these concepts: *System Alteration Deposits* and *Context Specific Functioning*, are a result of the use of DUPE communities, whilst the other, *Evolution Transfer*, is linked directly to our concept of gypsy agents.

We will now present these new concepts as contributions to the field.

Section	**Description**	**Inclusion**
DUPE community discovery	The ability to locate and recognise a Jini Registrar as a DUPE community	Mandatory
DSO Understanding	The ability to understand the contents of a DSO.	Mandatory
DSO Creation	The ability to create and advertise a DSO object.	Standard
DSO Discovery	The ability to discover DSO objects within a DUPE community.	Standard
Load time dynamic code redefinition	The ability to alter the state of Java byte code during start up.	Standard
Runtime dynamic code redefinition	The ability to alter the state of Java byte code during runtime.	Standard
DUPE Event Recognition	The understanding of a DUPE community event and its contents.	Mandatory
DUPE Specification Understanding	The understanding of the contents of DUPE specification details associated with DynamicClass and DynamicClassEntry objects.	Mandatory
Certificate Recognition	Differentiating between an illicit or unwanted certificate.	Mandatory

Table 6.1: Requirements for DUPE Compatibility

6.7 System Alteration Deposits

There are several situations where the placement of a system component within a community results in a beneficial change in the state of resources. Much of the discussion that we have had thus far considers *ad hoc* changes within communities and systems. The situation presented during this section demonstrates the use of DUPE in a *planned* scenario. We will describe how the introduction of a resource into a community can be for the purpose of generating an advantageous adaptation of interacting members.

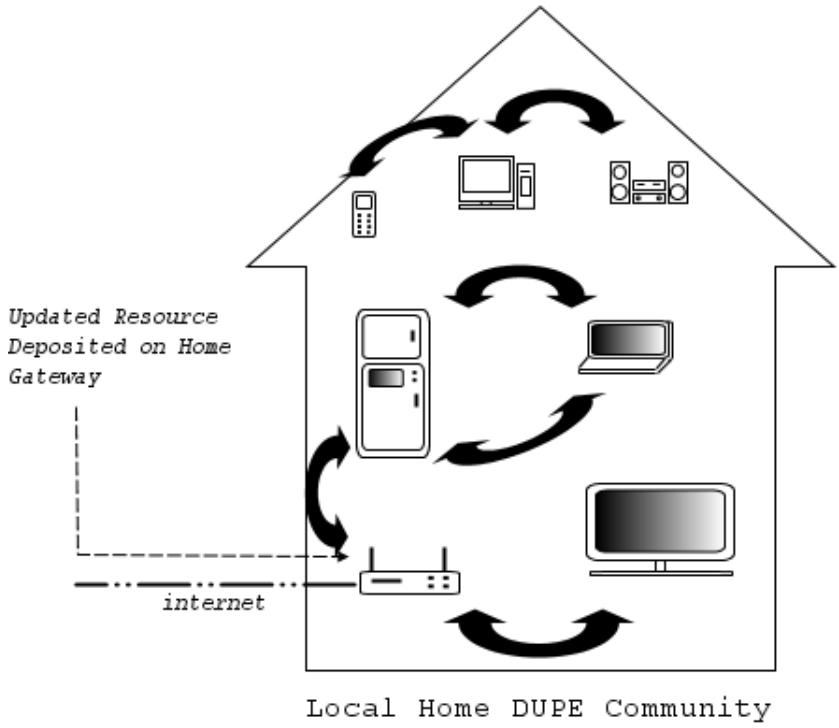


Figure 6.6: System Alteration using DUPE Community Resource Deposit

When introducing a new version of a resource to a community it can be placed within any DUPE member. There are scenarios that may be established using this technique:

- a mobile system could purposely enter the community and introduce the resource;

- the component could be located in a new system that is added as a permanent part of the community, for example a new HiFi appliance, and
- the new resource could be introduced to a current system within the community, for example a home gateway, as is shown in Figure 6.6.

The community alteration achieved by the introduction of the new component will proceed irrespective of how it was introduced into the community. Therefore, the most appropriate means of introduction is that which best suits the situation. The following situations demonstrate how this can be achieved. (The situations also provide the scenario for part of the analytical evaluation in Section 11.). We will now detail all the possible introductory scenarios.

6.7.1 Network Component Change

The alteration or upgrading of a system that is continuously interacting within a standard local network, such as a home network, could possibly cause some incompatibility with other systems. Specifically, it is likely in that, after an upgrade, one or two systems may no longer cooperate with each other. However, in a DUPE community the introduction of an upgrade component would initialise DUPE event mechanisms. Consequently all systems would signal that there is a new component available. If appropriate, the systems would obtain the component and adapt according to any adaptation rules. For example, in Figure 6.6 the deposited resource could be a new version of the communication component: `communication.CommTCP`. Therefore, as all of the current members of the community update their components, no member would become incompatible.

6.7.2 Defect Upgrade

There are constantly recalls and calls for system upgrades to be downloaded for systems. For example, it is very common to upgrade a system on a mobile phone to a new version. This upgrade may be to repair a defect, fix a security flaw, or, simply to improve performance. Upgrading a system in this manner is completely at the discretion of the user, and therefore, it is difficult for a distribution company to ensure the changes are completed.

A vendor is able to place a system upgrade on an access gateway to a home network that is running within a DUPE community; for example, this would be

achievable using an OSGI type arrangement. This allows the distributor to alter the components of any specific systems interacting in the community without ever having to directly access the system. For example, in Figure 6.6, the deposited resource could be an altered version of the reception package `appliance.reception` designed for appliances such as televisions as a fix to a defective component.

6.8 Context Specific Functioning

During the discussion on context awareness in Chapter 4, we illustrated how a system can react to its surroundings by changing its execution flow according to what context it has encountered. The scenario in this section, details how the DUPE framework can work in a similar fashion. However, DUPE creates a more flexible means of system awareness. Moreover, it has the ability to allow members to *adapt* to their community, based on these components.

The DUPE framework creates flexible, adaptive, community cooperation among systems. This level of cooperation differs from that achieved by context awareness in that, unlike a context aware system, a DUPE system does not alter its behaviour based on its current capabilities, or the encountered context (context in this case being community resources), it alters its progression based on other systems and their capabilities. Therefore, within a DUPE community, resources may be used by a system as a means of contextual adaptation.

6.8.1 Resources as Context

The resources that are offered by a DUPE member do not have to be executed by the system itself. For example, a server could act as a depository for newly updated, or location specific, versions of code. Therefore, it is possible to maintain a system that contains a set of resources that are specific to a location. This makes it also possible to have a set of resources that are available within several different communities which achieve different tasks in accordance to each specific community. Using DUPE in this way enables systems to implement in a specific manner according to the community it is currently interacting within.

An example of this level of use is a tourist application. A tourist application may be created that requires a particular resource, for example a `tourist.resources` package. This package may be designed differently according to each specific

tourist destination. If the adaptation rules of a DUPE system are then set to location based settings, when a tourist system using any DUPE middleware enters a tourist destination, it will adapt to its specific resource(s). This scenario is depicted in Figure 6.7.

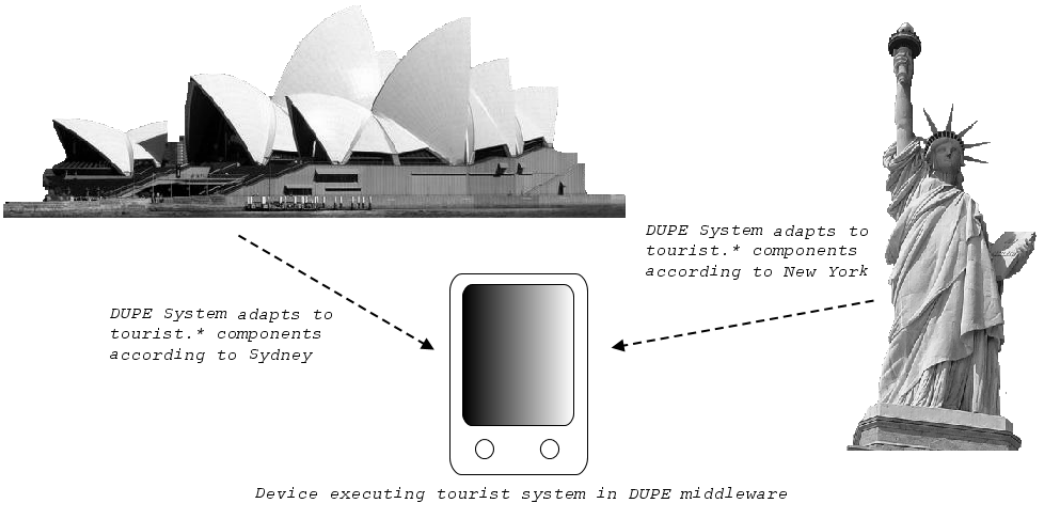


Figure 6.7: DUPE System Location Context Adaptation

This type of system adaptation is more than simply transferring data, such as showing the menu of the closest restaurant, or providing tourist specific textual information. Using DUPE in this way enables systems to act in a completely different manner according to their proximity. At one destination, the system may in fact be a menu for a restaurant, however, at another it might become a self guided tour complete with guiding voice and pictures (possibly triggered via other context systems, for example, a historic application providing details as the system moves throughout the area.). The possibilities are only limited by the resources available from within the current community.

Further to this scenario, the resources that act as context do not have to exist within the DUPE system. A deposit only version of DUPE that contains one or more resources may be created to act as a DUPE context point. For example, a light post in the middle of a town may contain a small embedded sensor device that is advertised with a registry somewhere in the vicinity. The light post may carry tourist information such as, a specific section of a mobile guided tour, or, the details for emergency contacts for that particular area. These resources could

then be used by any DUPE system entering the local community.

Furthermore, a DUPE system that was designed to deposit only, would never adapt to the other DUPE members. It should only be updated via, for example, direct access or secure access adaptation. This is because, as its sole purpose is to allow other DUPE systems to adapt on the basis of locality, not for it to adapt to them.

6.9 Evolution Transfer

In a DUPE community each system shares its components with other systems, mobile or fixed, within a particular (location based) proximity. We will also further explore, in Chapter 7, how the adaptation rules of DUPE systems are determined by the restrictions placed on each individual system, and how, consequently, the nature of each DUPE community structure will continually alter.

We specify that each *mobile* member within a DUPE community is essentially a mobile device running a DUPE compatible system. And, because of the nature of the movements of a mobile device, it is possible that these systems can become a mobile means of software transfer. By transferring software, a mobile system demonstrates similar attributes to that of mobile software agents, but, as a physical rather than a logical entity. We determine this characteristic to be the action of a *physical mobile agent* and propose that the likely increase in the number and movements of these physically mobile agents can be beneficial. For example, when a mobile agent discovers a community, it is able to gain and give system information. We have named these agents: *'gypsy agents'*. Gypsy agents cooperate on non-biased, no return required, barter terms. This concept was initially explored in Section 4.4.3 and we will now discuss it further in terms of the DUPE systems.

6.9.1 DUPE Gypsy Agents

Any system that is running within a DUPE compatible middleware, or is DUPE compatible itself, is termed a DUPE system. We have reiterated throughout our discussions that an implementation of the DUPE framework can be created for many different reasons; for example, an implementation may be specific to a particular JVM, or it may be created as a large system so it can handle server

type operations. Another implementation of the framework, one which we have detailed as a key implementation, is designed specifically for mobile systems.

We believe that mobile devices have the most to gain from the use of DUPE. They move between communities on a regular basis and consequently come into contact with many different component versions. Moreover, as they move between communities they will take components to each new community; the mobile system transfers the state of one community to the next community that it encounters.

6.9.2 Sustaining Community System State

Gypsy agents have the ability to move between many different communities and as they do they present new resources for the DUPE members of each community. And, they are also able to gain resources from each of the communities they encounter. This is a result of the gypsy agents retaining the resources they themselves have used to adapt[5]. As they move, they will introduce these resources to all new communities. The ability to do this is a direct result of system adaptation in the gypsy agent. The events provided by a DUPE community give all members the possibility to gather information from other members and do with it as they please, including keeping a copy of it. The use of this attribute is exclusive to gypsy agents and can be used as a means of system evolution and state transfer.

6.9.3 Spread of System Resources

We have now established that, as gypsy agents move among communities they can take with them the resource(s) they have gathered from previous communities. If the resource is new, or a more appropriate version of the one currently running within the new community, all its members will adapt accordingly. This includes other gypsy agents present in that community.

All gypsy agents will continue to move among communities, and as a result, so will the new version of the resource. We see this as a positive spread of a system resource, and we believe that it can be used as a means of achieving tasks, such

[5]The retention of community resource details is determined by the framework implementation or the specific user settings. Some systems may keep resources they gain, and are hence able to pass them on, while others may drop resources as they move.

as system upgrades and debugging. An example of the effects of the movements of gypsy agents is shown in Figure 6.8.

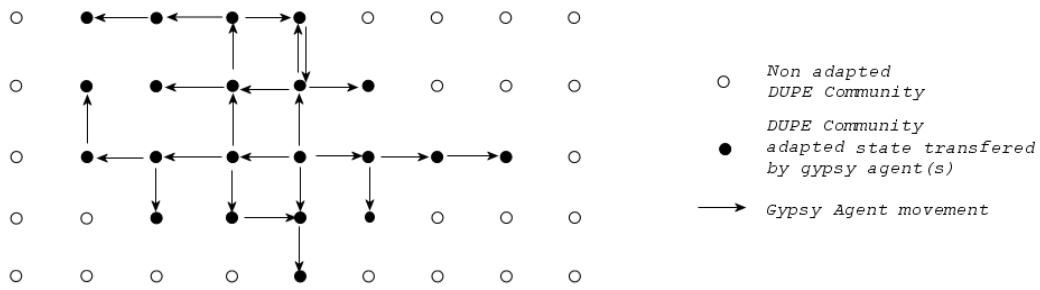


Figure 6.8: System Resource Transfer via Gypsy Agents

However, we also note that such attributes may be used for the spread of illicit community resources. To counter this, the security and adaptation controls (see Chapter 7) are designed to keep the impact of such problems to a minimum. The rate of spread achieved determines the usefulness of gypsy agents. However, as the movement of gypsy agents is linked to the movements of the humans carrying and using their devices, their use may only be predicted and analysed using modeling and probability. We will discuss the development of these models in Chapter 12.

6.10 Summary

We have now identified the attributes of the DUPE framework which enable it to establish DUPE communities. We have also indicated the three main contributions of DUPE communities and their interacting members. The three possibilities presented by the framework that are seen as major contributions are:

- **Systems Alteration Deposits**: the introduction of an updated version of a software resource(s) to a system within a DUPE community enabling all members to adapt.
- **Context (Location) Specific Functioning**: using DUPE enabled systems and their associated resources as a way of detailing the context of a location.

- **Evolution Transfer:** the movement of gypsy agents establishing a new means of transferring system components and enabling system evolution.

We have analysed these contributions and discussed how, by using the DSO and `DynamicClassEntry`, the framework remains flexible in its design, yet stipulates, precisely, its core cooperation specifics. The precise description of the communication within the DUPE community is necessary to ensure that DUPE systems are able to cooperate (for more details on the DUPE communication see the DUPE Specification in Appendix A). This in turn allows different framework implementations to form DUPE communities for heterogeneous systems.

All of these contributions will be assessed further within the rest of this thesis. Systems alteration deposits and context (location) specific functioning are used during the evaluation of the framework in Chapter 11, and we return to evolution transfer during our analysis of the effects of gypsy agents, through the use of epidemiology modeling, in Chapter 12.

Chapter 7

Security and Adaptation

7.1 Introduction

DUPE communities can potentially create a new means of interfering with systems. We acknowledge that system adaptation to remote code components entails a threat, in that any resource may be a virus or other detrimental code. We also acknowledge that the movements of gypsy agents are potentially a perfect means of virus transfer.

This chapter assesses these precarious aspects of the DUPE framework. How we deal with this aspect of the framework is separated into two categories:

- **Security**: the use of techniques such as digital signatures and certificates to provide a means of security and trust to DUPE members. We establish that signatures can be applied as a means of masking data from intrusive alterations during the transfer of resource information, and that this provides benefits to security and trust levels.
- **Adaptation Controls**: the use of user specific settings to govern system adaptation and evolution. The setting rules are based on the information that is obtainable from a DSO and DynamicClassEntry objects. It describes the attributes such as version numbering, and vendor and location identification.

Firstly, we assess the security risks. This assessment takes into account the different types of unwanted attacks and the associated problems that are likely to

be a hindrance to DUPE communities. The chapter will then discuss adaptation controls and explain how the use of class attributes, within the user preferences of the framework, are used for controlling how a system adapts to a community. Importantly, the framework's security technique must provide verification that remotely obtained class bytes remain unchanged from their initial state, and that they can be deemed to have originally come from a trusted source. This will help ensure that malicious code is not imported and executed by a DUPE system.

Throughout the discussion in the chapter, it is important to remember that, as noted in Chapter 5, not all Java files are, nor ever should be, loaded via the DUPE structure. Examples of such files include: standard J2SE packages and the group of `sun.*` packages.

7.2 Security

As DUPE systems use unknown class files from anonymous systems, there is continuously the possibility of the intrusion of malicious code. For example, during system adaptation, discovered resources are included and executed by a system as a result of its community interaction. It is possible that an unchecked malicious component may make its way into the system. Such a component could achieve endless alterations to the system, or the device on which the system is being executed. This potentially includes the possibility of complete system deletion. Consequently, DUPE systems must protect themselves from gathering unwanted code in place of a legitimate resource.

To avoid such situations, it would be possible to use a sandbox method similar to that applied to Java Applets [42]. However, a sandbox restricts the flexibility of the DUPE framework in terms of community cooperation, and limits the functionality of the target application. For this reason it is not applied for the DUPE framework.

Unfortunately, as we will demonstrate, a sandbox level of security may be the only way to completely secure the DUPE framework. To support this view, we will clearly identify the security levels of DUPE as a limitation within the framework. Either the sandbox method is applied and DUPE members are limited in their interaction capabilities, or, DUPE members remain flexible yet maintain a

slight possibility of potentially harmful, intrusive code.

Trust can be used by a system to determine the appropriateness of a resource. In this situation, trust acts as a level of security.

To account for security we have applied levels of community trust and system security to certify the safety of community interaction while maintaining the flexibility of the DUPE framework. The placement of the security in the framework is illustrated in Figure 7.1.

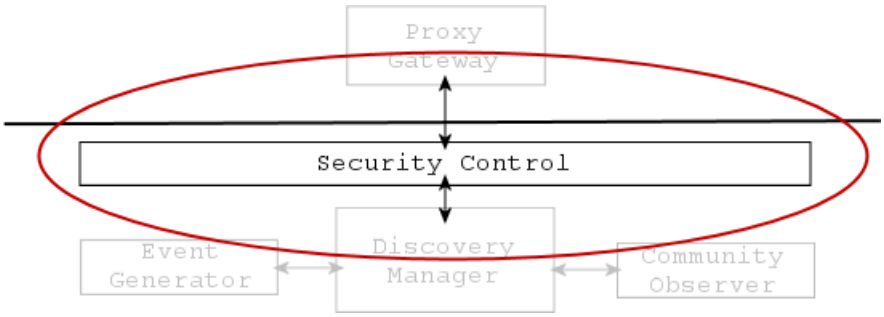


Figure 7.1: Security Section of the DUPE Architecture.

7.2.1 Vulnerabilities

There are many different types of attacks carried out on systems connected to any network. These attacks, or problems, are generally described as viruses or malicious code. The code sharing nature of DUPE opens a system up to a new means of acquiring malicious code. However, the problems associated with DUPE systems are similar to those experienced in internet interaction. We identify the main problem areas that are exposed by the use of DUPE as:

- **Spoofed DUPE members:** in this instant, illicit members of a community pretend to be valuable member of the community in order gain access to other members. This might be achieved by using, fake class substitution, or Class Trojan Horse techniques.
- **Class Trojan Horse:** a trojan horse is designed to represent a specific class (resource). A DUPE member may decide to include what seems to be

a useful and innocent class, however, the class contains additional hidden code which allows the unauthorized collection, exploitation, falsification, or destruction of data.

- **Direct Attack:** in this case, an attack on a DUPE system may be achieved by trying to gain direct access via the DUPE middleware. This is a standard networking problem and not specifically associated with the DUPE systems.
- **Virus Transfer:** the use of the movements of gypsy agents to transfer viruses throughout different systems. This scenario is potentially the most damaging to systems, as it allows illicit code to be transferred throughout communities.

These security problems are extremely dangerous. Each one potentially gives an intruder the same access to a system's device as that of the executing JVM. Depending on the JVM, this level of access could include complete system control. There are several security measures that we have included within the DUPE framework which are used to counteract these vulnerabilities. These will now be discussed.

7.2.2 Security Measures

The security of DUPE is split into two sections: direct security and trust security. We acknowledge that trust is generally only seen as precautionary security, however, common everyday use of the internet indicates that it is seen by the user to be more than that. For example, it is a common occurrence to find a certificate attached to a secure web page or Java Applet. If the user wishes to execute such an application they must identify and accept the certificate as a security check. Therefore, the user understands that by accepting the certificate they may be giving the application greater access to their system than is normally allowed. If a certificate is not verified then the application will be denied access and its execution will be partially or totally restricted. In this sense, trust in the certificate provides a wall of security controlled at the user's discretion.

We have taken the precautionary step of adding a security specification to the DUPE framework. Many of the techniques used for security have proved to be applicable to its structure and its cooperation with other DUPE systems. However, the introspection aspects of DUPE has produced circumstances within systems that are abnormal for standard system configuration, consequently, some security techniques will prove impossible (or nearly impossible) for some framework

implementations.

We will now look at the Security Manager and its use of Policy restrictions that aid the security of DUPE and explain how they are used. We will also explain how full security control can limit a DUPE system's overall functionality.

7.2.3 Policy Restriction and Security

Java's security policies can be implemented to define a Security Manager in order to provide security for many distributed techniques. It is used to determine what can be achieved by an application. For example, `java.security.SecurityManager` can check the permissions of sensitive tasks before they are executed. If a permission is disallowed then a task can not be executed. Using it in this manner the Java Security Manager can implement a sandbox environment which will limit some operations [81]. Moreover, this technique provides a more flexible means of establishing sandbox security control. In a general distributed scenario, all security permissions are set as false during an applications execution unless otherwise specified. This technique is applied to Jini, RMI, Aglet and Applet applications [42].

The creation of a sandbox security using this technique can also determine the level of redefinition that a resource is able to achieve within a system. The capabilities of this technique are best seen through their use in Aglet security [62]. Aglets[1] are one implementation of the Mobile Agents that we discussed in Section 3.5. They are a Java specific technique which makes use of web access and the JVM. Aglets use Policy permissions to determine the level of access an Aglet has to the host system; for example, file access permissions. They are also used to determine the level of protection granted to an Aglet; for example, permissions to dispose or dispatch an Aglet.

7.2.4 DUPE Policy Control

As a result of DUPE's use of Jini the same security measures are applicable; in particular, policy specification permissions. The Policy setup within the DUPE

[1]It is not necessary for us to describe Aglets in detail. For full details see IBM Aglets [54] or Lange and Oshima [62].

framework uses Java Security Managers to control the permissions of a DUPE system during execution. There are several permissions that need to be allowed for needs of the DUPE middleware during execution. They include:

- read and write access to all configuration files
- access to the Jini multi-casting addresses
- socket access for all Jini ports, and
- class redefinition access (if appropriate).

However, a target application may require different policy details. Therefore, the DUPE specification (Appendix A) states that there must be two separate sets of Permissions active during a DUPE system execution:

- a Permission set governing the DUPE middleware, and
- a Permission set governing the target application.

The specification further states that, if possible, the Permission set for the target application should be split further into two, leaving the DUPE systems's execution with three Permission sets:

- a Permission set governing the DUPE middleware,
- a Permission set governing local classes of the target application, and
- a Permission set governing downloaded classes of the target application.

Within all of these Permission sets, the most obvious permissions, as listed by Sun Microsystems [101], which should be enabled cautiously within DUPE systems are[2]:

- `java.security.AllPermission`
- `java.security.SecurityPermission`
- `java.io.FilePermission`

[2]The noted types of permission are those seen at the time of writing to be the most dangerous for DUPE systems. However, other permission types may later be found to be of concern.

- `java.lang.reflect.ReflectPermission`
- `java.util.PropertyPermission`
- `java.lang.RuntimePermission`

Individually, most of these permission types will be controlled simply. Ideally DUPE would enable its permissions in a similar manner to Aglets. In this aspect, restricting common permissions, such as `java.io.FilePermission`, is standard for all distributed applications. Therefore, we see that restricting their use is not detrimental to the execution of an application. However, if it occurs, user preferences may be used to enable these permissions.

This is an important feature of any DUPE implementation, without it a target application may receive permissions that are not appropriate; for example, the target application may have access to all DUPE configuration files. The extent of the security of a framework implementation is a design feature, or a execution user preference. A simple example of the permission distribution for a DUPE System is provided in Table 7.1.

DUPE Middleware	`java.security.AllPermission`

Local Classes	`java.io.FilePermission ...`
	`java.lang.RuntimePermission ...`
	`java.net.SocketPermission ...`
	...

Downloaded Classes	`java.net.SocketPermission ...`
	`javax.net.ssl.SSLPermission ...`
	...

Table 7.1: Example DUPE System Permission Distribution

However, in a trusted environment the Permission sets may be set more freely; for example, it may be appropriate to set the DUPE middleware permissions as `java.security.AllPermission` in a safe work community.

7.2.5 Class Loader Redefinition Security

Further to the common security precautions set out by the DUPE specification, there is one unique problem which restricts the execution of a DUPE System. The problem lies in the definition of class loaders. Class loaders, subclasses of `java.lang.ClassLoader`, are the only objects within the JVM which are able to define new classes and associate new Java Permissions (`java.security.ProtectionDomain`) [42]. This means that, if security allows for class loader definition, a new, or redefined class loader, is able to go beneath the DUPE security structure and add additional permissions. These permission can be used by any subsequent class that the class loader defines. This enables them to open up the entire application to unwanted attacks and intrusion. Furthermore, the class loader can redefine permissions, to enable any permissions that should remain blocked during a DUPE system's execution; for example, the class loader may enable `java.security.AllPermission` for any downloaded class.

As a result of this problem, other, less obvious, permissions that ideally would be released for the DUPE framework, must remain restricted. These permissions are from the `java.lang.RuntimePermission`. The most detrimental from this group include:

- `RuntimePermission.createClassLoader`
- `RuntimePermission.setContextClassLoader`
- `RuntimePermission.setSecurityManager`
- `RuntimePermission.createSecurityManager`
- `RuntimePermission.exitVM`
- `RuntimePermission.getProtectionDomain`
- `RuntimePermission.accessDeclaredMembers`

This problem is not unique to DUPE. The ability to install a class loader in other distributed Java techniques, such as Jini and Aglets, would still cause problems. However, these techniques simply restrict class loader installation as it is not seen as a disadvantage to do so. However, we want DUPE implementations to manage *any* application not only applications adapted for mobile distribution. Consequently, disallowing the creation of a class loader is disadvantageous as it

would limit the number of applications that could be run. For example, a DUPE Community may exist in a home network where several larger devices, such as media centre or PC control, require advanced application capabilities (this scenario is further explored in Section 11.7). Class loader definition is therefore necessary for the DUPE middleware. Moreover, without this capability the framework is unable to define the DCLS and a DUPE execution will terminate. The separation of security policies with the framework enables this to occur. However, the problem will still remain for the target application.

There is a way around this problem. However, as far as our research indicates, it is only manageable within one JVM with dynamic updating capabilities: the J2SE 1.4 HotSpot JVM using the JPDA structure for dynamic updating [99].

The introspective mechanism of the HotSpot JVM allows an application to watch the entire proceedings of another application [99]. It is capable of doing so as the JPDA structure physically watches a separate VM. The effect is that it can trace the entire execution of a class line by line, this includes the execution of a class loader. Therefore, a JPDA implementation can specify a watch on all ClassLoader instances, and look for unwanted code such as `setSecurityManager(...)` calls to the SecurityManager. In fact it is possible to watch for all calls to the SecurityManager. This means that it is able to stop a class loader if it is attempting to enable security permissions that should remain restricted. Therefore, the DUPE security Permissions that restrict class loader definition can be replaced by the JPDA monitoring structure. This security configuration is similar to that applied to Safe-Tcl [16, 86]. Safe-Tcl uses a dual interpreter mechanism to execute untrusted code within a safe interpreter and execute trusted code using a master interpreter [86]. In this respect, the JPDA structure can be seen as a master JVM controlling the execution of a safe JVM, and therefore, the master JVM is able to determine if code is executed with full or restricted access.

Other dynamic JVMs, such as DVM and J2SE 5.0, generally execute the target application within the same VM as the DUPE structure. As a result, they are unable to control the instantiation of a class loader without restricting it altogether. The class loader definition Permission could be granted, but doing so would leave the DUPE system with a security flaw. Consequently, it should only be granted in a trusted community. This difference between the structure of a JPDA DUPE implementation and others is shown in Figure 7.2.

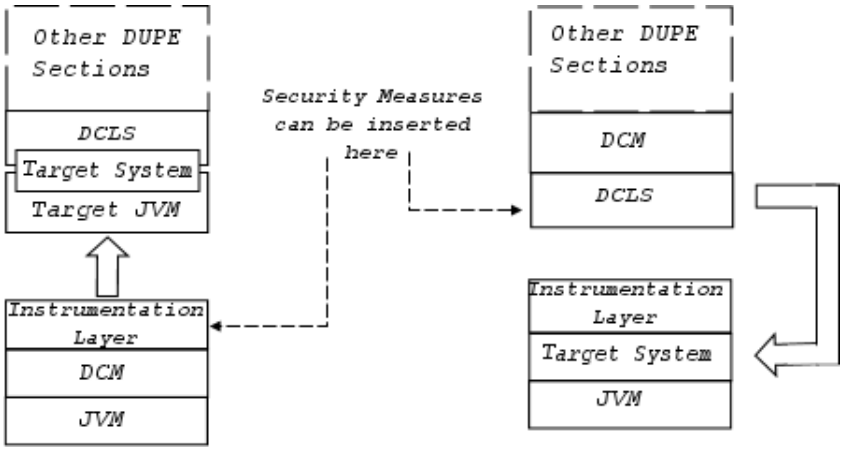


Figure 7.2: Dynamic JVM Implementation Separation

Other than using the security Permission to overcome this security problem, a DUPE implementation is able to simply disallow the loading and redefinition of class loaders within its structure. For example, in the redefinition and loading code of the DCM, prior to returning the class object, a simple inheritance type check may suffice. An example is shown in Figure 7.3.

```
public Class loadClass(String className){
     //locate class details as normal
     //define the class withih the virtual machine
     Class returnClass = this.defineClass( ... );
     //check class for ClassLoader instance
     Class superClass = returnClass;
      //check all superclasses for ClassLoader
     While((Type superClassType = superClass.getGenericSuperclass()) != void){
             if(superClassType instanceof Class){
                     Class superClass = (Class) superClassType;
                     if(superClass.name().equals(java.lang.ClassLoader){
                             return null; // or throw Exception
                     }
             }
             return returnClass; //passed all tests, return as normal
     }
}
```

Figure 7.3: An Algorithm for Controlling Class Loader Redefinition

However, as the exclusion of class loader redefinition from the DUPE mechanism presents a restriction to the target system, and, as we identified during the introduction in Chapter 1, one main task of this research is to give *any* application

access to DUPE communities, this area remains as a limitation of the DUPE framework.

In contradiction to all this negative point, the policy control of the DUPE framework is the best solution to the majority of security problems. Yet, as a result of the class loader policy problem, limitations will persist for most implementations of the framework, for now.

We have clearly identified in this section, that there are problems within the class loader monitoring and that class loader redefinitions allow the introduction of new, overwriting, policy rules. Moreover, as the newly loaded class loader can be used as the core loading of the new class (as long as the classes are easy to locate), it is impossible to determine if that class (within its code structure) will introduce new policy files. However, we maintain that it is possible to control class loaders and recommend that user selected options should be designed to allow them to indicate if they wish to exclude all class loaders from redefinition and loading, or if certificate checks (discussed in Section 7.2.7) must be performed on class loaders. This would allow the user to choose whom they trust to define a new class loader or redefine an old one. However, neither of these recommended features is included in the DUPE specification as it is not perceived that all implementations will be capable of such tasks. The framework design only includes the details of standard DUPE policy restrictions.

7.2.6 Recommended Changes to JVM Structure

We will now provide two solutions to the identified short comings of the JVM's class loader policy mechanism. Both of our recommendations relate to JVMs with dynamic updating capabilities in which class loader subclasses have the ability to introduce new Policy details.

1. **Disallow Class Loader Policy Redefinition:** remove the ability for subclasses of the `java.lang.ClassLoader` class to define new Policy details and allow the introduction of a new Policy only via the bootstrap class loader. This is extreme, and although it would solve the problem it is likely to restrict many applications; for example it will restrict the RMIClassLoader in RMI applications [105].

2. **New Class Loader Policy Redefinition Policy:** define a new Java Policy that determines if a class loader is allowed to define new policy details. We believe that this solution is the most appropriate as it allows applications to define new policy details if required, but also to disallow them. However, the concern with this solution is the class loader hierarchy; for example, decisions such as, whether or not a new policy level is based on that of the instantiating class loader. It may be that these settings are specific to the application, or in fact the JVM. However, we do suggest that, across the board, the base class loader should determine all initial settings of policy redefinition. Therefore, any subsequent restrictions may be a case by case scenario.

These solutions are only what we see as possible ways to end the problem; other solutions may exist. However, no matter what the final solutions are, we maintain that with simple alterations to the class loading capabilities of a JVM, the safety of DUPE community interaction would be significantly improved. This is identified as future work in Section 13.1.

7.2.7 Digital Certificates

Certificates provide a simple means of trust to many internet applications. It is common to verify a certificate prior to using secure sites on the internet, using Java Applets, or when installing new applications on a systems. The last of these activities is, to the larger extent, what is achieved by DUPE, and therefore, we reason that certificates can, and should, be used as a means of identification and trust for all DUPE resources.

A common practice for distributed applications is that each jar file is signed by one or more certificates [109]. This allows for the verification of class details by using the Public Key Infrastructure (PKI). In this technique, a digital certificate is a binding of a public cryptographic key to an identity. The public key is found within the certificate and is used to verify that the signature associated with a class was created using the private key of the sending party (the identity). The key is also then used to verify that the class details remain unchanged from the time of the signature's creation and association. This process can be repeated using multiple certificates in a certificate chain. When using a certificate chain, not all certificates are used to verify class bytes. Instead, other than the initial certificate which is associated with the class, they validate another certificate from the chain. Therefore, as each certificate in the chain is verifiable, it is possible to

verify back through the chain, from the initial certificate, until a trusted authority is reached. This trusted certificate can be seen to verify all the certificates in the chain as trusted, including the one used to sign the class. For simplicity, we will refer to this PKI technique as PKI verification for the remainder of the section.

The PKI verification should be applied to any implementation of the DUPE framework. The certificates are gathered within the DUPE structure as `java.security.cert.Certificate` objects. These objects, as previously defined in Section 5.6.3, then become part of the DynamicClassEntry instances within the DSO and eventually part the DynamicClass object representation of a system resource. Furthermore, as the main purpose of a DSO is as a remote class representation within a DUPE community, each class should be individually signed.

The verification of DSOs and their contents is a two stage multiple-step process. The initial stage occurs the first time any DUPE system gathers the class details, that is, the first time a class is obtained from the Jar file and not via a DUPE resource. The steps involved in this process are:

1. Locate appropriate Jar file.
2. Analyse certificates associated with all classes within the Jar file and, if appropriate, those associated with the entire Jar file. If there are no certificates for a class user preferences will determine the action taken by the system. The default action of all DUPE systems should be to treat the class as hostile and not to use it.
3. Determine if the certificates, more specifically the signing certificate from the chain, are appropriate for this DUPE execution. The certificates appropriateness may be determined by either questioning the user or, preferably, preset settings. (Steps 2 and 3 are used to speed up the adaptation process by immediately eliminating classes signed, or not signed, by specified identities).
4. Verify the class bytes using PKI verification.
5. Load the associated class bytes.
6. Create the DynamicClass within DUPE associating the certificates, attributes and class bytes with each other.

7. Advertise a newly cerated DynamicClassEntry within the systems associated DSO, including the certificate information for quick community processing.

As a result of DUPE community interaction, once the first stage has been completed, other systems are able to gather the resource and its associated certificate(s). The second stage steps are:

1. Discover the appropriate DSO object as a result of community events or internal requirements.
2. Search the DSO for appropriate DynamicClassEntry (appropriateness is initially determined via class name and adaptation requirements).
3. Analyse associated certificates in the same manner as in stage 1 steps 2 and 3. (As with stage 1 steps 2 and 3, here step 3 is provided to speed up the system adaptation process).
4. Once a DynamicClassEntry is found to be both appropriate and trusted according to certificate checks, the full resource is downloaded.
5. Verify the DyanmicClass details received from the resource, including PKI verification of the class bytes.
6. Use the class bytes to either define a new class or redefine a current class.

These two stages give a DUPE community a a level of trust; including a preliminary check and a discovery check. Moreover, when gathered remotely, all resources are checked twice, both for initial trust prior to class gathering, and for final verification before any resource is included within the systems structure.

However the use of certificates remains an issue and research area in itself. Specifically within the DUPE framework, there is a requirement to trust the initial signer of the remote class. That is, there is a requirement for the identities within the certificate chain to be already known to the system, or trusted by the system. In DUPE this requirement can be bypassed by setting the security setting to localised settings, such as those required for local adaptation (an example of such adaptation is provided using a tourist scenario in Section 11.8). Unfortunately, there remains a possibility of unsolicited code entering the system in such an environment. The extent to which a user is deterred by this possibility will determine the level of security applied during a single execution of a DUPE system.

Nevertheless, some issues concerning DUPE's trust and security, in particular those related to the use of certificates, are seen as future work related to this research (Section 13.1).

7.3 Adaptation Controls

Adaptation controls enable DUPE systems to adapt according to the version of a resource that is most appropriate and available within reach of a system. This feature is helpful when DUPE is used for the removal of errors, the addition of extra features to a system, or the running of community specific algorithms. Furthermore, as the evolution of systems is an internal linking process, a change in the version of one resource can lead to the need for a new resource, or the update of another. A single adaptation process may initialise a system to adapt, or evolve, based on how many more resources are found within the community. The location of the adaptation control within the DUPE model is shown in Figure 7.4.

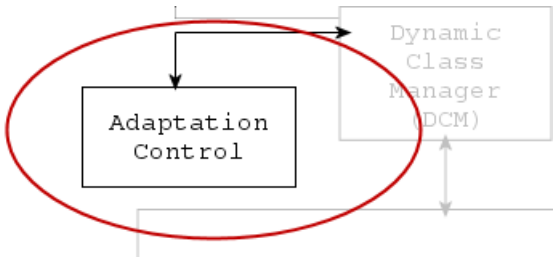


Figure 7.4: Adaptation Control within the DUPE Architecture.

Version manipulation within Java systems is generally achieved through use of a jar file version attribute, as used, for example, in Web Start [109]. DUPE is based on the discovery and use of other systems' resources and consequently class discovery is not necessarily confined to jar files. However, for simplicity and adaptation control, DUPE requires all files to be in jars. This allows the version attribute of a class to be assigned during any type of component gathering, including:

- **Loading via classpath:** as the application is initially contained in a jar file as each class is obtained, so too are its associated attributes. DUPE is, therefore, able to easily gather all required information and store it as part of the DynamicClass and DynamicClassEntry.

- **Loading via JVM:** at times it may be best to load a component directly from the source system's JVM. This system has obviously loaded the component at some stage itself, storing and linking the attribute details.
- **Personalised loading:** as a target system has the ability to load classes in its own unique manner, the framework specifies that all systems provide attribute details for each class. Conceivably, personalised loading techniques, for example URL based class loading or internal search class loading, make use of jar files. It is, therefore, reasonable to assume that they also have access to the manifest details.
- **DUPE class loading:** the information obtained when gathering a class within a DUPE community contains a copy of its original attributes. Therefore, the attributes can still be analysed and stored within the structure of the downloading DUPE system.

During the execution of a target system, the continual upkeep and referencing of the resource attributes is essential to system adaptation. Moreover, this is also an essential component in maintaining the security aspects of DUPE systems.

7.3.1 Adaptation Reasoning

In order for a DUPE system to control its adaptation, all DynamicClassEntry objects contain details that differentiate them from other versions of the same resource. A system can use this information to carry out an analysis using its own adaptation rules to determine if it should use a resource.

There are many different reasons for change; for example, a system may wish to evolve in accordance with the version that is commonly used within a specific community, location based, adaptation. In this example, a system would proceed with adaptation irrespective of any version identifier. Therefore, system adaptation relies on the particular settings of the resource and its own adaptation settings. Moreover, each DUPE system should have its own adaptation settings specifically selected according to the wants of a user.

We will now describe the adaptation process beginning with resource identification and moving to the use of the identification.

7.3.2 Setting Resource Identification

Class identification can be set by using the attributes of a manifest file that is associated with a jar file [95]. Using multiple class names within the manifest allows it to detail each specific class that exists in the Jar file. These details can then describe the entire package.

A manifest file provides a means of identification to all the files associated with a jar file. The manifest is used to describe the version and other specific details of each class, such as the vendor of an entire package. DUPE makes use of this feature to determine the adaptation identification of all community resources. It uses multiple names within the manifest to describe the version of each specific class. An example of the manifest file, `disney.mf`, which can be used within a DUPE enabled system is presented in Figure 7.5[3].

```
Manifest-Version: 1.0
Name: duck
Specification-Title: Ducks
Specification-Version: 2.12
Specification-Vendor: A.Ryan
Implementation-Title: Duck1Melb
Implementation-Version: 2.12.78
Implementation-Vendor: Ryan

Name: duck.Daisy
Specification-Title: Ducks
Specification-Version: 2.13
Specification-Vendor: A.Ryan
Implementation-Title: Duck1Melb
Implementation-Version: 2.12.1
Implementation-Vendor: Ryan

Name: duck.Donald
Specification-Title: Ducks
Specification-Version: 2.13
Specification-Vendor: A.Ryan
Implementation-Title: Duck1Melb
Implementation-Version: 2.12.23
Implementation-Vendor: Ryan

Name: mouse.Mickey
Specification-Title: Mice
Specification-Version: 4.11
Specification-Vendor: A.Ryan
Implementation-Title: Mouse1CA
Implementation-Version: 13.12.18
Implementation-Vendor: Ryan
```

Figure 7.5: `disney.mf`

Within the manifest details, a general adaptation identification set will apply to all classes *except* those signed with specific adaptation details. For example, if `Disney.jar` contains, amongst others, `duck.Donald` and an additional four Java class files: `duck.Huey`, `duck.Dewey` and `duck.Louie`, and the attributes of each class is determined by the manifest file `disney.mf` (Figure 7.5) the adaptation details of each separate class will be different. For example, all the files will

[3]A manifest file should be written vertically, however, for the purpose of simplicity, Figure 7.5 does not comply with this requirement.

have the Specification-Version 2.12 as determined by the `duck` package identification, except for `duck.Donald`, which has its own Specification-Version of 2.13. The same holds for all other identification values. This allows DUPE systems to determine the details of each and every class, even if one or more are not individually identified.

7.3.3 Finding Resource Identification

We have iterated, constantly, that the information for all exportable DUPE resources on a host system is contained within an associated DSO, and, that these resource details are in the form of a collection of DynamicClassEntry objects. In Section 5.6.3 we discussed how the DynamicClassEntry object is a direct representation of a class available as a resource on the host DUPE member (the contents of the DynamicClassEntry were given in Table 5.2).

Within the DynamicClassEntry there exists the `attributes` variable of type `java.lang.String[][]` (see Table 5.2 for details). This variable stores the attributes that were assigned to a class by its initial manifest file. Each array element consists of two values: `attributes[i][0]` element is the attribute identifier and `attributes[i][1]` element is the value assigned to the attribute. Both variables are stored as a String representation of the actual value. For example, the attirbute value `Specification-Title:Ducks` is stored as:

```
attributes[i][0] = "Specification-Title"
attributes[i][1] = "Ducks"
```

As already mentioned, DUPE system adaptation is not constrained to simple version upgrades. We further this to specifically point out that, situations may occur where an older or more specific version of a class is more applicable. For this reason, the user of a system is given some control over their adaptation settings. This allows them to specifically dictate how their DUPE system adapts. To accommodate this requirement, all attributes are represented as a two dimensional String array element. This enables DUPE systems to create different types of adaptation attributes. For example, a company may create their own attributes and assign them to resources. If a system understands, or is able to notify a user of these new attributes, it can adapt correctly. Otherwise, a system that is unable to flexibly add attributes to its attributes list would ignore the attribute and adapt according to those it knew.

The adaptation attributes are based on the attributes found within a manifest file. The standard list of manifest attributes are shown in Table 7.2.

Specification Details	**Implementation Details**
Specification Version	Implementation Version
Specification Title	Implementation Title
Specification Vendor	Implementation Vendor

Table 7.2: Standard Attributes in a Manifest File

All attributes are analysed, separately. This allows each to be analysed at individual stages and according to individual rules.

The flexibility of adding attributes to a knowledge list of a DUPE system would be part of its specification. Therefore, users who desire this ability can use a particular implementation whilst others may choose a simpler one. For example, an embedded system is likely to be strict on its attribute understanding and only adapt according to the attributes it originally knew about; however, an advanced mobile device such as a PDA could be more flexible and provide the user with the option to select new attributes to include as part of their structure, and determine the rules for their adaptation.

7.3.4 Using Resource Identification

The attributes supplied by a manifest file must initially be gathered by a DUPE system via the original Jar file. This stage is only ever carried out by the system which will gather the class for the first time either loading the class during standard execution or accessing it from the file system as a result of a DUPE request[4]. The manifest details can be obtained by using the `java.util.jar.Attributes` class and its static nested class `java.util.jar.Attributes.Name` [102].

The DUPE system will store all attributes of each class as part of a representing DynamicClass. How the attributes are saved within the framework is specific to each implementation. For example, embedded systems may find it beneficial to store them as a `java.lang.String[][]` variable for storage reasons. More complex systems may use the added benefits given by the `java.util.jar.Attributes`

[4]As a system can load classes via specialised class loaders, it is assumed that all classes are gathered from a Jar file and that the manifest details of the Jar are available to the system.

class. No matter how a system stores the information, it should only be included in the DynamicClassEntry and the DynamicClass that is passed throughout the community as a two dimensional String array variable (according to the previous section).

Once the attributes are obtained as a result of a community search, within a DynamicClassEntry, they should be used according to individual adaptation rules. Although a system maintains its own rules for adaptation the process of adaptation reasoning should make sure that all rules are adhered to. The incorporation of a simple algorithm that flows through all attributes and assesses each is recommended. An example is provided in Figure 7.6.

```
begin checkNewClassVersion (String[][] newAttributes, String[][] currentAttributes)
  int checkedCount /*how many attributes were set to be chencked*/
  int passedCount /*how many attributes checked and passed*/

  /*loop through and check all the attributes*/
  for each currentAttribute
        /*if using attribute for adaptation check*/
        if checking currentAttribute
               checkedCount++

               for each newAttribute
                      if newAttribute same type as currentAttribute
                            /*adaptation check*/
                            if newAttribute better then currentAttribute
                                   passedCount++
                            end if
                            break
                      end if
               end for

        end if
  end for
  if checkedCount = passedCount /*passed all adaptation checks*/
        adapt to new resource version
  else
        do not adapt and ignore new resource version
  end if
end checkNewClassVersion
```

Figure 7.6: Example Adaptation Algorithm

7.4 The Limitations of DUPE Safety

As a result of the detailed description of DUPE's security and adaptation controls, there are several restrictions on a target applications.

- All target applications must have been loaded from within a Jar file at some stage. This Jar file must have an associated Manifest file that details the version status of all class files. Without this detail, the DUPE adaptation components cannot be set.

- Some DUPE implementations may limiting class loader redefinition; as discussed in Section 7.2.3.

7.4.1 Constructing Target Application

As highlighted in the earlier chapters of the thesis, there should be no constraints placed on a target application of the framework. (Although, we do note some small restrictive observations during our implementation testing (see Chapter 11)). However, for security and adaptation purposes, there are several compilation and distribution elements that must be applied to all target applications participating as DUPE members.

All DUPE framework implementations require several aspects of a target application to be correct. These are:

1. The application must be compiled using `javac` with the `-g` option. This option is necessary for class updating within some JVM's, for example J2SE 5.0. It will not, however, affect the execution of the system in any way within other JVM's.

2. The application must be contained in a signed Jar file and stored in the trusted section of the DUPE file structure. This section of the DUPE file structure is generally the `dupe/trusted` folder; however, this may vary according to implementation features.

3. The application's associated Jar file must have an associated Manifest file containing all adaptation information.

4. The Jar file must be signed with a minimum of one certificate.

These requirements are necessary for a target system. We believe that they are not unreasonable requests. It is standard procedure to have some slight variations on the compilation and execution of distributed systems. Therefore, all these requirements should not pose a problem to any system.

7.5 Summary

This chapter has addressed the aspects of the DUPE framework that create concern: security and adaptation reasoning. By clearly specifying security requirements, we have attempted to resolve the more precarious aspects of the DUPE framework.

One interesting point that is discernible from our discussion on DUPE's security and reasoning, is that DUPE is susceptible to many of the same problems that are generally associated with internet and email applications. We argue that it is possible to control the means by which a system will adapt to a community and, therefore, it will maintain a secure environment via trust and security barriers. At the same time, we also point out that policy control, a seemingly large defense against intrusion, can only be applied to particular implementations of the framework without restricting the use of DUPE. This is not to say that the security of the DUPE system is highly flawed. We only indicate that without extensions to the JVM specifications, we can not provide DUPE systems with unquestionable security and protection from unwarranted attacks. However, this is no different to most other system cooperation methodologies.

This chapter is the final chapter of the design sections of our work. The remainder of the thesis focuses on framework implementations, their evaluation and modeling the movements of gypsy agents.

Chapter 8

DUPE Lite Implementation

8.1 Introduction

The next three chapters will provide the technical and reference details for three implementations of the DUPE framework. These are:

- **DUPE Lite:** a limited implementation with no redefinition capabilities designed for standard JVMs.
- **DUPE 5.0:** a complete implementation using J2SE 5.0 JVM.
- **DUPE JPDA:** a complete implementation using J2SE 1.4 HotSpot JVM[1].

Although, the different versions of DUPE were programmed separately, we have, mostly, used the same package structure. For example, as specified by the framework design, all versions use Jini (v1.2). Furthermore, as we have designed the implementations to be as similar as possible, it enables us to more accurately assess the differences in the finer details of the structures. Using the same structure for all implementations, enables our testing to be more accurate. However, even though there are several similarities within their code structures, the differences in the coding techniques for each particular JVM, significantly determine how each may be used. For example, only DUPE JPDA and DUPE 5.0 are able to dynamically alter the state of a targeted system during runtime. DUPE Lite is

[1]The HotSpot JVM is also available within J2SE 5.0, however, J2SE 5.0 provides other means of dynamic class updating and instrumentation and consequently J2SE 1.4 is used for this framework implementation.

limited to load-time adaptation.

As mentioned, there are aspects of all implementations that are the same. These parts of the implementations will not be repeated from chapter to chapter. We will identify, in each chapter, the sections of the framework, according to the framework design in Chapters 5 and 6, and the security and adaptation discussed in Chapter 7, which are specific for the implementation. All other sections of the framework carry on from preceding implementation chapters. For each implementation, initially, where appropriate (Chapters 9 and 10), we provide a detailed technical analysis of the corresponding JVM[2]. The details of the DUPE implementation then follow.

We begin in this chapter with DUPE Lite.

8.2 General Features of DUPE Lite

DUPE Lite is a limited implementation of the DUPE framework. As such, it is not able to fully achieve all the features described in the framework design chapters, in particular, runtime dynamic adaptation. The other two core aspects of the DUPE framework, Distributed Connectivity and Community Cooperation, are implemented by DUPE Lite; however, DUPE Lite is only able to adapt to a community using a stop-start technique. This is a direct result of our choice in JVM. However, DUPE Lite advertises all its resources in a DUPE community and therefore, from a language cooperation aspect, is DUPE compatible. Therefore, according to our DUPE compatible requirements from Section 6.5, it remains a limited DUPE middleware.

Although, these limitations restrict its adaptation, they do not restrict any other section of the framework. All framework implementations will be most restricted by their choice of JVM as this is essentially the most individual section of any implementation. The limitations of the DUPE Lite are present because we chose to use a standard JVM, moreover, DUPE Lite shows that, even if a DUPE middleware can not dynamically update classes during runtime, it may still communicate within a DUPE community. The exploration of the possibilities that arise from this concept are identified as future works (Section 13.1).

[2]General discussions on Virtual Machines are provided in Chapter 2

DUPE Lite maintains all DUPE community discovery specifications, as discussed in Chapters 5 & 6, by implementing all relevant interfaces, and constructing the DynamicClassEntry class and all requirements of the DSO. Table 8.1 provides the package list for DUPE Lite. The table identifies the name of each package and details its purpose.

Package	**Details**
`dupe`	Base package of DUPE contains startup file.
`dupe.common`	Holds the common classes that are used in other packages.
`dupe.config`	Configuration setup classes. Acts as a separate application.
`dupe.discovery`	Community Management (DCLS).
`dupe.security`	Security Control.
`dupe.versioning`	Adaptation Reasoning.

Table 8.1: Package Structure for DUPE Lite

The `dupe.common` package provides the most common classes that do not belong in a specific sector. At this stage, this package, within all DUPE implementations, includes only the `dupe.common.DynamicClass` class. DynamicClass is the direct implementation of the DynamicClass of the framework. It contains the basic information required for safe adaptation of classes and only includes the variables that were specified in Table 5.4.

8.3 Implementation Construction

We will now discuss the technical aspect of this implementation according to the DUPE framework design listed in Section 5.1. The structure of this implementation is shown in Figure 8.1.

DUPE Lite maintains, as closely as possible, the exact DUPE framework design. However, some sections of the design are affected by limitations of the implementation. These are shown in red in Figure 8.1. All other requirements of a DUPE compatible middleware remain, making intercommunication with other DUPE systems, possible.

During the discussion on the DUPE Lite implementation, and in fact all of the implementations, we will identify only the key classes. Although, there may be

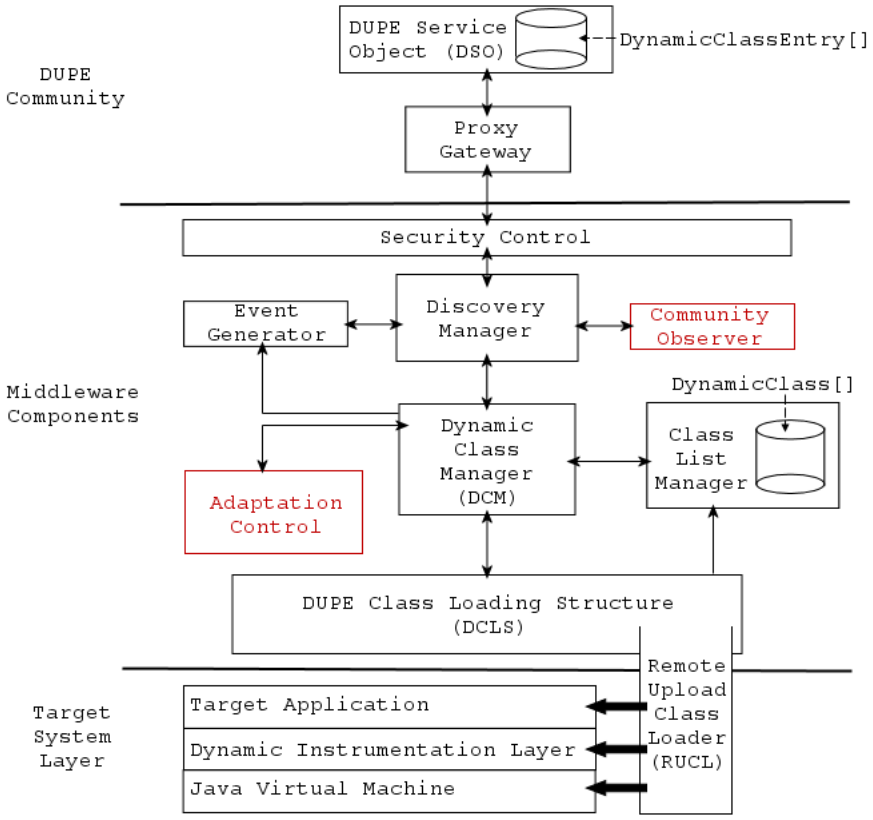


Figure 8.1: Structure of DUPE Lite

instances during the discussion where a class also requires the analysis of another smaller class that has not been identified. The sections are written so as to provide the necessary information without confusing the reader's image of the implementation structure.

8.3.1 Distributed System Communication and Code Sharing

The core elements of the Dupe Class Loading Structure (DCLS) are implemented within the `dupe.discovery` package. This section of the framework contains all classes used for DUPE community interaction. This package is developed in the same manner within all of our DUPE implementations. Many classes within the

package are a result of Jini requirements. We will now discuss each of the core classes of this package. These details remain the same for all implementation chapters.

DUPE Class Loading Structure (DCLS)

The most important class of this section, and throughout the entire implementation, is `dupe.discovery.RemoteUploadClassLoader`. RemoteUploadClassLoader (RUCL) implements the essential interfaces for DUPE connectivity, `dupe.discovery.RemoteClassLoader` and `dupe.discovery.RemoteClassLocator`, as well as being a subclass of the ClassLoader class. RUCL is DUPE's main class loader and provides all other sections with the information relating to the classes of the target application. The RUCL controls the instantiation of the classes; it also indicates to the Dynamic Class Manager when a class cannot be located locally and triggers a community search. This is only achievable if it has a direct connection with the bootstrap class loader of the target application.

DUPE Service Object (DSO)

The DUPE Service Object (DSO) is designed exactly as specified by the framework. It is established using the LoaderType and DynamicClassEntry classes located within the `dupe.discovery` package. The complete details of the DSO were provided in Section 5.6.3. (These details are also in the DUPE Specification, see AppendixA.).

Proxy Gateway

The proxy gateway design allows a searching system to access the full resource details of other DUPE members. When a resource is wanted by a DUPE member, it will make use of the known RemoteClassLocator interface available via the DSO. This interface is also implemented by the RUCL so that a searching system can call specific methods via the proxy to gather the details of any class. Specifically, the interface method `generateDynamicClass(...)`, as was detailed in Section 5.6.3, is executed.

The proxy gateway is supplied to the DUPE system via a dual level class mechanism which joins the DSO to the rest of the structure. Two classes provide this mechanism, `dupe.discovery.RemoteClassLoaderGateway` and `dupe.discovery.RemoteClassLocatorProxy`. As the searching system must make sure that it is not calling any remote methods on itself, this technique is essential.

Target Application

The target application is governed via the DCLS. This allows the target application to utilise all the elements of the DUPE middleware, and for the middleware to access all the required information of the target application. The initialisation procedure of the target application is as follows:

1. An application is started within DUPE Lite using a command such as:

   ```
   java dupe.Dupe -Djava.security.policy=d:\dupeLite.policy
      -Djava.rmi.server.codebase=http://192.168.0.100:8080/
      coffeeMachine.CoffeeMachineGUI 192.168.0.100
   ```

   where the line follows the procedure:

   ```
   java dupe.Dupe -security_options -jini_options
      target_application arguments_for_target
   ```

2. DUPE Lite will initialise the target application within its own structure. Consequently, all settings applied to the target application will be available for the target application but contained separately from the settings applicable to the middleware. This is in accordance with the security specifications in Section 7.2.3.

3. The target application is then initiated, via reflection, using the RUCL as its bootstrap class loader.

4. During the progression of the target application, all classes are loaded by the RUCL. This is then able to provide the rest of the structure with all necessary class information.

8.3.2 Dynamic Systems Alteration

This section of the DUPE Lite implementation is restricted. DUPE Lite does not allow for runtime dynamic class updates, and as a result of this, it can not achieve runtime system adaptation. This, is a direct result of JVM choice of a standard J2SE 1.4 JVM.

However, DUPE Lite is able to adapt to a community at load-time. This means, that if an application is started within the proximity of a DUPE Community, it is able to search through the community resources for appropriate classes. Therefore, some constructs of the Class List Manager remain, and it continues to use DynamicClass objects for resource purposes. This will now be discussed.

DynamicClass

The DynamicClass is structured exactly as defined in Section 5.7.3. For all our implementations, we use the class `dupe.versioning.SerializableAttributes` to hold class attributes[3]. The complete details of the DynamicClass were discussed in Section 5.7.3. According to these details, when transferred among DUPE systems, the SerializableAttributes variable of the DynamicClass details is altered to a `java.lang.String[][]` variable.

Class List Manager

The Class List Manager (CLM) implementation uses a `java.util.HashMap` for the storage of all DynamicClass objects (this remains for all our DUPE implementations). The CLM provides very restricted access to this collection and maintains all references to each class as required. In the DUPE Lite implementation, the CLM is far less complicated than in our other implementations. There is no need for it to redefine the details of DynamicClass objects as they are never altered during runtime. Therefore, the CLM's initial version of each DynamicClass remains throughout execution of the system.

[3]The SerializableAttributes class was originally designed to allow DUPE Lite to be implemented as a distributed updating technique. However, this concept proved to be outside the scope of this thesis and has been identified as future work. Nevertheless, we continue to use the SerializableAttributes class as it is not detrimental to the DUPE community cooperation aspects of the implementation and may be used for future extension purposes.

8.3.3 Community Interaction and Cooperation

The interaction of a DUPE system in the community is established using the event mechanisms provided by the Jini service object (the DSO). According to the DUPE compatibility requirements (Section 6.5), this section of the framework is specific and must be the same for all DUPE implementations. As a result of this, all implementations include the required aspects of the DSO and its associated classes; however, here, we will not reiterate all associated constructs in full. For the complete details see Chapter 6 and Appendix A.

Discovery Manager

The Discovery Manager is the Jini hub of the implementation. It is designed to constantly discover communities and register the DSO within all communities. As the framework specifies, the Discovery Manager establishes a Community Observer for each interacting DUPE community and provides access to all communities.

Community Observer

The Community Observer watches the DUPE community by listening for DUPE community events. It is able to check all resource information prior to accessing it for any class redefinition. And, as a result, all class details are synchronized in the appropriate places so as to maintain the internal safety of the implementation during execution. In the DUPE Lite implementation the community observer only analyses community events during startup, as runtime events are useless to the middleware. However, DUPE JPDA and DUPE 5.0 continue to listen for community events throughout their entire execution. All the DUPE implementations are capable of permanently saving discovered class details for future application executions; user preferences can be set to control this feature.

Event Generator

All community events are created via the Event Generator. The Event Generator uses the Discovery Manager to aid this task. The loading of a new class, either from internal class details or externally gathered details, (the calling of `loadClass`) will, once completed, trigger the event to update all advertised DSO's, which will in turn create community events.

The details to update the DSO are provided to the Event Generator in the form of a DynamicClassEntry from the DCM. Consequently, there is minimal computation achieved within the Event Generator to create the actual event. This is designed as such to lower the computation timing that is required for this task.

If possible, once an event has been generated, a system should remain connected to the community to enable other members to gather the new resource details. However, once one other member gathers the resource, that creates another place from which the resource can be gathered, and so forth. This chain reaction will produce many events; however, as detailed by the framework in Chapter 7, they will all be ignored by systems that have already loaded the resource requirements; they will also be ignored by all members not wanting the new resource details.

8.3.4 Security Control

Before any reloading of a class, the security settings of the implementation are verified. All the security aspects of the framework implementation are within the `dupe.security` package, and have been implemented according to the security specifications given in Chapter 7.

The security specifications consists of verifying signatures and the checking of security certificate chains. The security mechanisms are constantly checked throughout all operations of community cooperation of the DUPE system, particularly by the Community Observer (via the DCM) as it is the first point of entry for all resources. Configuring the appropriate settings for the security of the DUPE structure is explained in Section 8.4. Most of the security features of DUPE Lite are not changeable. You cannot turn off certificate checks as these are not only beneficial to a single system, they are also essential for the maintenance of trust throughout a DUPE community. The definition of a class loader, using downloaded code, is the only aspect of security that may be altered by a user. Allowing this to occur is only a risk to the user's system, and will not affect the execution of other community members.

8.3.5 Adaptation Control

The adaptation section of the implementation is the most accessible to a user. All other sections of DUPE Lite are configured for specific purposes and are not governed by a user in any way, except for the alteration of some security measures

(although this should always be done cautiously). As we discussed in Section 7.3, there are many different reasons for system adaptation, and these reasons should be based on the specifications of the manifest file class identifications.

The package `dupe.versioning` contains all the required classes to allow the framework to govern its adaptation, and the setting used within this package can be reached through use of the configuration interface (Section 8.4). The initial adaptation settings can be applied as the main identification attributes within the manifest file. These are presented in Table 8.2.

Specification	**Setting Details**
Specification Version	Can be set to allow only evolutionary change. For example, a class with the Specification Version number 23.01 will only adapt to a resource with a version number greater than 23.01.
Specification Title	Is used to control and specify that a new resource must be from exactly the same Specification Title. This is set to 'true' for situations where manufacture title identification is necessary, and should be 'false' to maintain a more flexible adaptation such as adapting according to location resources.
Specification Vendor	This should be used in a similar manner to the Specification Title, however; it creates even more restrictions by maintaining the vendor of the original class. A false setting is recommended within a trusted community as it provides a system with a wider range of resources to use for adaptation.
Implementation Version	This is used in the same manner as the Specification Version except it is applied to Implementation details.
Implementation Title	This is used in the same manner as the Specification Title except it is applied to Implementation details.
Implementation Vendor	This is used in the same manner as the Specification Vendor except it is applied to Implementation details.

Table 8.2: Adaptation Settings for DUPE Lite

8.4 Configuring DUPE Lite

Two different techniques may be used to alter the configuration of DUPE Lite. These techniques are:

1. using the supplied GUI interface, or
2. altering the DUPE configuration files within DUPE's bin directory.

Essentially, both methods will result in altering the DUPE configuration files within the `dupe/bin/` directory.

The configuration files (in the form of `name.cfg`) detail the information that DUPE Lite uses to control its adaptation. For example, the adaption control file `dupeVersioning.cfg` (shown in Figure 8.2) controls the adaptation settings for any DUPE execution.

```
#Wed Sept 10 15:42:01 EST 2005
SPECIFICATION_VERSION_CHECK=true
SPECIFICATION_VENDOR_CHECK=false
SPECIFICATION_TITLE_CHECK=false
IMPLEMENTATION_VERSION_CHECK=true
IMPLEMENTATION_VENDOR_CHECK=false
IMPLEMENTATION_TITLE_CHECK=false
```

Figure 8.2: `dupeVersioning.cfg`

The details within the files are stored in the format that is produced by the `java.util.Properties` class, as this is used to manipulate Attribute objects in a Java application. Using this technique results in a simplification of configuration and also provides a format for the configuration files.

8.4.1 Configuration GUI

The configuration GUI is a simple window implementation that allows a user to make minimal alterations to the constructs of DUPE Lite. It is expected that a user may wish to manipulate the adaptation settings for a system, however, and as previously noted, there is little need for a user to become concerned

with the security aspects. As a consequence, these settings will generally remain consistent. Figure 8.3 shows a screen shot of the GUI application running within a Windows based Desktop.

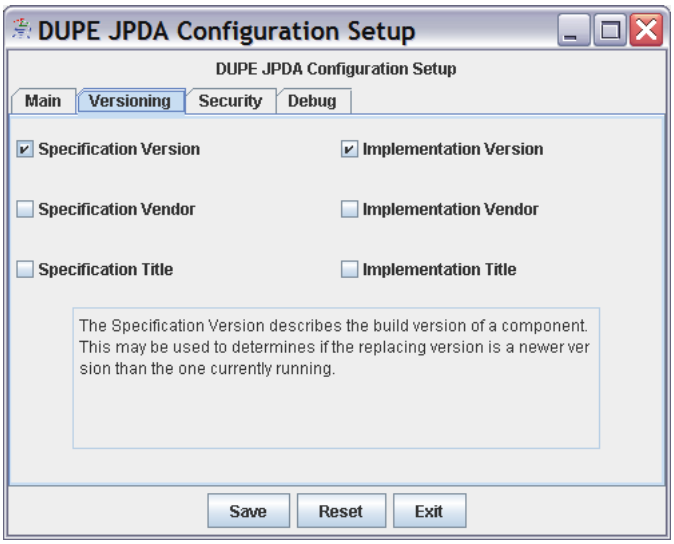


Figure 8.3: DUPE Lite Configuration Setup GUI

8.5 Limitations

There is one main limitation of the DUPE Lite implementation; it is unable to dynamically alter classes at runtime, and therefore cannot adapt to communities during runtime. We have already discussed this limitation and given reasons for the implementation of DUPE Lite earlier in the chapter. And, as will be shown in the next two chapters, DUPE Lite's adaptation features, although load-time only, are, generally, as flexible as our other implementations.

8.6 Summary

DUPE Lite is the simplest of DUPE implementations. It is designed for standard J2SE JVMs and consequently is unable to dynamically alter state during runtime. However, it will allow a system to cooperate in a DUPE community, albeit at a limited level, and provide other DUPE systems with its available resources.

The next two chapters describe fully compatible DUPE implementations. Both implementations are designed using the framework specification; therefore, these next two chapters have several areas in common with this chapter. These areas will be identified, however, their details will not be reiterated.

Chapter 9

DUPE 5.0 Implementation

9.1 Introduction

This chapter details DUPE 5.0, the first of our fully compatible DUPE implementations. DUPE 5.0 makes use of the newest Java Virtual Machine technology, J2SE 5.0. We based our choice of J2SE 5.0 on our strengthening belief that this Virtual Machine implementation is the most likely of the future JVM's that include dynamic runtime class updates to be commonly distributed among mobile devices.

The elements that alter DUPE 5.0's design structure significantly from that described in the previous chapter are all due to the addition of the class redefinition techniques, and the associated algorithms. Moreover, as a result of this, the exact DUPE framework design could be applied.

This chapter is structured the same way as the previous chapter. However, the sections of the implementation which directly relate to sections of DUPE Lite will not be repeated. This chapter concentrates on the aspects of DUPE 5.0 that make it an individual implementation of the framework. The areas of the framework covered are shown in green in Figure 9.1.

To begin our discussion of DUPE 5.0 we now provide a detailed analysis of the J2SE 5.0 JVM.

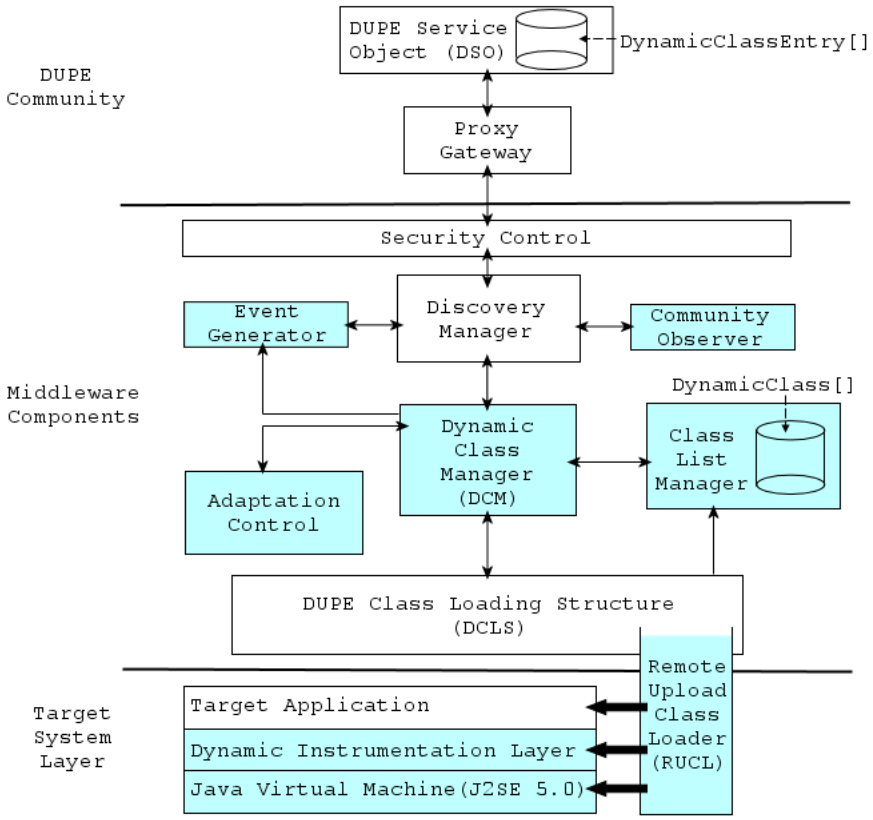


Figure 9.1: Structure of DUPE 5.0

9.2 Java 2 Platform Standard Edition 5.0

The Java 2 Platform Standard Edition (J2SE) 5.0 is the most recent, major revision of the Java platform and language specification. This edition is a significant upgrade from the previous edition and includes many new features. These include, for example, type generics, Metadata and Instrumentation [6]. The latter feature is the most influential for a DUPE middleware implementation.

Currently, this edition of the language has programming packages for most desktop operating systems, and as the J2SE 1.4 specification is most prevalent throughout mobile devices, we see that it is a likely future step that J2SE 5.0 will be

distributed as a standard. Furthermore, as J2SE 5.0 is, as yet, a common standard VM, it has complete backwards compatibility to version 1.4[1].

The class redefinition capabilities of J2SE 5.0 are provided by the Java Programming Language Instrumentation Services (JPLIS), which is a standard JVM package. Therefore, it is not necessary to include any separate Java packages. As a result, a standard J2SE 5.0 installation will always include dynamic redefining capabilities. This attribute allows DUPE 5.0 to be implemented within any operating system that has the J2SE 5.0 installed; the only requirements for classpath definition are those required by DUPE and the target application.

9.2.1 Java Programming Language Instrumentation Services

The Java Programming Language Instrumentation Structure (JPLIS) is new to the JVM. It is designed specifically for the instrumentation of Java byte code during runtime and initial class loading. The core constructs for class redefinition within the JPLIS are provided in the `java.lang.instrument` package, in particular the interface `java.lang.instrument.Instrumentation`. The design of the instrumentation setup is somewhat new to the Java architecture. The design makes use of the newly defined *agent* methodology. An agent is a class object which is capable of altering the runtime byte code of classes within the same application. All agents contain a `premain(..)` method. This method supplies the system with several other mechanisms that are associated with the JPLIS of the JVM. The `premain` method is called prior to the calling of the `main` method. Within any application there can be more than one agent running.

The construction of the agent class is essential to dynamic class redefinition, and should follow the guidelines of the appropriate api [106]. These guidelines state that, the Instrumentation class of an application will be provided during the initial startup of the JVM. To trigger the creation of an Instrumentation agent for an application, the following command-line command is used:

`-javaagent:`*`jarpath[=options]`*

[1]The author would like to note that although the official Java 5.0 specifications [6] profess Java 5.0 to be backwards compatible, we have at times found this to be incorrect.

In the command the *jarpath* is the Jar of an agent[2]. An agent Jar file will be specified within its associated manifest file through the use of specific attributes [106]. These are shown in Table 9.1.

Attribute	**Details**
`Premain-Class`	Identifies the agent class using the class name. This attribute is required; if it is not present the JVM will abort.
`Boot-Class-Path`	A list of paths, directories or libraries, to be searched by the bootstrap class loader. These paths are searched by the bootstrap class loader, in listed order, if the platform specific mechanisms of locating a class have failed. This attribute is optional.
`Can-Redefine-Classes`	Boolean value (`true` or `false`). Identifies if the agent is able to redefine classes. This attribute is optional, the default is `false`. It is generally recommended, however, if the agent is established for class Instrumentation.

Table 9.1: Manifest Attributes for an Agent Jar file

An example manifest segment is shown in Figure 9.2. An example of its use within `jarAgent.jar` could be:

```
java -javaagent:jarAgent.jar package.myInstrumentedApplication
```

```
Manifest-Version: 1.0
Created-By: Adrian Ryan

Premain-Class: package.MyAgentClass
Can-Redefine-Classes: true

...
```

Figure 9.2: Agent Attributes in a Manifest File

[2]More than one agent can use the same Jar file.

9.2.2 Achieving Redefinition in Java 5.0

Changing the class details during runtime is relatively simple within Java 5.0 as long as there is a means of gathering the new class details, aligning them with the old details, and making sure all startup options are set correctly.

The most important decision for the procedure is determining which class contains the instrumentation capabilities. Once that is established, the chosen class maintains, or is, an instance of the `java.lang.instrument.Instrumentation`. It must also include an implementation of the premain method. It may also provide an accessor method for the Instrumentation object, if it is required. This will allow other classes to use the Instrumentation object. It supplies the redefinition capabilities of the JVM to other classes that are not specifically designated as manipulative classes. This is a useful feature for classes such as those that discover class bytes. A very simple example of is provided in Figure 9.3.

```
import java.lang.instrument.Instrumentation;

public class MyInstrumentator {
        private static Instrumentation instInstance;

        public static void premain(String options, Instrumentation inst)
        {
                instInstance = inst;
        }
        public Instrumentation getInstrumentation(){
            return instInstance;
        }

}
```

Figure 9.3: `MyInstrumentor.java`

In this figure, the premain method assigns the Instrumentation object provided by the JVM to a variable. It is this variable which is used for class redefinition. The example provided in Figure 9.3 also allows the Instrumentation instance (`instInstance`) to be used by other classes. This is achieved by calling the `getInstrumentation()` method. However, as previously mentioned, this feature may not always be necessary.

The redefinition method of the Instrumentation object is:

```
public void redefineClasses((ClassDefinition[] definitions).
```

The argument of this method, (`ClassDefinition[] definitions`) is an array of classes that are set to be redefined. This becomes evident by looking at the specification of the ClassDefinition's sole constructor, `ClassDefinition(Class<?> theClass, byte[] theClassFile)`. It is simple to see that `theClass` argument is the current reference to the Class being redefined and `theClassFile` argument is the new byte code to be allocated to the class. The steps to achieve class instrumentation within an application are as follows[3]:

1. Start the application with correct command-line command; for example, `java -javaagent:myJarAgent.jar package.myInstrumentedApplication`
2. Gather the new byte code for a particular class.
3. Find the current Class object of the class to be redefined.
4. Create an instance of ClassDefinition using the new byte code and the current Class object.
5. Create an array of all the ClassDefinition objects, even if there is only one object.
6. Use the Instrumentation object, found within the target Instrumentation class, and call the redefineClasses method using the ClassDefinition array.

The most difficult stage of the above procedure is obtaining a reference to the current Class object. This object reference is only obtainable if:

a. any current object holds an instance of the required class and it can be used to gather a Class reference using reflection.

b. there is access to the instantiating class loader of the required class. This class loader is able to obtain a reference to the Class object by using the `Class findLoadedClass(String name)` method.

Limitations of J2SE Instrumentation

The J2SE 5.0 API [106] clearly details the restrictions of its Class redefinition capabilities. However, it also states that these restrictions may be lifted in later releases. The details of the current limitations are summarised in Table 9.2.

[3]Steps 2 and 3 for instrumentation are interchangeable.

Enabled Abilities
Any **redefinition** of any **method** body is allowed.
Any **redefinition** of any **attribute** is allowed.
Any **redefinition** of the **constant pool** is allowed.
Disallowed Abilities
Any **addition**, **removal** or **change** of **fields** is not allowed.
Any **addition**, **removal** or **change** of **methods** is not allowed.
Any **change** of **method signatures** is not allowed.
Any **change** in the **inheritance structure** is not allowed.

Table 9.2: The Redefinition Restrictions set by J2SE 5.0

From this table it can be seen that the instrument package allows any class redefinition to change its class method bodies, the state of the constant pool, and current class attributes. However, any redefinition of a class must not add, remove or rename, fields or methods, change the signatures of methods, or, change the inheritance structure of the class.

9.3 General Features of DUPE 5.0

DUPE 5.0 is able to achieve all the features of the DUPE framework. The three aspects of the DUPE framework, Distributed Connectivity, Dynamic Adaptation and Community Cooperation are all available within DUPE 5.0. However, as can be derived from the previous section, there are restrictions on the byte code used for class redefinition which directly affect the adaptation capabilities of the implementation.

Table 9.3 provides a list of the packages used within DUPE 5.0. The table identifies the package and its purpose.

It is clear from the table that a new `dupe.dynamic` package has been included from the DUPE Lite implementation. This package along with the `dupe` and `dupe.discovery` packages determines the progression of the target application and controls the redefinition of classes. Consequently, many classes within these packages are specifically designed according to the requirements of the implementation.

Package	Details	Differences
`dupe`	Base package of DUPE contains startup file.	Different in both.
`dupe.common`	Holds the common classes that are used in other packages.	Common to both.
`dupe.config`	Configuration setup classes. Acts as separate application.	Minor differences.
`dupe.discovery`	Community Management (DCLS)	Some Major Differences.
`dupe.dynamic`	Dynamic Control (DCM)	Completely Different.
`dupe.security`	Security Control	Vastly Similar.
`dupe.versioning`	Adaptation Reasoning	Common to both.

Table 9.3: Package structure for DUPE 5.0.

In Table 9.4 we detail the implications of the restrictions of the J2SE 5.0 JVM for DUPE 5.0. Unfortunately, there are no settings (for the current release of J2SE 5.0) that may be applied to alter any of these restrictions. The table is, therefore, a situation redefinition of the general J2SE 5.0 instrumentation restrictions presented in Table 9.2.

9.4 Implementation Construction

The structure of DUPE 5.0 maintains exactly the design aspects of the frameworks specification. All requirements of a DUPE compatible middleware remain and communication with other DUPE compatible middleware is possible. We presented the implementation structure of DUPE 5.0 in Figure 9.1. This figure clearly shows that we did not need to apply any alterations, as it can be seen that it replicates exactly the framework we described in Chapters 5 and 6. The figure also shows that many of the implementation design sections are exactly the same as that applied in DUPE lite.

We will now detail the elements of the implementation that differ from DUPE Lite (the blue sections in Figure 9.1). Although we will discuss the minor differences of similar implementation sections, we will not discuss any section in detail where the complete analysis can be derived from the previous chapter.

Abilities	**Implications**
Enabled Abilities	
Redefine method body.	This allows DUPE 5.0 to redefine all methods within a class that are associated with a resource.
Redefine attributes.	This allows DUPE 5.0 to redefine all method attributes within a class that are associated with a resource.
Redefine constant pool.	This allows DUPE 5.0 to redefine all details of all instances of the class, associated with the resource, via alterations to the constant-pool details.
Disallowed Abilities	
Can not add, remove or change fields.	New resources must not add, remove or change any fields from the class they are associated with.
Can not add, remove or change methods.	New resources must not add, remove or change any methods from the class they are associated with.
Can not change method signatures.	The method signatures of the new class details within the new resource must remain the same as the originals.
Can not change inheritance.	The inheritance structure of the class must remain the same as the original. Particular care must be taken not to add any new parent, or interface class.

Table 9.4: The Adaptation Attributes of DUPE 5.0

9.4.1 Distributed System Communication and Code Sharing

The manipulation of the target system within DUPE 5.0 is achieved slightly differently to DUPE Lite. As a result of this alteration, there are several significant changes in many of the core classes. The classes most affected are a part of the DUPE Class Loading Structure.

DUPE Class Loading Structure (DCLS)

As in DUPE Lite, the `dupe.discovery.RemoteUploadClassLoader` (RUCL) is again a significant class. It provides the necessary connection to Java's class loading structure. This connection is more essential within this implementation as the J2SE 5.0 instrumentation mechanisms rely on the access to Class objects that are only available using a class loader (see Section 9.2.2).

The RUCL is an initial point of control for the entire implementation, and as a result, all the required instrumentation attributes, including the `premain` method and the Instrumentation object, are within its structure. Furthermore, as the RUCL maintains a close relationship with the target application, it is the only class that is able to gather a reference to Class objects. Therefore, it also maintains a constant connection with the Dynamic Class Manger (DCM). This is specifically for use during class redefinition. In fact, the redefinition of a class will be finalised by the RUCL using the `redefineClasses` method provided by its Instrumentation object.

Although the redefinition capabilities of the structure are completed here, that is the only relationship it has with the DCM. All the other necessary computation for determining the needs, purpose and requirements for class redefinition remain in the DCM or the Adaptation Control and Security Control sections; as specified by the framework design.

Target Application

The target application is governed in the same manner is in DUPE Lite. However, as a result of the requirements for J2SE 5.0 instrumentation, the start up command of DUPE 5.0 is different. To begin an application within the DUPE 5.0 middleware the following command line is used:

```
java -javaagent:DUPE_JAR -Djava.security.policy=d:\policy.all
  -Djava.rmi.server.codebase=http://192.168.0.100:8080/
  dupe.Dupe coffeeMachine.CoffeeMachineGUI 192.168.0.100
```

where the line follows the procedure

```
java -javaagent:dupe.jar -security_options -jini_options
  dupe.Dupe target_application arguments_for_target
```

9.4.2 Dynamic Systems Alteration

Java Virtual Machine

The entire application, DUPE 5.0 and target application, is initialised and run within the same J2SE 5.0 JVM.

Dynamic Class Manager

Reloading class details using J2SE 5.0 requires, as previously mentioned, access to the original Class object. This object is available by using the instantiating class loader or any instance of the class. The DUPE framework does not have the ability to keep track of the class objects, it only controls the loading of class files. However, as all classes originate via the RUCL, it is possible to find a Class instance by using its `findLoadedClass` method.

The overall process of redefining a class is achieved within the DCM. However, as previously mentioned, the initial Class reference is obtained using the RUCL as is the final Class redefinition. The DCM will be given the details of a class in the form of a DynamicClass. The class name found within this object is then used to locate the current version of the class. Once this information is gathered it is sent to the RUCL and the `redefineClass(...)` method is invoked. This process is according to our discussion during Section 9.2.2.

Class List Manager

The Class List Manager (CLM) is exactly the same as in DUPE Lite (Section 8.3.2). However, the information of the classes is obtained in a different manner, and this implementation has more control over the DynamicClass instances. These added aspects of the CLM are a result of the runtime class redefinition

capabilities of the implementation. This design of the CLM remains for DUPE JPDA.

9.4.3 Community Interaction and Cooperation

The only aspect of DUPE community interaction that has altered from DUPE Lite is the addition of runtime event generation. In DUPE Lite, event generation derives from the loading of a new class, either from internal class details or externally downloaded details. In both DUPE 5.0 and DUPE JPDA, an event is also generated after the redefinition of a class.

For all events the details to update the DSO are to be provided to the Event Generator in the form of a DynamicClassEntry. All other elements of the events remain the same as detailed in the previous chapter.

9.4.4 Security Control

The package `dupe.security` is designed so that it may be adopted by a different implementations of the framework. Consequently, we have made use of this package for the security of all our implementations. However, for DUPE 5.0 and DUPE JPDA, the security is continuously active during the entire execution. This is for extended protection during runtime adaptation, which was unnecessary for DUPE Lite.

The consistency of this element of the framework implementations has allowed us to determine whether:

a. the security measures can be applied to different framework implementations, and,

b. if the security measures are defined well enough so that cooperation between different middleware remains possible.

We will discuss the analysis of these points in Chapter 11.

9.4.5 Adaptation Control

As with DUPE LIte, the adaptation section of DUPE 5.0 is the most accessible section for a user. And, as we have done with the `dupe.security` package, we have created a flexible `dupe.versioning` package to supply all adaptation controls to the two implementations.

Therefore, the initial adaptation settings of DUPE 5.0 can be applied to the main identification attributes within the manifest file. For DUPE 5.0, the settings that are applicable achieve the adaptation restrictions presented in Table 9.5.

Specification	**Setting Details**
Specification Version	Can be set to allow only evolutionary change. For example a class with the Specification Version number 23.01 will only adapt to a resource with a version number greater than 23.01.
Specification Title	Is used to control and specify that a new resource must be from exactly the same Specification Title. This is set to 'true' for situations where manufacture title identification is necessary, and should be 'false' to maintain a more flexible adaptation such as adapting according to location resources.
Specification Vendor	This should be used in a similar manner to the Specification Title, however, it creates even more restriction by precisely maintaining the vendor of the original class. A false setting is recommended within a trusted community as it enables the systems to achieve more precise adaptation.
Implementation Version	This is used in the same manner as the Specification Version except it is applied to Implementation details.
Implementation Title	This is used in the same manner as the Specification Title except it is applied to Implementation details.
Implementation Vendor	This is used in the same manner as the Specification Vendor except it is applied to Implementation details.

Table 9.5: Adaptation Settings for DUPE 5.0

9.5 Configuring Dupe 5.0

The configuration of DUPE 5.0 is achieved via a GUI similar to the design in DUPE Lite. This is possible as the security and adaptation controls are exactly the same, and as such, can be manipulated in the same manner; that is:

- via the supplied GUI, or
- by altering the DUPE configuration files within DUPE's bin directory.

9.5.1 Configuration GUI

The configuration GUI for all implementations are so similar that it is unnecessary for us to include it here. The general picture of the configuration application can be seen in Figure 8.3.

This configuration setup remains the same for DUPE JPDA.

9.6 Limitations

It is likely that all implementations of the framework will be limited only by the technique they use for dynamic instrumentation. This is the case for DUPE 5.0.

This implementation maintains all discovery and community cooperation requirements of the DUPE framework, and when tested (see Chapter 11), is capable of being used as the middleware for productive DUPE members.

9.7 Summary

The J2SE 5.0 implementation of DUPE, DUPE 5.0, is a fully compatible implementation of the framework. The requirements of the J2SE 5.0 redefinition components are made accessible with the standard framework design, and only slight modification to the connection of the framework components needed to be carried out. This implementation is able to execute standard Java programs. However, it contains the security restriction of ClassLoader redefinition that we

discussed in Chapter 7.

The next chapter provides another complete framework implementation which includes advanced security features using the J2SE 1.4 HotSpot, DUPE JPDA. The DUPE JPDA implementation has many similar sections to DUPE Lite and DUPE 5.0, therefore, as with this chapter, only the aspects unique to DUPE JPDA will be detailed.

Chapter 10

DUPE JPDA Implementation

10.1 Introduction

This chapter details DUPE JPDA. This implementation of the framework is the most complete of our implementations, it includes all requirements of a DUPE compilable middleware and advanced security features. DUPE JPDA is built using the JPDA mechanisms of the J2SE 1.4 HotSpot JVM [103]. This virtual machine was specifically chosen for DUPE implementation as it enables us to implement advanced security features. These security features overcome the security flaw that we pointed out in Chapter 7.

The elements that alter the DUPE JPDA design structure significantly from that described in the previous chapters are all due to the uniqueness of its class redefinition technique. These alterations, however, were not detrimental to DUPE community interaction, and as a result these aspects of the design remain as originally described for DUPE Lite (see Chapter 8).

This chapter is structured the same way as the previous chapter. We reintroduce the DUPE framework and show the areas that are specific to this implementation. These areas are indicated in blue in Figure 9.1.

To begin our discussion we now provide a detailed analysis of the J2SE 1.4 HotSpot JVM.

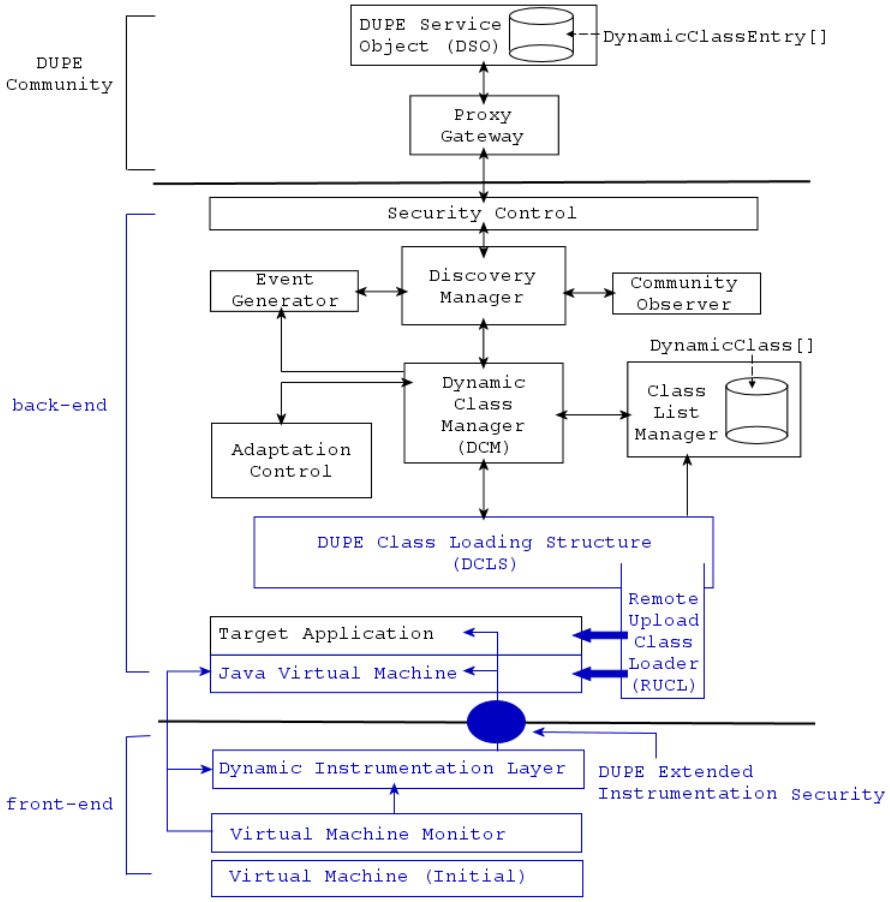


Figure 10.1: Structure of DUPE 5.0

10.2 Java 1.4 HotSpot Virtual Machine

The Java HotSpot Virtual Machine, version 1.4.2, is probably the most widely adopted JVM. The runtime aspects of this release include fast thread synchronization, garbage collection, and improved support for debugging and profiling [103]. It is the unique debugging capabilities of this JVM, including dynamic updating, that allow it to be applied as the base JVM for a DUPE implementation.

The language constructs that may be applied in an application using the HotSpot JVM are extensive. They include, the packages from the standard Java API as well as those provided for its extended capabilities. Unfortunately, the added features of this JVM also provide it with a large footprint during the execution

of any application[1].

As mentioned, this version of the JVM also includes dynamic updating capabilities. This is provided by the incorporated Java Programming Debugger Architecture (JPDA). The JPDA is an additional package for the JVM implementation. As this section of the JVM is the reason why it is used for a DUPE implementation we will now focus our discussion on the details of its workings.

10.2.1 Java Programming Debugger Architecture

The Java Programming Debugger Architecture (JPDA) is the debugging centre for the J2SE 1.4 HotSpot JVM. The only requirement for using it is the inclusion of the `lib/tools.jar` file as an addition to the JVM's classpath. The JPDA consists of two main interfaces, one protocol and two software components. The components of the JPDA are listed in Table 10.1 (details for the table come from Sun Microsystems' technical publications [103, 99]).

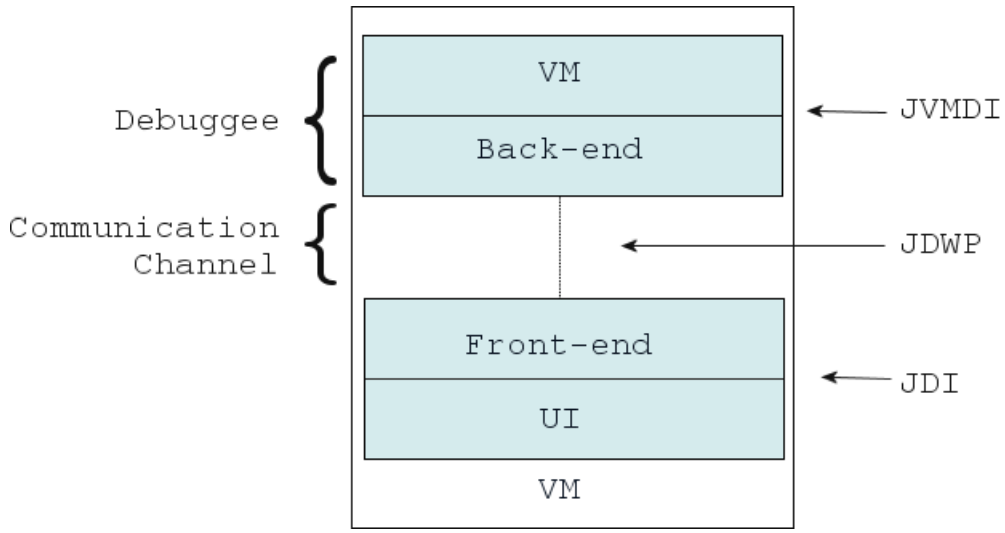


Figure 10.2: The JPDA Structure

[1]We will only discuss the sections of the HotSpot JVM that enable an implementation of the DUPE framework to be constructed. A complete specification of the JVM would be significantly large and outside the scope of this work. Further details of the HotSpot JVM can be found in Sun Microsystems' HotSpot white paper [103].

Section	**Details**
Interfaces	
Java Virtual Machine Debugging Interface (JVMDI)	Describes the functionality that is provided by the VM which enables the debugging of applications.
Java Debug Interface (JDI)	A pure Java interface for debugging Java applications. It is the high level interface which enables programmers to control the state of the VM
Protocol	
Java Debug Wire Protocol (JDWP)	Defines the format of the information that is transferred between the VM and the implementation of the JDI.
Software Components	
Back-end	Controls the application which is under debugging. The back-end is responsible for communicating requests from the debugger front-end to the debuggee VM and for communicating the response to these requests (including desired events) to the front-end, using the JVMDI.
Front-end	Determines the usage of the JPDA by implementing the JDI. All communication is achieved via the communication channel using the JDWP.
Others	
Debuggee	The general term used to identify the process that is being debugged. It consists of the application, the implementing VM and back-end of the debugger.
Debugger	The implementation of the JPDA structure which is controlling the debuggee.
Java Virtual Machine (VM)	The VM running the application being debugged. We also show that there is an initial VM running the Debugger.
Communication Channel	The Link between the front-end and the back-end which provides transport mechanism for all messages as specified by the JDWP.
UI	The general term used for the implementation tool that uses the JPDA structure. For our work this can be seen as the rest of the DUPE framework implementation; specifically, the DCLS and DCM.

Table 10.1: Components of the JPDA's Structure

The relationship among all the components and their relationship to the JVM are shown in Figure 10.2. From the figure it can be seen that the JPDA structure makes use of two Virtual Machine (VM) areas. The first VM is the initial execution JVM. It controls the front-end and any user interface (UI). This VM is the standard JVM used by normal applications. The second VM is controlled by the Java Virtual Machine Debugging Interface (JVMDI). This is an instance of the `com.sun.jdi.VirtualMachine` class that is instantiated by the first JVM. It acts as an application in itself and provides the back-end section of the JPDA structure (shown in Figure 10.2). This dual VM feature allows an application to be executed whilst also being controlled and watched by a separate Java debugging application. However, this feature also makes the JPDA execution slow due to high resource use. The term debuggee is used to describe the back-end section of the final system. The debuggee essentially contains a target application executing within a monitored VirtualMachine object. The other section of the structure is known as the debugger. The debugger monitors and controls the execution of the debuggee.

The JPDA allows the debugger to monitor the execution of the debuggee through VM events. The events provide runtime information on the application running within the monitored debuggee. All events must initially be requested by the debugger using specified VirtualMachine methods and calling for a specific event; for example, creating an instance of the `com.sun.jdi.request.MethodEntryRequest`. The events are then heard by the debugger as `com.sun.jdi.event.MethodEntryEvent` objects. For example, a request for all instances of the MethodEntryEvent is achieved by using the `methodEntryRequesting(...)` method, shown in Figure 10.3. Once the event has been requested there are several other manipulative methods that can be used by the debugger to control the exact timing of all event generation. This includes the ability to filter out specific packages, or only include a specific package (this is also shown in Figure 10.3).

```
public void methodEntryRequesting(EventRequestManager erm) {
    MethodEntryRequest mer = erm.createMethodEntryRequest(); //create request
    /*Filter options placed here*/
    //mer.addClassFilter("package.Class"); //example filter
    mer.setSuspendPolicy(EventRequest.SUSPEND_ALL); //specify suspend rule
    mer.enable(); //enable the request
}
```

Figure 10.3: Requesting a MethodEntryEvent from the Debuggee VM

When heard, VM events provide all the necessary information and set the correct VM status for dynamic class updating. For example, in Figure 10.3 each MethodEntryEvent will cause the debuggee VM stack to lockout general execution calls until unlocked. This could be a matter of nanoseconds or seconds depending on the reason. It is during this time that the debuggee application can be altered. The MethodEntryEvent is used within DUPE JPDA for this reason.

Within the JPDA system structure, the target application, which is part of the debuggee, should run as normal. This requires that all command line variables are passed through and all VM settings are applied in the necessary manner. This is achieved within the JDI section of the application using the `com.sun.jdi.VirtualMachine` and the `com.sun.jdi.connect.LaunchingConnector`, and must be programmed into each implementation of the JPDA structure. The LaunchingConnector object will provide an initial connection between the back-end and the front-end (the communication channel) and maintain this connection for the entire execution of the VM. These sections are all necessary to allow safe execution and safe dynamic updating of the target application.

10.2.2 Achieving Dynamic Updates in the HotSpot JVM

The JPDA provides the ability to dynamically alter a class structure through use of its HotSwap class file replacement mechanism [103]. This section of the JPDA is a part of the functions within the higher levels of the JDWP and the JDI and can only be called on the debuggee if it is being monitored by the debugger using the JPDA structure.

The process of class redefinition during runtime involves two main steps:

1. There must be a reason for the alteration. Specifically, the use of the VM event mechanism is used to pinpoint the exact location and time period for the class update. Moreover, an event can only be generated as a result of the execution of the debuggee applications. Therefore, events such as the MethodEntryEvent and the StepEvent can be used by the debugger system to identify when a particular class is held in an accessible section of the JVM stack. The use of the VM event mechanism in this way is necessary for class redefinition.

2. The JVM stack enables the debugger to gain access to the class structure of the debuggee without causing complications to its execution. To ensure

the safety of the class, and any current objects, the events used to trigger the update will cause the JVM stack to lock. This, in turn, enables the current class frame to be popped, altered and replaced safely [99].

The methods within the debuggee VirtualMachine object which provide the necessary requirements for class instrumentation are:

1. `public void redefineClasses(Map classToBytes)`
2. `public void popFrames (StackFrame frame) throws IncompatibleThreadStateException`

However, popping the frame from the stack is not necessary if the front-end is able to establish correctly when the class to be altered is present in the stack. And, as mentioned earlier, it is possible to achieve this by using the VirtualMachine events (some of which will cause the stack to lock automatically).

To initialise the redefinition of a class the deguggee's VirtualMachine object must be provided with the correct information on the altering class object in the form of a Map argument. The Map contents consist of at least one instance of `com.sun.jdi.ReferenceType`. This class contains the current class version details along with the new set of verified class bytes (in the form of a byte array) [103]. The ReferenceType object representation of any class can be obtained from the VirtualMachine object by calling the method `List classesByName(String className)` using a full class name and gathering the correct class (if there are multiple versions loaded) from the returned List.

If necessary, the popping of frames can be achieved using the current `com.sun.jdi.ThreadReference` object which is only available from the VirtualMachine whilst the stack is locked. The class redefinition is then achieved using the VirtualMachine method `public void redefineClasses(Map classToBytes)`. Once redefinition has occurred within the VirtualMachine both the debuggee and the debugger return to executing as normal. The only difference is that the debuggee application is now using the new version of the class throughout all the associated class instances.

10.2.3 Limitations of JPDA Class Redefinition

There are some limitations to the class redefinition capabilities of the HotSpot VM. As defined within the JDI API [96], the redefined method of the VirtualMachine interface is the final point in detecting incorrect class definitions. Any incorrect changes to the debuggee application will result in this method throwing a `java.lang.UnsupportedOperationException`. The checks that determine this are:

- check if the VM allows class redefinition. Can be assessed using the `canRedefineClasses()` method.
- check if the VM allows a new class definition to include new methods. Can be assessed using the `canAddMethod()` method.
- check if the VM allows complete unrestricted redefinition of classes. Can be assessed using the `canUnrestrictedlyRedefineClasses()` method. This feature allows a new class definition to:
  - change the schema (the fields)
  - change the hierarchy (subclasses, interfaces)
  - delete a method
  - change class modifiers
  - change method modifiers.

Generally, when using the JPDA, the JVM will be set to allow class redefinition, method redefinition and method addition. The allowance of unrestricted features is dangerous to the execution of a system unless the entire class structure is redefined. For example, the deletion of a class method that continues to be called by another class could eventually cause a fatal error. Therefore, in general some limitations remain in place.

10.3 General Features of DUPE JPDA

DUPE JPDA is able to achieve all the features deemed necessary by the DUPE framework. However, as indicated in Section 10.2.3, the JPDA has limitations which directly affect the adaptation capabilities of the implementation. Yet, these limitations are not detrimental to the overall cooperation of DUPE JPDA

enabled systems, they only effect their level of adaptation.

As with DUPE 5.0, DUPE JPDA maintains all DUPE community discovery specifications by implementing all relevant interfaces, and constructing the DynamicClassEntry class and all requirements of the DSO. Table 10.2 provides the package list for DUPE JPDA.

Package	**Details**
`dupe`	Base package of DUPE contains startup file.
`dupe.common`	Holds the common classes that are used in other packages.
`dupe.config`	Configuration setup classes. Acts as separate application.
`dupe.discovery`	Community Management (DCLS).
`dupe.dynamic`	Dynamic Control (DCM).
`dupe.security`	Security Control.
`dupe.versioning`	Adaptation Reasoning.

Table 10.2: Package Structure for DUPE JPDA

In the previous section, we listed the features of JPDA which limit its class redefinition capabilities. These limitations correspond to the dynamic runtime class redefinition limitations of DUPE JPDA. Therefore, we are able to construct the types of resources that DUPE JPDA is capable of using. In Table 10.3 we list these limitations and recommended system settings according to the DUPE categories previously identified in Chapter 7.

Although these limitations restrict the resources that are used during adaptation, they do not restrict any other section of the framework; they are simply a reflection on the JVM applied to this particular implementation. The limitations of the JPDA, or those that may be applied, are common to most dynamic virtual machines as a result of type safety requirements for executing applications.

10.4 Implementation Construction

The structure of DUPE JPDA maintains, as close as possible, the design of the frameworks specification. However, due to the requirements of the JPDA structure, sections of DUPE JPDA do not match exactly. Irrespective of this, DUPE

Setting	**Implications**
Class Redefinition	
true	Allows DUPE JPDA to redefine classes associated with all resources. Does not, however, determine any further specifics of a new class definition, and by itself will only allow redefinition based on a class with the same methods (only the methods algorithms may differ). This should be applied always as the base setting.
false	Completely limits the use of all DUPE resources and disables the adaptation capabilities of the framework. This setting is not recommended and should never be used for a DUPE JPDA instantiation.
Add Method	
true	Enables a redefining class to include new methods within its structure. This is generally not seen as detrimental to the overall system structure and is recommended.
false	Disables the ability of the new resource class definition to include new methods.
Unrestricted Redefinition	
true	Allows full alteration of the system structure according to resources from the community. This setting should be only applied when trust levels are set high and adaptation can be controlled on a basis of vendor and/or class version with backwards compatibility applied.
false	Will disallow the extended adaptation of classes; should be used as default.

Table 10.3: The Adaptation Setting of DUPE JPDA

JPDA is a fully compatible DUPE middleware (this will be shown in Chapter 11). Figure 10.4 is a re-illustration of the design layout of DUPE JPDA using both the specification design of DUPE and the JPDA structure (shown in Figure 10.2) for illustrative purposes.

The figure details the JPDA event path (noted using a red path arrow) that is generated when class redefinition, as determined by the DCM, is triggered. This part of the framework is where it differs slightly to that of the original framework. However, this is specific to the JPDA design and cannot be avoided. As in Chapters 8 and 9, during the discussion of DUPE JPDA we will only identify the key classes of the implementation.

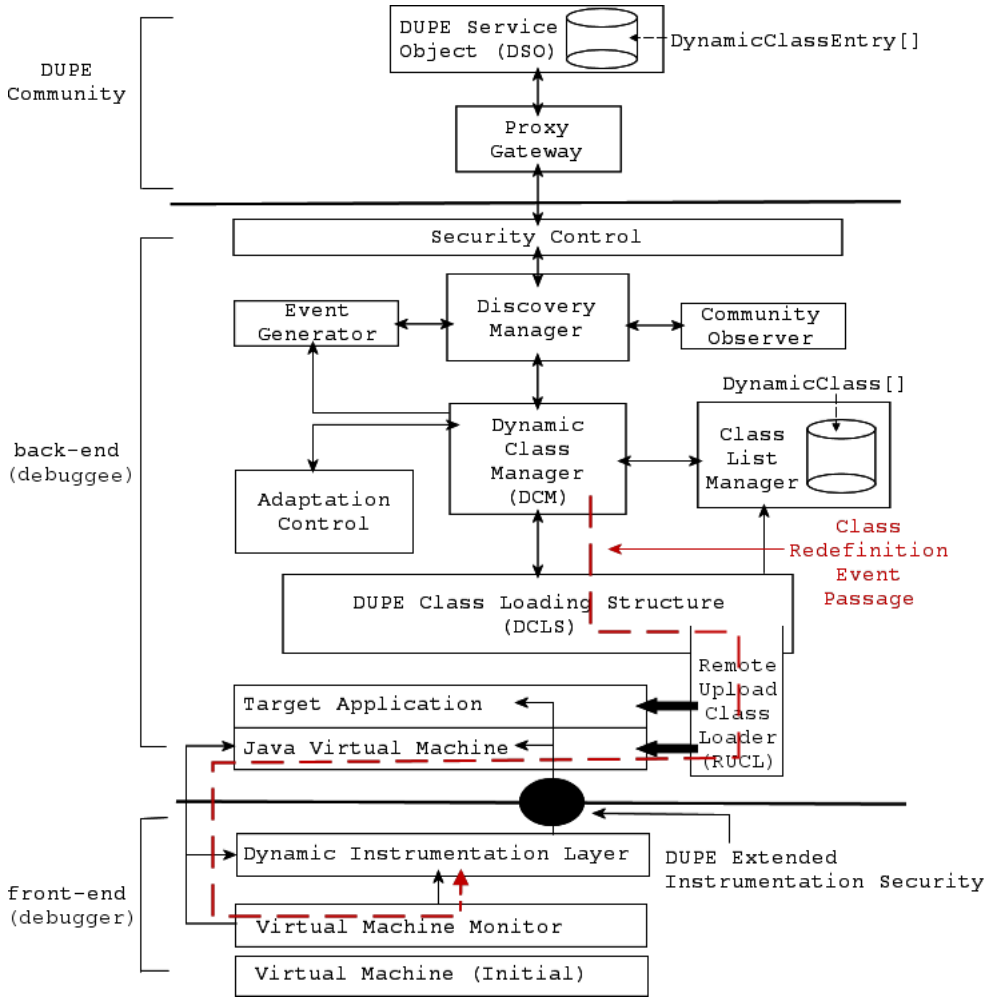


Figure 10.4: Structure of DUPE JPDA

The package structure of this implementation is exactly the same as the DUPE 5.0 package structure. However, many of the classes within the `dupe`, `dupe.discovery` and `dupe.dynamic` packages are different.

10.4.1 Distributed System Communication and Code Sharing

As with our other DUPE implementations, the core elements of the DUPE Class Loading Structure (DCLS) are within the `dupe.discovery` package of DUPE JPDA. And, again the most important class is RUCL implementation. However,

as the RUCL is DUPE's main class loader it must have access to all loaded class details. To obtain this access, the RUCL must be implemented as part of the back-end structure as this is where all classes are loaded and manipulated (depicted in Figure 10.4). It is not possible for the RUCL to be in the front-end as it must not only control the loading of the classes, but also indicate to the DCM when a class cannot be located locally and trigger a community search. This indication is only achievable if it has a direct connection with the bootstrap class loader of the target application. Again, this aspect of the implementation is in the back-end.

Target Application

The target application is initialised by the front-end, as a part of the back-end, using an instance of `com.sun.jdi.VirtualMachine`. It is essential that the target application is executing within the RUCL, and that the RUCL is able to return information to the rest of the DUPE structure for necessary adaptation and loading requirements. Therefore, the RUCL is initialised first as a separate application (along with many other sections including the Discovery Manager and Security and Adaptation) in the back-end. The target application is then executed using the RUCL as it based class loader. The updating of the target application is triggered by calling the `redefineClass` method of `dupe.discovery.RemoteUploadClasLoader`.

As a consequence of this design structure, when initialising the VirtualMachine object, the front-end will append the DCLS to the initial class call and use that to begin the application. This process is as follows:

1. Application started within DUPE JPDA using a command such as:

   ```
   java dupe.Dupe -Djava.security.policy=d:\policy.all
     -Djava.rmi.server.codebase=http://192.168.0.100:8080/
     coffeeMachine.CoffeeMachineGUI 192.168.0.100
   ```

   where the line follows the procedure:

   ```
   java dupe.Dupe -security_options -jini_options
     target_application arguments_for_target
   ```

2. The front-end structure will gather all required information for target application and append `dupe.Remote`[2] to the start of the initialisation command.

3. The VirtualMachine object will begin executing `dupe.Remote`.

4. Upon start up, `dupe.Remote` will initialise the target application using all appropriate JVM settings and system arguments.

This procedure provides the front-end with the ability to receive communications, in the form of a `com.sun.jdi.event.MethodEntryEvent`, from the back-end based on the execution of the target application. The events are a result of the Virtual Machine Monitor listening specifically for calls to class loading methods of the RUCL (shown in Figure 10.4).

10.4.2 Dynamic Systems Alteration

The class redefinition aspects of the structure are controlled by the JPDA structure, hence, the reason for the design changes from the original framework. It is essential that the VirtualMachine object contains the RUCL. To listen for the call to the redefineClass, a Virtual Machine Monitor has been included within the discovery package, `dupe.discovery.VMControl`. This class controls the dynamics of all class updating by triggering VM events where appropriate. This design is for implementation purposes and the VMControl class must handle all the stages of physical class redefinition. The main components of this framework section are now discussed.

Java Virtual Machine

The DUPE JPDA application and the specified target application are initialised using the J2SE 1.4 HotSpot VM with JPDA setup completed. Within the structure itself, an instance of VirtualMachine is established controlling the back-end section of the structure.

Dynamic Class Manager

Most parts of the Dynamic Class Manager (DCM) are found within the back-end as they provide the essential class information for adaptation. This allows

[2] `dupe.Remote` is the initialising class for the DCLS.

the system to execute correctly by minimising the communication that is passed between the front-end and the back-end.

As discussed previously, the reloading of a class is controlled by the front-end and is triggered when the back-end calls the redefineClass method of the RUCL. Once this event is heard by the front-end, the JPDA elements of the implementation gather all the required details from the executing stack (according to the procedures discussed in Section 10.2). These details include the new class byte code and the current class details. Once the redefinition map is created, the redefine method is called on the VirtualMachine object by the VMControl object, and class redefinition proceeds. The step by step process of redefinition is as follows:

1. The Community Observer discovers a community resource.
2. Adaptation Control and Security Control are used to verify the resource.
3. If appropriate, the DCM instructs the Community Observer to gather the complete DynamicClass details from the resource.
4. All Security and Adaptation controls are rechecked.
5. When appropriate, the DCM instructs the DCLS to redefine the class associated with the DynamicClass instance.
6. Subsequently, the DCLS will call the `redefineClass()` method.
7. This `redefineClass` call causes the associated VirtualMachine to generate an instance of MethodEnteredEvent[3].
8. VMControl hears the MethodEnteredEvent and checks the redefinition status.
9. If required, the VM stack access is locked.
10. As the VMControl has access to all classes within the back-end, it is able to gather all necessary information for class redefinition from the stack. This information is:
    - the current class object, and
    - the new class byte code.

[3]The MethodEnteredEvent is set to only generate an event if the DCM enters the `redefineClass` method.

11. The VMControl then instructs the VirtualMachine to redefine the class by calling the `redefineClasses()` method with the constructed Map of class information.

12. The stack is released and the back-end progresses using the new class details.

10.5 Limitations

The limitations of this implementation are the restrictions set by the JPDA structure (see Sections 10.3 and 10.2.3 for details).

10.6 Systems in DUPE Communities

From the earlier chapters on DUPE's design and our implementation chapters, it may be established that there are some slight constraints on a target application that can be executed using a DUPE middleware. Currently these are compilation, distribution and execution constraints. These requirements were identified in Section 7.4.1. In brief they are:

- the target application must be compiled with the `-g` option.
- the target application must be contained in a Jar file and stored in the `dupe/trusted/` directory of DUPE file structure (the `dupe/trusted/` constraint is implementation specific, therefore, it may be different, or non-existent, for other framework implementations.).
- the associated Jar file must have an associated Manifest file that contains all adaptation information.
- the Jar file must be signed with a minimum of one certificate.

The constraints on the target application are consistent with common programming practices; for example, it is common to distribute Java applications using signed Jar files [109]. Consequently, we do not view these constraints as negative aspects of the frameworks.

10.7 Summary

The J2SE 1.4 HotSpot implementation of the DUPE framework, DUPE JPDA, is controlled by the class redefinition abilities of the JPDA, and is structured according to its requirements. It was therefore necessary to alter the design of the framework to incorporate a back-end and a front-end design. However, the general design element of the implementation remains the same, and all DUPE community cooperation complies with the specification of DUPE compatibility. As a result of the JPDA structure, DUPE JPDA is the most complete and safe of all our DUPE implementations.

The next chapter provides a DUPE framework evaluation. The evaluation is achieved by analysing the results of DUPE systems, working either separately or together within the same community, and recording details such as timing and memory use. In Chapter 12, we then model the movements of gypsy agents. In doing this, we aim to determine if this particular element of the framework's use provides a significant contribution to mobile system evolution.

Chapter 11

Evaluation

11.1 Introduction

This chapter provides a formal evaluation of the DUPE framework. For our evaluation, we give the results from both situational and technical measurement testing.

The situational testing we analyse is designed to evaluate the communication and cooperation aspects of DUPE communities. From this, we are able to test DUPE member cooperation in relation to:

- the same target application executing in the same DUPE middleware
- the same target application executing in different DUPE middleware
- different target applications executing in the same DUPE middleware
- different target applications executing in different DUPE middleware.

The technical evaluation is provided by analysing two sets of results from the testing of technical measurements. The first measurement is used to determine if the overhead of a DUPE framework implementation is of concern. This assessment is achieved using measurement tests from the JVM HEAP during DUPE system execution. The second is a set of technical tests to determine the load time differences between a system running with DUPE capabilities and a system executing as a stand alone application.

11.2 Testing Environment

We use the same testing environment for all tests. It consists of three or more components, where the number of components required is dependent on the testing scenario. These are listed in Table 11.1. The DUPE community is always established within a wireless 802.11b network, as this is currently the most widely used network for mobile device communication. However, due to limitations of device hardware and operating system, no measurements are given of the framework implementing within a mobile device[1].

11.3 Situation Tests

The situational tests that we analyse are designed to determine if the system interaction within a DUPE community holds true according to the DUPE framework and the requirements of a dynamic community. Specifically, the situations are for the purpose of analysing:

- **DUPE Community Interoperability**: during this analysis, we aim to determine if different DUPE middleware implementations are able to form DUPE communities, discover resources within DUPE communities and transfer resource details among other DUPE members. All tests are applied using heterogeneous DUPE implementations.
- **DUPE Member Security**: during this analysis, we aim to assess the security measures recommended, and applied, by the DUPE framework. The testing scenarios used within this section determine if the security measures are successful, and if they cause any unforeseen complications to DUPE community interaction.
- **Adaptation Tests**: during this analysis, we aim to analyse the behaviour of DUPE systems. The analysis obtained is the most detailed of all tests. We analyse different aspects of community resource transfer and system adaptation. Results are given in Sections 11.6, 11.7 & 11.8.

[1]We have discussed throughout the thesis that the DUPE framework is designed for mobile systems; however, as currently, mobile devices, in general, lack the support of a dynamic updating technique, we are unable to test such a circumstance. Consequently, we indicated in Chapters 3 & 9 that the reason for our choice of dynamic technique for DUPE 5.0 is that we predict J2SE 5.0 to be the first commonly dispersed dynamic JVM among mobile devices.

Component	Property
PC Desktop	• Pentium IV 3000Mhz • Windows XP OS • 1024MB RAM • 802.11b Dlink card
Compaq laptop	• Compaq Armarda V700 • Windows XP OS • 512MB RAM • Pentium II 330Mhz • 802.11b PCMIA Dlink card
Wireless Access Point	Dlink 802.11b
2nd PC Desktop	• Pentium II 2200Mhz • Windows XP OS • 512MB RAM • 802.11b Dlink card

Table 11.1: Components of the Testing Environment

11.4 DUPE Community Interoperability

The first situation test for community interaction is designed to determine whether community activity among different DUPE systems is achieved. We ran several test cases on several different platforms to analyse if different DUPE systems using different framework implementations can work together as a DUPE community. All scenarios involve the use of one, two or all of the test DUPE implementations: DUPE Lite, DUPE JPDA and DUPE 5.0. It is not imperative during this analysis for us to describe the target application specifics, nor is it necessary for the details of the transferred DUPE resources to be discussed. We test these aspects of the DUPE framework during Section 11.6, and during our technical testing, beginning with Section 11.9. For this initial testing scenario, we wish to keep the situation description simple. This allows us to concentrate on illustrating the level of interoperability obtained within DUPE communities. The results of the testing scenario are provided in Table 11.2.

Within Table 11.2 each junction block describes the level community cooperation that can be established using the respective DUPE middleware. The community cooperation includes:

- resource transfer
- resource analysis
- system updating, and
- complete system adaptation.

The results from this testing section are simple. However, what they tell us is that heterogeneous DUPE systems are able to discover and use each other's DSO services. This, therefore, indicates that the DUPE community language is successful.

11.5 DUPE Member Security

To measure the security aspects of the system, we use several testing scenarios to determine if the correct behaviour for DUPE systems is maintained irrespective of security constraints. We also want to determine whether the security and trust techniques are strong enough to stop a DUPE system from using a community

	DUPE Lite	**DUPE 5.0**	**DUPE JPDA**
DUPE Lite	Limited community. All members display load time updates only.	Limited community. DUPE Lite systems display load time updates only.	Limited community. DUPE Lite systems display load time updates only.
DUPE 5.0	Limited community. DUPE Lite systems display load time updates only.	Complete DUPE community interaction for all members.	Complete DUPE community interaction for all members.
DUPE JPDA	Limited community. DUPE Lite systems display load time updates only.	Complete DUPE community interaction for all members.	Complete DUPE community interaction for all members.

Table 11.2: Community Cooperation between Framework Implementations

resource that has either no *associated certificates* or is signed by an *non-trusted certificate*[2].

The testing procedures used identify if the community resources that are accessed derive from a trusted source and contain valid class file details. And, if they do not, the DUPE system will then dismiss them. We provide results of our security testing using two different scenarios:

1. **Behavioral Studies**: we apply tests on the behaviour of DUPE systems to determine if the security and trust measures interfere with the activity throughout a DUPE community.
2. **Certificate Recognition**: the use of certificate recognition determines the trust levels applicable to a DUPE system during its interaction within any DUPE community. We construct tests to determine if the trust aspects of the framework are met during execution, and if they are useful for DUPE security.

11.5.1 Security Test - Behavioral Studies

The aim of behavioural testing is to determine whether a DUPE community can continue to work with the signing of community resources incorporated. The behaviour of all DUPE compatible systems can be measured by monitoring the target application during three different stages of execution. These stages are:

- execution without DUPE compatibility (outside any DUPE framework)
- execution with DUPE compatibility, however, with all security set as inoperable[3]
- execution with DUPE Compatibility, all certificates valid, all members trusted and all security operational.

We applied the testing using all the DUPE implementations[4]. By testing the validity of the class bytes supplied by resources, we can determine if the contents

[2]We established in Chapter 7 that trust is determined on an individual basis using the frameworks Security Control settings.

[3]The security of the entire DUPE community is switched off.

[4]Although, this was not necessary as all implementations use the same security package, we tested all implementations to look for anomalies and to ensure that our testing regime was comprehensive.

of the resources were unaffected during transfer. During the tests, all DUPE members consisted of the same target application and were run in the same manner. Therefore, any successful adaptation resulted in full redefinition of a system class resource. The checks assessed during the analysis were:

1. **Application Execution**: will the execution of the target system be correct when all security options are in place?
2. **Discover Community**: is a DUPE system able to become a member of a DUPE community when all security options are in place?
3. **Transfer Resources**: as a DUPE member, was the DUPE system able to discover, advertise, transfer and obtain DUPE resources according to the DUPE security measures?
4. **Redefine Class Details**: is the DUPE system able to redefine its structure using appropriate resource details, while security on option is in place?

Table 11.3 gives a results of the resource transfer testing situations.

	DUPE Lite	**DUPE 5.0**	**DUPE JPDA**	**No DUPE**
Application Execution	yes	yes	yes	yes
Discover Community	yes	yes	yes	N/A
Transfer Resources	yes	yes	yes	N/A
Redefine Class Details	N/A	yes	yes	N/A

Table 11.3: Implementation Results of Security Checks

These testing scenarios cover all interactive aspects of a DUPE community. Therefore, as all situations produce positive results (this includes DUPE members only adapting where applicable), then the test results show that the security measures do not interfere with DUPE community operations. Table 11.3 clearly supports this assertion.

Further to testing if the DUPE systems interact correctly, irrespective of their security settings, the testing procedure also included negative testing. These testing scenarios were applied to test the validity of resource code by forcing incorrect data to be discovered within a DSO. This test established that when the byte code supplied by a DSO was an invalid format, according to Java class byte specifications, then, irrespective of any correct certificates and adaptation

settings, the class redefinition is aborted. This particular aspect of testing was an implementation test, not a framework test. We, therefore, could not extend this analysis.

11.5.2 Security Test - Certificate Recognition

Certificate recognition is the main trust mechanism of a DUPE community. However, the validity of a certificate is not determined by the DUPE framework but by the security and adaptations settings applied by the user (see Chapter 7 for full details on certificate recognition). Consequently, the testing procedures used to validate this area of a DUPE community only determines whether the DUPE framework allows DUPE middleware to check that a certificate exists within a DSO, that the resource code is signed by the associated certificates and that the remote code is only loaded by a DUPE system if it is determined valid. The testing is setup to find out if DUPE members can recognise resource instances where the certificate does not exist, or where the certificate is not trusted by the current DUPE system execution settings.

We established in Section 7.4.1 that all DUPE framework implementations require several aspects of a target application to be correct. These aspects were:

- the target application must be compiled with the `-g` option
- the target application must be contained in a Jar file and stored in the `dupe/trusted/` directory of DUPE file structure
- the associated Jar file must have an associated Manifest file that contains all adaptation information, and
- the Jar file must be signed with a minimum of one certificate.

The last two of these requirements are a direct result of the security mechanisms, while, the second requirement is a security precautionary setup.

The certificate checking tests that were carried out, using all DUPE implementations, adhered to all the security requirements. However, for testing reasons certificates were deleted from some DSO resources and invalidly signed code was used in others. These DSO's were used to examine the implications of such a situation. The results from all of our certificate testing are detailed in Table 11.4.

	DUPE Lite	**DUPE 5.0**	**DUPE JPDA**
Execute unsigned application	no	no	no
Execute application signed with untrusted certificate	no	no	no
Execute resource with invalid signature but valid certificate chain	no	no	no
Instantiate, or redefine, unsigned remote class	no	no	no
Instantiate, or redefine, untrusted remote class	no	no	no
Instantiate, or redefine, invalidly signed remote class	no	no	no

Table 11.4: Certificate Security Checks

Table 11.4 presents all the test results as negative, as required. The table is written as such to demonstrate the purpose of the test. Although, the certificate analysis testing is implementation specific, it does demonstrate that the use of certificates with the DUPE community allows member to govern their adaptation progress in accordance with their individual requirements.

11.6 Adaptation Testing

Adaptation tests are used to provide a study of the community interaction and demonstrate the cooperation aspects of the framework. There are many different reasons for adaptation. We limited our testing to the findings from two separate situation studies which were designed to cover as many of the adaptation requirements as possible. However, no matter how generalised our scenarios are there may be adaptation scenarios not covered. This is due to the open-ended nature of DUPE communities and system adaptation. All tests are designed for fully compatible DUPE systems.

Specifically, test scenarios show the results from the two main DUPE community examples we have previously used in the thesis (see Chapter 6):

- **The Home Community**: a case study of the interaction within a home based DUPE community.
- **Localised Adaptation**: a case study of the movements of DUPE members throughout localised communities using a tourist application as our test bed.

This method of analysis provides access to adaptation testing results from community interaction within the many foreseeable scenarios of a DUPE community. This includes the adaptation of gypsy agents; however, we cover these in more detail in the next chapter.

We have separated our testing scenarios into two sections (Section 11.7 and 11.8). As mentioned, each section is in the form of a case study, and therefore, the results are qualitative and not presented as quantitive measurements. It is suggested that the reader analyse the case studies to form a picture of the different adaptation circumstances being described. We will conclude each case study with our observations.

11.7 Adaptation Test - The Community

Home automation services are made possible through the use of techniques such as OSGi [84] and UPnP [72]. These technique provides home services with access to each other and to external information, including the internet, via the use of a home gateway and a communication channel. This test scenario is based on the concept that each home service, including the gateway, is running as a DUPE member of a DUPE community established within the home network.

A home automation system is designed to allow all services, in most cases appliances and devices, to interact and use each other's capabilities. Examples of its use include phone call notification where a user is informed of a phone call via the television, and device interaction in which a user using a coffee machine will be informed if there is milk in the fridge or not.

The DUPE framework enables each home system to evolve in accordance with the home system settings and the current status of the other services within the DUPE community. Moreover, as the home network is a DUPE community there are several different scenarios that may cause a system to adapt or evolve. These adaptations represent a significant number of the common adaptation situations that may occur in any DUPE community.

11.7.1 Scenario Analysis A - Addition of New System

In this scenario, a new DUPE system enters the community; for example, a friend with a PDA. The resources of the entering system can be in three different states according to the resources currently within the DUPE community. These states are:

i. **New System Containing New Resource Version**
The system entering a DUPE community contains a new resource version[5]; for example, when a communication package, that has a Specification Version of 1.2, is commonly used within the community upon entry of a new system, for example, a coffee machine, that contains the communication package with Specification Version 1.3, all systems will upgrade. To be exact, all fully compatible DUPE systems within the community will adapt to the state of the coffee machine's communication package. Limited DUPE systems will only adapt according to their limitations; for example, DUPE Lite systems will only adapt after a restart. However, irrespective of this, the adaptation occurring in this context also includes, if required, the adapting systems gathering new, previously unknown resources.

ii. **New System Containing Old Resource Version**
The system entering a DUPE community is running an older, or less specific, version of a resource package; again we use the communication package. In this case, because the current members of the community contain a more appropriate version of the communication package, the newly entered member adapts and alters its resource state to that present in the community.

iii. **New System Containing Same Resource Version**
The system entering a DUPE community contains the same version of a resource package currently in the community. In this situation neither the new member, nor the old members of the DUPE community change state.

11.7.2 Scenario Analysis B - Resource Upgrade

A permanently setup DUPE community provides software vendors with the ability to alter system components by placing them on any current member within

[5]In our scenarios we set the version update according to the `Specification-Version` variable of the Manifest file (see Section 7.3 for further details).

the community. A vendor may wish to use such a capability, for example, to repair a software bug or, distribute update software components. This is a concept we introduced in Section 6.7.

The new resource components will most likely be placed on the home gateway of the community, as this service provides a connection to the internet. When the component is placed on the gateway, the home gateway itself may adapt, if appropriate, and then generate a new community event[6]. Following this, all other home community members can adapt. However, as DUPE enables systems to use components found within their system structure, within their normal system reach, and, via resource discovery, a component may actually be placed anywhere in the home network, and a member would react in the same manner. This would include the placement of the resource of a visiting system within community for the sole purpose of upgrading community components. We tested this scenario in the following manner:

i. **Resource Placed on Home Gateway**
   A new class version, in a Jar file, was placed within a gateway system that was designed to find it within itself. According to the attributes of a dynamic community, when the gateway created an instance of the class found in the new resource, a DUPE community event was generated and all other members began to obtain the resource data, where appropriate. However, not all systems obtained the resource directly from the gateway. As more systems were running the new component version, they were able to distribute it throughout other community members. This meant that the gateway did not have to upload the resource contents to all of the DUPE members.

ii. **Resource Contained within Entering Systems**
   A new class version, in a Jar file, is contained within a system that enters the community. The results of this simulation were exactly the same as those shown previously in Section 11.7.1, Scenario Analysis A(i). The tests showed that although this scenario was designed for a different task, its fundamental, underlying adaptation scenario was the exact same concept as system evolution. Furthermore, as the current community members have adapted to the resource, when a new member enters the community it will also have access to it.

[6]If the home network does not adapt to the resource, if developed to do so, it can recognise the addition of the resource and notify the community that it is accessible.

11.7.3 Observations of Adaptation Test - The Community

The scenarios presented by the home DUPE community demonstrated several advantages that DUPE provides when some members are semi-permanent. It was interesting to observe the adaptation of a single system and the subsequent chain reaction of system adaptation throughout the entire community. Therefore, it is reasonable to suggest that if a new version of a generic component, for example a communication component, is placed somewhere within a community, then one by one each community member would adapt. Moreover, the scenarios also illustrated that each separate discovery of the system resource does not need to be achieved via the same member. We conclude that this particular feature of the DUPE member interaction is a result of the framework specification that all (most) community members can receive and send resources, as illustrated in Chapter 4. Thus, as a new resource is introduced to each member, the number of systems that offer the component as a service increases.

11.8 Adaptation Test - Localised Adaptation

This adaptation test is based on a dynamic tourist application. We developed a small program that can be executed within a DUPE middleware with the adaptation settings set for location based adaptation. The tourist system is a general tourist application. When implemented as a DUPE system it is able to execute a destination specific version of the application. The test in this scenario represents the adaptation of a wandering system adapting according to its current community and location. There are several different factors that determine this method of adaptation. All of these factors are associated with the adaptation settings of each DUPE system, as we discussed in Section 7.3. All the movements throughout a localised community maintain the common scenario of random community adaptation; consequently, it is not necessary to test all possible adaptation situations. We will now provide a test scenario which is designed to analyse the ability of a DUPE system to change state according to its current community.

11.8.1 Scenario Analysis

We designed a tourist application contained within the `tourist` Java package. It has two main Java classes:

`tourist.Tourist:` a Tourist object is the applications instantiating class. However, it is not the main class of the Tourist application. This function belongs to the `tourist.TouristInit` class. Moreover, when using a DUPE middleware, this class is the only file that is needed on a user's system. The Tourist class is extremely small in design so it would easily fit on a resource limited device.

`tourist.TouristInit:` the TouristInit object, as previously mentioned, is the most important object for the tourist application. The TouristInit class initiates all other classes for a specific version of the application. The tourist application is designed so that it is easier to establish which version it is executing during community adaptation. Neither the TouristInit class, nor any class resulting from the TouristInit class, needs to be initially stored within the executing DUPE system.

The Tourist system application was designed so that, if appropriate, a central server can be located at a tourist destination. However, this is not necessary as DUPE communities allow members to gather resources from other members and, as a result, create or maintain destination specific systems.

Our scenario tested the Tourist application as a DUPE System[7], and analysed its ability to alter behaviour based on a current proximity. Moreover, each proximity the system entered contained a different (simulated) tourist destination application. The scenario showed that under such circumstances the tourist application worked efficiently and correctly whilst running within the DUPE framework, and was able to adapt according to a current community. This testing scenario was specifically used to analyse the capabilities of a target application, particularly its network capabilities, while executing as a DUPE system. Therefore, several of the tourist destination applications (those adapted by a DUPE tourist system) called for access to images and information that were located on remote HTTP servers; for example, photographs of the tourist destination. The testing during this aspect of the analysis was positive. It was seen that DUPE allowed the application to interact within the network as it would without DUPE. As a result of resource reloading, it obtained access to images and information from locations that were initially unknown.

[7]DUPE 5.0 and DUPE JPDA were used as DUPE middleware for tourist applications.

11.8.2 Observations of Adaptation Test - Localised Adaptation

During the testing scenario the execution of an incomplete target tourist program was attempted. In this case the results indicated that a DUPE system is able to load multiple classes that are all unknown to its structure. The positive observation resulting from this is that there are no restrictions on the quantity of resources a system can discover using a DUPE middleware. Moreover, these results also demonstrated that, if it is possible to determine the details of the first class that must be discovered, it is possible to execute an entire application that does not exist within a system. For example, it is possible to execute the command:

```
java ... dupe.DUPE unknownApplication.UnknownClass ...
```

and the DUPE framework will gather, if available, all details of the unknown application and execute it.

The results obtained from testing the Tourist program also indicates that the DUPE framework is as useful for complex applications as it is for simple applications. Furthermore, it also affirms that a specialised application executing as a DUPE System can act as an advanced dynamic web browser, and in a similar manner to Web Start [109]. The application is not only able to display information but also regenerate an entire application.

11.9 Technical Testing

The technical testing of the DUPE framework is based completely on our test implementations: DUPE Lite, DUPE 5.0 and DUPE JPDA. These are tested using tools that provide measurements during the execution of a system. However, some tests were not applicable to DUPE Lite due to its runtime adaptation limitation. The technical tests are designed to provide an insight into how a DUPE framework operates and the impact its operation has on the target application and device. The core intent of these measurements is to determine if the framework has an acceptable level of physical overhead, and whether the additional time it takes for a system to locate and instantiate a class is an acceptable trade-off for its distributed capabilities. Specifically, we provide the following measurements:

- **VM Overhead Measurements**: these measurements analyse the added load the DUPE framework gives to the memory and processing capacity of an executing system.
- **Application Speed Tests**: these are speed measurements that are analysed in order to determine the overall effect of DUPE in terms of application execution timing. The measurements analysed are:
  - Class and Object Loading Speeds
  - Adaptation Speed
  - Community Interaction Speed
  - Gathering Resources
  - Registering within a Community

11.10 VM Overhead Measurements

We took measurements of the size of the JVM HEAP size during execution. The same target system is executed each time, and the measurements are taken during the following three different executing environments:

- as a normal stand alone system
- as a DUPE compatible system using DUPE Lite
- as a DUPE compatible system using DUPE 5.0
- as a DUPE compatible system using DUPE JPDA.

All testing measurements were accurately obtained using the JProfile (Version 4.0.2) tool [37]. The measurements were then averaged from a large set of executions of the exact same scenario. Within these scenarios, all the measurements were obtained at three key executing points:

- during standard application processing (initial/normal adaptation)
- during adaptation[8]

[8]During execution stages such as class redefinition, it is not possible to measure the target application whilst it is executing outside a DUPE middleware or with DUPE Lite.

- post adaptation.

The results of the analysis are provided in Table 11.5 and shown graphically in Figure 11.1.

The results of the HEAP measurement testing indicate that during adaptation the HEAP size of the application rises by only a small amount, and that once an initial adaptation has occurred the HEAP size remains approximately at the higher level. Therefore, the most appropriate measurement to use for the calculation of any overhead of a DUPE framework is after system adaptation. This is because after system adaptation, the measurements represent the state the entire system is most likely to be in during most of its execution. The measurements during this period are:

- DUPE 5.0 produces on average overhead of approximately 500 MB.
- DUPE JPDA produces on average overhead of approximately 700 MB.

Furthermore, by dissecting the HEAP measurements during the testing, the analysis also revealed that the average measurement of a DynamicClass object, which may initially be seen as the largest and most variable object of the DUPE framework, has a smaller than expected average size of 24 bytes per instance. This figure can be used to indicate that due to the small size of the these objects, the actual overhead size of the DUPE framework will not alter significantly as the target application execution continues, and more classes are instantiated.

Application Setup	**Adaptation State**	**Minimum** (MB)	**Average** (MB)
Normal	*N/A (Normal)*	488	515
DUPE Lite	*N/A (Normal)*	723	756
DUPE 5.0	*Initial/Normal*	926	998
DUPE 5.0	*During*	1002	1069
DUPE 5.0	*Post*	1012	1036
DUPE JPDA	*Initial/Normal*	926	964
DUPE JPDA	*During*	1223	1260
DUPE JPDA	*Post*	1212	1250

Table 11.5: HEAP Size Results

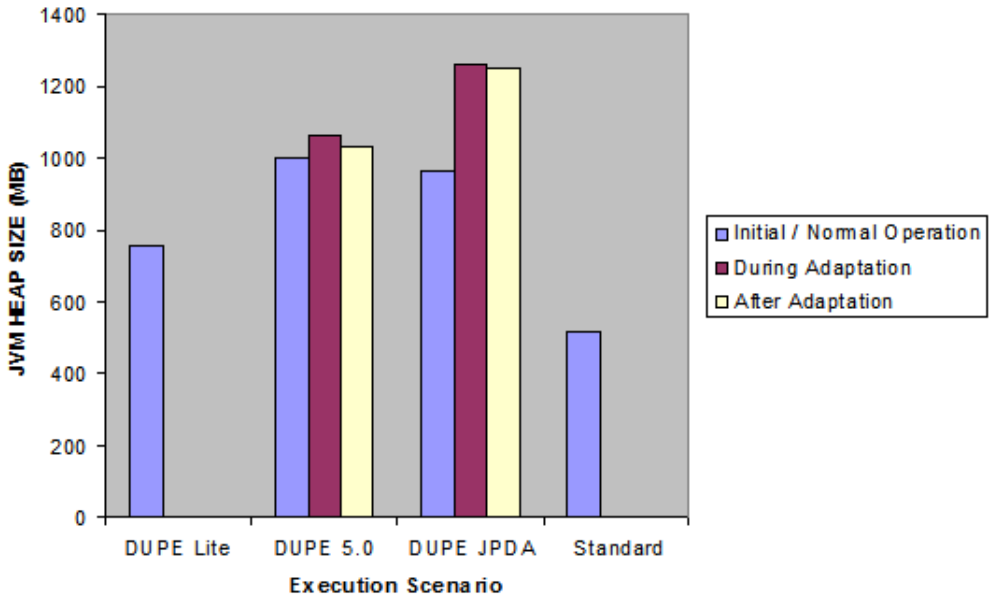


Figure 11.1: JVM HEAP Size Graphical Representation

11.11 Application Speed Tests

To understand further the impact a framework implementation has on an application, we analysed tests to determine the added time they give to system execution. The measurements are used to determine the general impact of the framework during standard operation. That is, the measurements were not taken when DUPE related processing, such as class redefinition, was being achieved. However, we also measured the impact the framework has during DUPE related processing. This measurement is used to compare the impacts of the different DUPE implementations and determine how each might execute on devices with limited capability.

11.11.1 Class and Object Loading Speeds

During the speed measurements of class object instantiation, the measurements determined to be useful are those of redefinition and object loading speed . These results provide the details to illustrate the close relationship that the DUPE implementation has with its specific dynamic updating technique. Tables 11.6 and

11.7 give the averaged timing results from 20 execution measurements on all of our randomly chosen test machines. This test is not applicable to DUPE Lite as measurements were taken during runtime adaptation.

Application Scenario	**Initial Load Speed** (ms)	**Subsequent Loading Speed** (ms)
Standard Execution	0 - 10	0
DUPE 5.0 Execution	78 - 94	0
DUPE JPDA Execution	2656 - 2828	0 - 16

Table 11.6: Loading Class Bytes and Object Instantiation - Small Class

Application Scenario	**Initial Load Speed** (ms)	**Subsequent Loading Speed** (ms)
Standard Execution	0 - 130	0
DUPE 5.0 Execution	78 - 103	0
DUPE JPDA Execution	2969 - 3921	0 - 16

Table 11.7: Loading Class Bytes and Object Instantiation - Large Class

It is easy to see from these results that the DUPE JPDA implementation, as can be predicted from Chapter 10, requires a much greater amount of processing time in comparison to the other executing scenarios. From this we can suggest that DUPE 5.0, in terms of execution speed, is best suited for mobile devices.

11.11.2 Adaptation Speed

There are two different aspects of the adaptation speed that are measured. First, the measurements of aspects directly affected by the community interaction speeds, and second, the class updating speed the result from the updating technique applied to the framework implementation. In Table 11.8, we show the measurements for the latter. We provide all aspects of community interaction speed testing in Section 11.11.3. For the same reason as the previous tests, these tests are not applicable to DUPE Lite.

As all classes vary in size, we present the loading speed from a variety of different sized classes. This allows the testing to give a realistic indication as to the speed

Implementation	**Min**(ms)	**Average**(ms)	**Max**(ms)
DUPE 5.0	16	40	130
DUPE JPDA	134	682	2469

Table 11.8: Reloading Class File Measurements

of each execution. Moreover, as we provide a minimum, average and maximum test value, it may also be seen that a small sized class will load at (approximately) the minimum speed, whilst a larger than normal sized class is likely to load at the maximum rate.

The analysis of the speed measurements further supports the view that, as indicated by Sun [106], J2SE 5.0 improves immensely on the instrumentation time of the previous technique J2SE 1.4 using the JPDA structure. However, they also indicate that neither implementation of the framework inhibits the speed of a system's execution to a degree that system use is affected. And, although DUPE JPDA does create a slower application, it may be the case that a user will conclude that its extended capabilities outweigh any loss of execution speed.

11.11.3 Community Interaction Speed

The results of the community interaction time, gathering class data, analysing Dupe Service Objects and discovering DUPE communities were predictable. This is a result of the exact nature of this section of the framework. For example, when a DUPE system is gathering resource details, the variables that play a part in determining the community interaction time are:

1. the *size of the class bytes* being gathered
2. the *speed of the delivery system*, and
3. the *active state of reloading and discovery mechanism (loaded state within the JVM).*

We found that, for example, gathering the same resource from two different systems, gave slower testing times when the testing system was busier. However, these results do not come as a surprise and are common measurements of distributed applications.

We will now provide further analysis of resource gathering and system adaptation.

11.11.4 Gathering Resource Testing Results

The gathering of resource details is measured as a single timing entry which takes into account the period of time it takes to:

- find a DSO
- search through all the resource details within the DSO, and
- determine their usefulness for system adaptation.

It is reasonable to suggest that classes which may be used for adaptation by an application may produce a time lapse. These tests aim to measure this overhead. Specifically, a major aim of these tests, is to determine whether searching through unusable resources presents unacceptable lag delays for DUPE systems.

As we did with the JVM analysis in Section 11.9, we have analysed both ends of the extreme for a class file size (*small* and *large*) along with measuring the average speeds of standard, *miscellaneous*, application classes[9]. All times the results from the analysis of all possible different situations within the test environment are averaged to determine an over-all time.

The results are divided into three categories:

Resource Unusable: the discovery of a resource that will never be applicable to the target application.

Known Resource - Tested Unusable: a resource that may be used within the target system, yet, due to adaptation settings or security constraints will not be used.

Known Resource - Tested Usable: a resource that is known within the target application and can (will be) loaded or reloaded accordingly[10].

Each category is identified clearly as a separate timing entity during tests, and as a consequence, if viewed as a single measurement the results are misleading. The results of the testing are presented in Table 11.9 for ease of comparison.

[9]The standard miscellaneous application classes are those that we have created for our analysis in Section 11.3.

[10]The timing of a known and usable resource does not include the period of time taken to load and define the class.

Implementation	**Resource Unknown** (ms)	**Known Tested Unusable** (ms)	**Known Tested Usable** (ms)
DUPE Lite	0	0 - 230	10(min), 6572(avg), 20163(max)
DUPE 5.0	0	0 - 310	15(min), 6719(avg), 23213(max)
DUPE JPDA	0 - 16	0 - 400	127(min), 8609(avg), 23046(max)

Table 11.9: Gathering and Analysing Resource Details

A foreseeable concern of the DUPE framework is that unnecessary CPU usage time may be allocated to tasks which have no direct impact on the target application. One such task is the discovery of a resource which is deemed immediately unusable; for example, an application that will never make use of classes from the `helloWorld` package. This application, if set as DUPE compatible, will, however, discover all resources present in the same DUPE community, including those from `helloWorld`. The analysis of these resources should not hinder the progress of the application in any way.

The timing results obtained during this analysis show that the effect that the discovery of an unnecessary resource has on the target application is minuscule and in most cases immeasurable, hence a measurement of zero micro seconds. These results are positive. They indicate that unnecessary computations that occur during DUPE community interaction have minimal, if any, impact on the processing of a DUPE system.

11.11.5 Registering within a Community

To gain an overall view of the timing of a DUPE system, we analysed the period of time it takes a DUPE member to discover and register within a community. The speed of this aspect of the framework will determine, along within other factors such as resource transfer time, how long a system needs to be within the proximity of a community before it is able to benefit from interaction.

Implementation	**Discovery and Registration** (ms)
DUPE Lite	940
DUPE 5.0	1004
DUPE JPDA	6824
Jini 1.1 HelloWorld	240

Table 11.10: Community Discovery and Registration Average Speeds

Table 11.10 gives the average discovery speeds of entering DUPE members recorded over many different executions of each of our test frameworks. In order to show how the framework has affected these speeds, we have also included the measurements of a simple HelloWorld Jini 1.1 application. As DUPE makes use of Jini for discovery purposes, we can view this timing as the overhead created by the DSO and DUPE's gateway proxy structure.

The overhead of the discovery time corresponds to the overhead in general of the middleware. DUPE JPDA takes longer to discover the communities, adding approximately 6 seconds to the standard Jini discovery time, while, DUPE 5.0 and DUPE Lite add only about 0.5 of a second to the same time. We also note that the capabilities of the host system and the setup of the network always have an impact on the discovery time. However, this is standard for all distributed applications.

11.12 Other Observations

The testing procedures applied during this chapter showed DUPE to be an effective means of component exchange. And, as is evident from our technical evaluation, all implementations could possibly be used within mobile devices.

During our observations of the DUPE testing, we encountered some instances where unexpected results were obtained. These instances were interesting to observe as, in general, they were a result of the unknown limitations of each dynamic updating technique. The most problematic of these results were found during post-redefinition observations.

Specifically, we note that Java Swing components, `javax.swing`, in both DUPE 5.0 and DUPE JPDA displayed irregular performance. At times a main component of a Swing GUI, for example `javax.swing.JFrame`, would produce a new GUI Frame on the screen and leave the older versions visible. This problem is, however, not something we see as detrimental to the framework design, and is most probably due to a combination of repaint problems and Stack pointer variables. Moreover, problems such as these are obviously related to the dynamic updating within each respective JVM, and therefore, not a direct result of the DUPE framework.

11.13 Summary

Our test DUPE implementations all provide the functionality of the DUPE framework (although DUPE Lite has limited functionality). Not only are they able to generate and cooperate within DUPE communities, our tests show that the DUPE communication language design succeeds in allowing different applications, using different DUPE middleware, to share resources. The analysis in this chapter also demonstrates that, in comparison to DUPE JPDA and DUPE Lite, DUPE 5.0 is far more suited to mobile devices. DUPE 5.0 provides complete DUPE compatibility, maintains faster execution speeds, creates the least amount of overhead for target systems, and as a result, consists of a small system footprint. However, this is not to say that, for example, DUPE JPDA is of no use; for instance, J2SE 1.4 is currently the most deployed JVM, and subsequently, DUPE JPDA is the most applicable technique for most current systems.

The analysis provided in this chapter is a framework analysis. It focuses on the fundamental aspects of DUPE, and looks at the middleware implementations in order to assess the added overhead given to systems. It weighs this up against the added benefits of community interaction. However, as the concept of a DUPE community is new, the analysis we have provided can only be used to determine whether the DUPE middleware inhibits the execution of a system to a degree that it is no longer beneficial.

In order to provide further indication of the possible benefits of DUPE, the following chapter provides models of the expected movements of gypsy agents. These models are used to determine the usefulness of DUPE as a software transfer and deployment technique. The models also provide us with the means to discuss how gypsy agents provide an alternative and unique means of system evolution.

Chapter 12

Model Analysis

12.1 Introduction

This chapter provides an analysis of gypsy agents. We achieve this by developing mathematical models based on the movements of these agents among DUPE communities. We provide the analysis of both a simple model and complex models. The simple model is mathematically conclusive. However, our complex models could only be developed as a result of simulation analysis. These models are used to determine probabilities on how fast a system resource might be dispersed via gypsy agents. Then, to conclude our modeling, we develop a relationship between gypsy agent movements and a positive perspective of epidemic models.

12.2 Modeling Gypsy Agents

The movements of gypsy agents are random. Consequently, an exact measurement of their movements is unobtainable. We will instead show that with random graph modeling and probability analysis we can provide valuable insight.

There are several techniques that have been used to determine the spread of software throughout a network; for example, the modeling of web community crawling [27], computer virus progression [57], and AI techniques, such as, flocking and spawning [90]. Each of these examples is a measurement of the movements of a semi-predictable software system. Gypsy agents are physical mobile agents which are governed by human movements. This element must be taken into account

and consequently the modeling is different. Accordingly, we identify biological epidemiology scenarios and modeling as useful. The relationship among these models and computer viruses has been addressed [57], however, their use in combination with random graphs for software transfer models has not. We aim to show that this type of modeling best represents the nature of gypsy agents, and is the most appropriate means of determining their distribution attributes. Thus finalising their contributions to the field.

During Sections 4.4.3 and 6.9, we argued that the movements of gypsy agents may be similar to that of the spread of infectious disease [78]; both are a result of human interaction. In modeling and determining the probability of the spread disease, the use of random graphs [15, 78] can be applied in a similar manner to that used to determine the evolution of networks [33]. However, such infectious disease models are extremely complex; for example, those applied by Newman, Strogatz and Watts [78]. We do not require such an in-depth analysis. The aim of this section of our work is not to discover a new means of mathematical modeling, we only wish to use it to analyse the usefulness of the gypsy agent concept. That is, the contribution of gypsy agents to the thesis is the concept itself. We use modeling analysis as a measurement and proof of concept. However, in Chapter 13, we do suggest that more complex modeling may be developed as a future contribution to the field.

We will present the usefulness of gypsy agents in the form of an evaluation of the probability that their movements are active enough to spread a software component version through communities at a reasonable rate. Firstly, however, we will briefly discuss other current methods of software transfer. We will then apply a simple line model to obtain very general results. We will follow this with more realistic complex models. Although, as will be discussed, most comparisons between other techniques and our models are not appropriate due to differences in analysis.

12.3 Techniques for System Evolution

The distribution of system updates is most commonly achieved by depositing the necessary update components on web based servers and making them available for users to download. There are different ways to use this technique. For example, the user's system may be automatically timed to search for an update. At times, in this scenario, the user is unaware of their system's updating procedures; for

example the Windows operating system is set (as default) to search for updates at periodic time intervals, unbeknown to the user. However, these setting can always be altered at the users discretion. A consequence of this technique is that a high degree of user interaction is required for system evolution. As such, there is no movement of the system update itself only its downloading by users. For example, any model for a Windows XP security update would show a high initial download that progressively declines as all users install the update. Therefore, it is reasonable to state that, the download rate of a software component is dependent on the needs and wants of all users. Consequently, any subsequent model is specific to the software component, and not measurable against the more general use of gypsy agents movements.

Peer-to-Peer (P2P) sharing is direct system cooperation that results in file transfer. It allows systems to obtain data from one device directly to another device without a middle server. The connectivity between systems in P2P situations is more flexible and usable than server downloading, as the required data can be gathered from a wider range of systems. However, the system update procedure is still restricted by the need for user interaction. Any P2P connectivity is flexible in terms of what systems it allows to communicate, consequently, there is no requirement for a system to be mobile or PC based. Therefore, it is possible that community based cooperation could be built on top. Doing this may allow systems to cooperate in a network where there is a possibility of adapting to the current state of a community, and subsequently, update system states accordingly. Although, currently, this seems far-fetched. The modeling of P2P interaction by Zhong, Shen and Seiferas [124], is similar to ours. And, although their model is not based on community cooperation or dynamic systems, it does demonstrate the interaction of P2P applications. However, their results, and the reason for their results, are different to ours. Consequently, it would be unreasonable to compare their results with ours.

Gypsy agents can carry a new updated version of any system with them and distribute it amongst users. Although, comparisons are minimal, what we can determine without any comparative measurement is that gypsy agents encourage system evolution according to a chosen variable, for example location, without any user intervention. This is unique in comparison to both web downloads and P2P. What we need to determine is, whether the distribution is at a relatively reliable rate.

In response to this query, we will now begin our model analysis for gypsy agents.

12.4 Basic Line Model

The scenarios presented throughout this chapter model multiple gypsy agents. At each generation stage, one agent will move from each community to one of its neighbouring communities. Although this movement will occur if an agent is infected or not, we are only interested in the movements of the infected agents.

This is the most basic of our models. Its simplicity is shown in Figure 12.1 (communities are shown as nodes, and infected communities are shown using dark circles; all others circles represent non-infected communities).

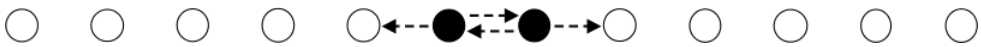


Figure 12.1: Straight Line Movements of Gypsy Agents

In this scenario any contiguous set of infected points will change in one of 3 ways:

- 2 more nodes get infected by the endpoints moving out
- 1 more gets infected by the endpoints both moving right
- 1 more gets infected by the endpoints both moving left
- no more get infected by both moving in.

These movements are shown in Figure 12.1 as arrows. The probabilities of each contiguous set are all equal: 1/4.

In general, the probability that N nodes are infected at a generation[1] n equals:

> probability that N are infected at $n-1$ times probability that no more get infected
>
> *plus*

[1]We refer to a step in time as a generation according to common epidemiology terminology.

probability that $N-1$ are infected at $n-1$ times probability that one more gets infected

plus

probability that $N-2$ are infected at $n-1$ times probability that two more get infected.

That is:

$$P_n(N) = \frac{P_{n-1}(N)}{4} + \frac{P_{n-1}(N-1)}{2} + \frac{P_{n-1}(N-2)}{2}$$

except in the following special cases:

$$P_1(1) = 1$$
$$P_2(2) = 1$$

From this we can find that:

$$P_3(4) = P_2(2)/4 = 1/4$$
$$P_3(3) = P_2(2)/2 = 2/4$$
$$P_3(2) = P_2(2)/4 = 1/4$$
$$P_4(6) = P_3(4)/4 = 1/16$$
$$P_4(5) = P_3(4)/2 + P_3(3)/4 = 4/16$$
$$P_4(4) = P_3(4)/4 + P_3(3)/2 + P_3(2)/4 = 6/16$$
$$P_4(3) = P_3(3)/4 + P_3(2)/2 = 4/16$$
$$P_4(2) = P_3(2)/4 = 1/16$$

and so on. Interestingly, the numerators form the pattern:

1
1 2 1
1 4 6 4 1
1 6 15 20 15 6 1

On observation it is simple to see that this represents every 2nd row of Pascal's triangle. And that the denominators are powers of 4.

The actual formula depends on when you start time. If you start at $t = 1$ (as in the above), then the probability that N communities are infected at time n is:

$$P_n(N) = \begin{pmatrix} 2n-2 \\ 2n-N \end{pmatrix} /4^{n-1}$$

Prove by induction.

Everything else (maximum number infected, average number infected, and so on) all follow from properties of Pascal's triangle.

To further verify this model we ran a simulation analysis

12.4.1 Simulation Results

The simulation runs through 2000 random generations of gypsy movements (this number of runs remains for all simulation throughout this chapter). The results from the simulation will now be compared against those from the model.

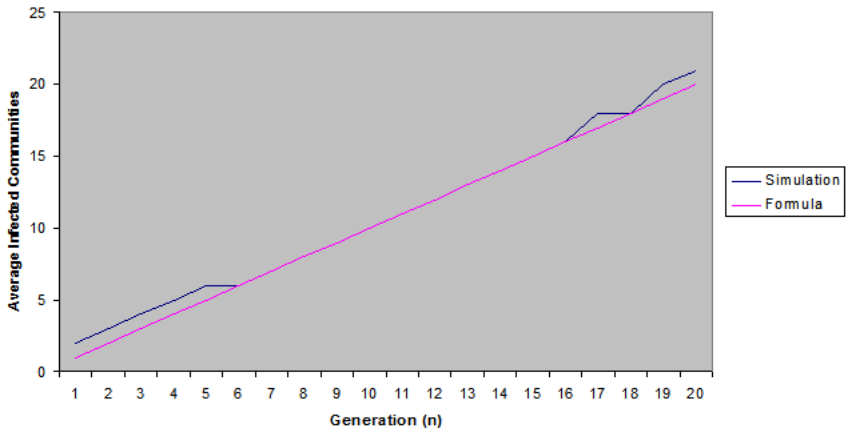


Figure 12.2: Line: Average Number of Infected Communities

The medium rate of distribution comparison is shown in Figure 12.2. The results match. Furthermore, the slope of the line indicates that the rate of distribution of gypsy agents is useful for component transfer.

For further descriptive analysis we look at distribution curves. This analysis details the predictability of gypsy distribution by calculating the probability of N infected communities at a selected generation n ($P_n(N)$). Again, the results are

positive. We provide the results for $n = 10$ and $n = 20$ (Figures 12.3 and 12.4 respectively) to show the comparison of the model results against our.

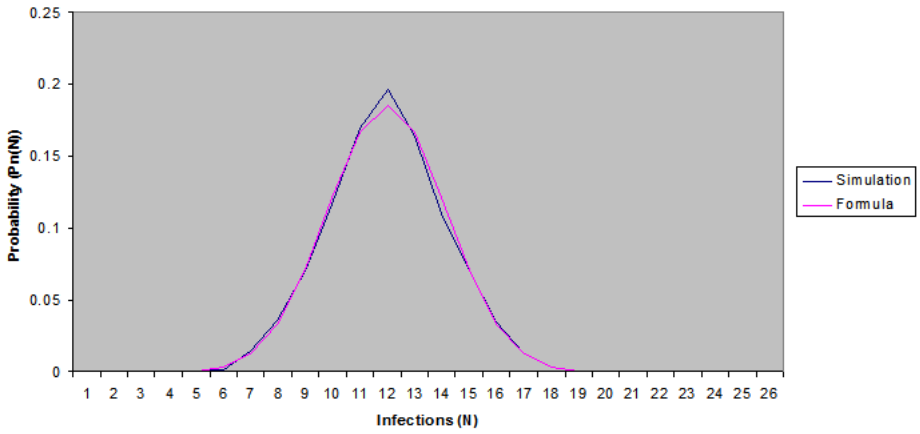


Figure 12.3: Line: Histogram Distribution at $n = 10$

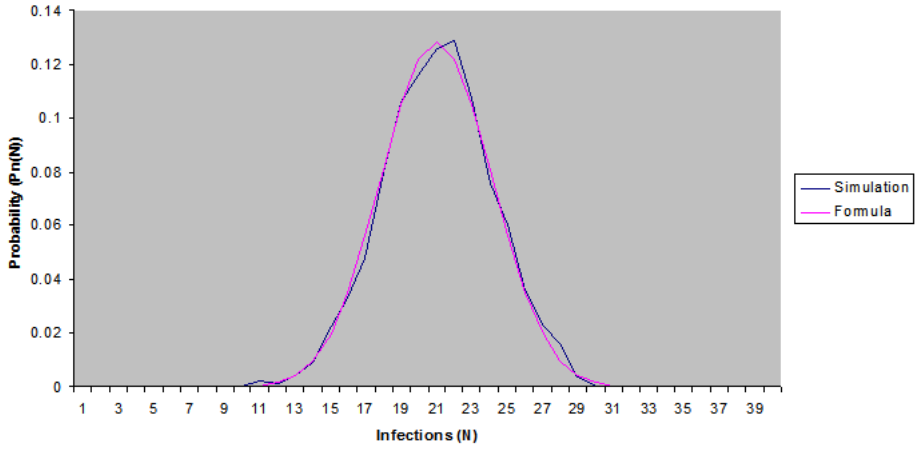


Figure 12.4: Line: Histogram Distribution at $n = 20$

12.5 Complex Models

Although the basic line model provides an insight into the movements of gypsy agents, it is unrealistic. A more realistic arrangement of communities is obscure.

In reality DUPE communities will never be arranged according to a particular predetermined pattern. The introduction of complex model patterns may provide valuable details on gypsy agent movements. Such analysis would produce results similar to, for example, the use of complex webs models for the community growth, in terms of internet relationships, adopted by Dorogovtsev and Mendes [33]. Their models were based on web page links and pages using random graphs for their analytical conclusions. We will apply a similar scenario technique to our complex modeling.

The literature provides many results on complex random movements that are applicable to our work. For example, the recent work of Burioni and Cassi [19] provides an overview of random walk modeling. Their work includes the analysis of a random walk through different complex graph arrangements. This common random graph analysis is applicable to our work.

However, overly complex arrangements of DUPE communities are unnecessary. Instead, we use four complex, shape based community arrangements: *square*, *triangle*, *diagonal square* and *hexagon*. These arrangements were chosen, making modeling simpler, for they are the branching factors of 4, 6, 8 and 3 respectively. The shapes are illustrated in Figure 12.5 and the branching factor can be seen in Figure 12.6.

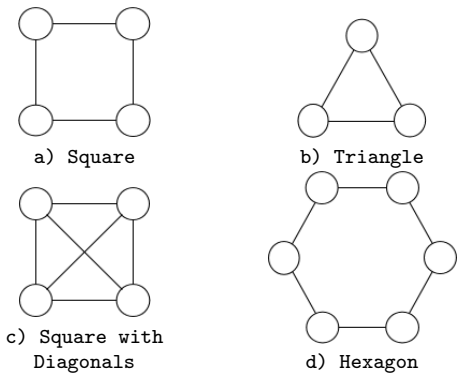


Figure 12.5: Shape Based Complex Mode Designs

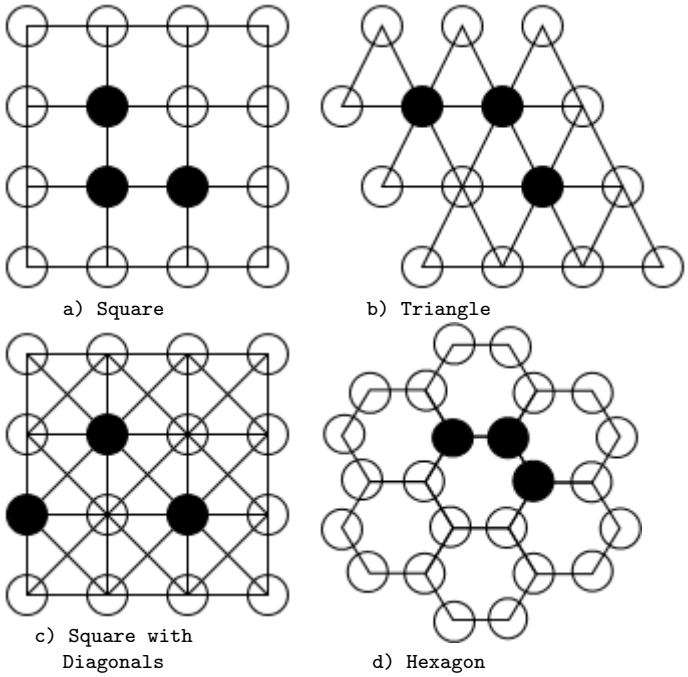


Figure 12.6: A Random Walk through Complex Models

Within these graph patterns, there are two types of movements that are appropriate to gypsy movements. These are:

1. a traveling salesman (a planned, non-random, walk that passes through all communities), and

2. a random walk.

However, considering that gypsy agent movements are a direct result of human movements, the random walk is most appropriate. Figure 12.6 provides an example of a random walk after three generations (contacted communities are shown using blacked nodes). This random walk directly represents the movements of a single gypsy agent.

Models for traveling salesmen, and random walks, are covered intensely in the literature [15, 19, 33, 78]. Standard results from these areas are applicable to our work, therefore, we will not reiterate them. We focus our modeling on the random movements of many gypsy agents, as this is more appropriate for our purposes, and, according to our research, it is also less prevalent in the literature.

Moreover, while similarities to epidemic models are present, we will show that the epidemic models are not exactly the same as our models.

We will now provide an individual analysis of each model. The analysis results are derived from simulations. Although it is possible to write a program to calculate the exact model, such as that provided for the line model (see Section 12.4), we found that such programs required system capabilities beyond those of a standard PC, and thus, exceeding the capabilities of the systems we had access to. As such, the complete execution of such programs, and subsequent analysis, is noted as future work (see Section 13.1).

The process for developing a model is repetitive for all models. In the latter models several steps of the modeling process will therefore be skipped. However, it is simple to decipher their complete modeling process using our first complex model, the square model (see Section 12.5.1).

It is important to note that terms related to 'infections', such as infects, are used in positive context; for example, it is good for a bee to *infect* a flower with pollen from another.

12.5.1 Square Model

The first of the random movement models is based on a square node graph. This is the most common use of random graphs. The number of possibly infected nodes N can be determined according to generation step n:

$$N(n) = 2n(n+1)$$

We ran a simulation to determine the movements of gypsy agents. Each simulation was run 2000 times. The results of the simulations produced two main graph types. Firstly, we established a medium curve. This best represents the expected number of infected communities at each generation. The results of this part of the square simulation are provided in Figure 12.7.

The results are simple. They show a steady progression that approximately follows a curve of $0.5156n^2 - 1.1566n + 3.8132$ according to n generation.

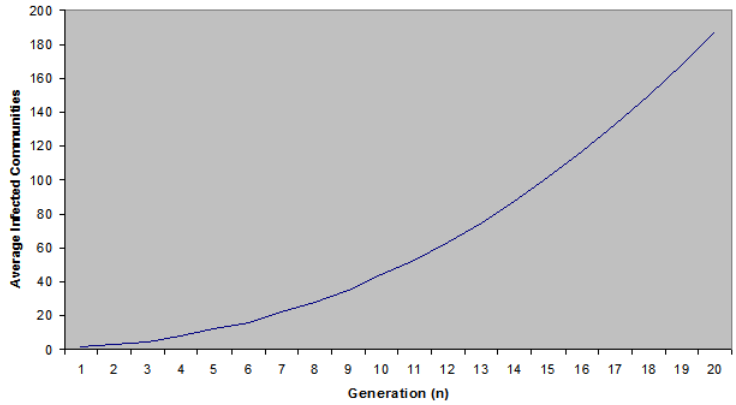


Figure 12.7: Square: Average Number of Infected Communities

The second of the graphed results are probability distribution. These indicate the probability of N infected communities ($P_n(N)$) after n generations. We provide the results for generation 10 (Figure 12.8) and generation 20 (Figure 12.9).

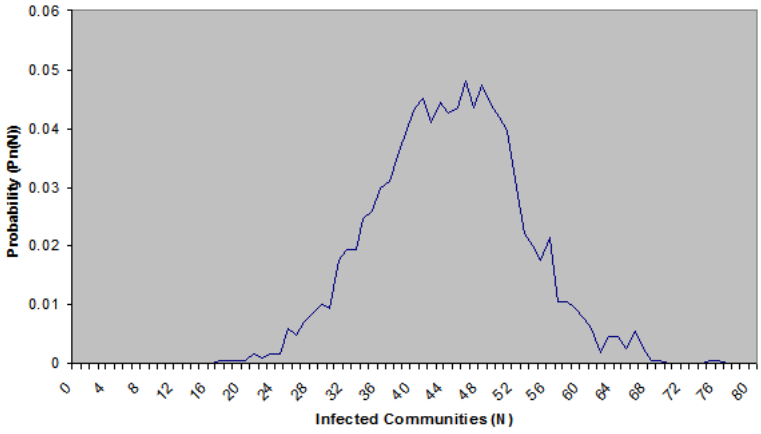


Figure 12.8: Square: Histogram Distribution at $n = 10$

These graphs indicate that the distribution presents a bell curve. This is indicative to the results from the line simulation (see Section 12.4), this is positive.

The distribution analysis provides an insight into the probability of the spread of infection as generations progress. It indicates that lower generations have a

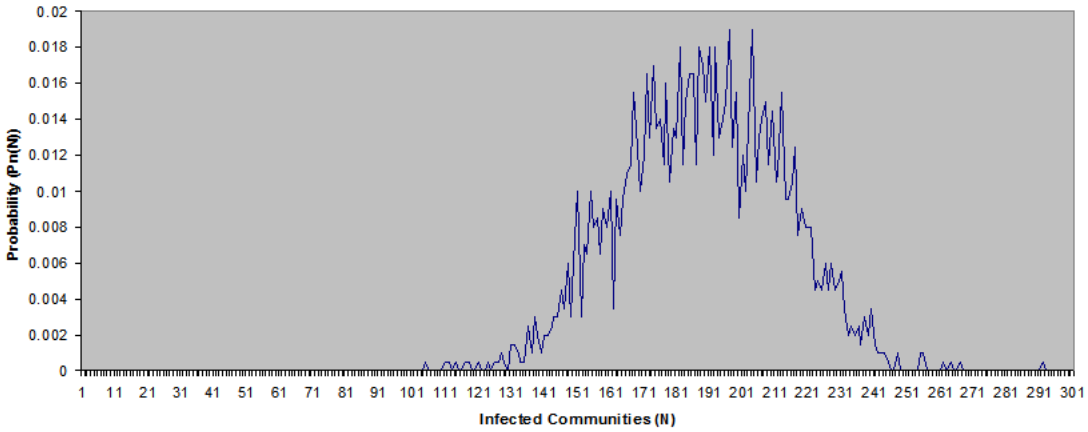


Figure 12.9: Square: Histogram Distribution at $n = 20$

predictable infection. For example, in Figure 12.8 the curve remains close to the medium. Whereas the larger generations, such as that shown in Figure 12.9, have a wider curve. Consequently, lower generations are more predictable than larger generations. This is common for such a distribution.

We now analyse the simulation in order to show an approximate of an expected distribution ratio according to time n ($Ed(n)$). A simple equation is:

$$Ed(n) = \frac{Infected\ Mean\ at\ Generation\ n}{Possible\ Communities\ at\ Generation\ n}$$

Therefore, the square model the ratio $Ed(n)$ is:

$$Ed(n) = \frac{0.5156n^2 - 1.1566n + 3.8132}{2n(n+1)}$$

This equation is plotted in Figure 12.10.

The results of this analysis show that, although the infection rate peaks at the start, it levels off to a gradual rise. The degree of this rise, although low, is enough to indicate that gypsy agents move continually into uninfected communities.

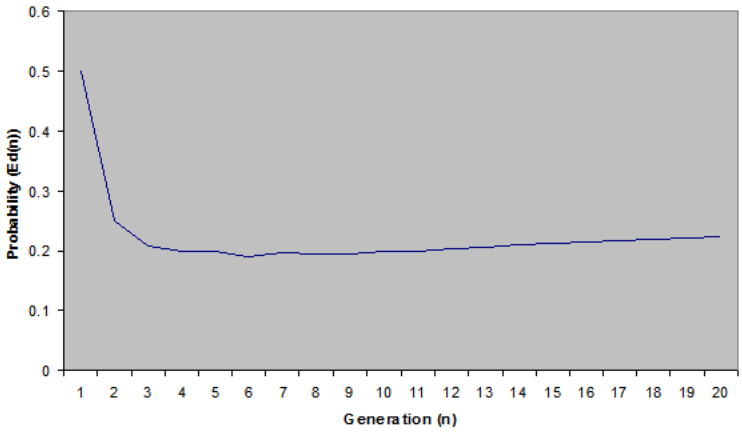


Figure 12.10: Square: Generation n Infected/Possible ($Ed(n)$)

Assuming this equation holds for large numbers, asymptotically, as $n \to \infty$ $Ed(n) \sim 0.2578$.

We will now provided the same analysis for each of our complex models. However, we will not elaborate on them. It will be obvious to the reader that the comments provided for this model hold true for all of our complex models.

12.5.2 Triangle Model

The number of possibly infected nodes of the graph N at time n is:

$$N(n) = 3n^2 + 3n + 1$$

The medium distribution analysis, resulting for 2000 simulations, is provided in Figure 12.11. This approximately follows a curve of $0.6098n^2 - 1.5291n + 4.3044$.

Distribution probability ($P_n(N)$) curve examples are provided in Figure 12.12 ($n = 10$) and Figure 12.13 ($n = 20$).

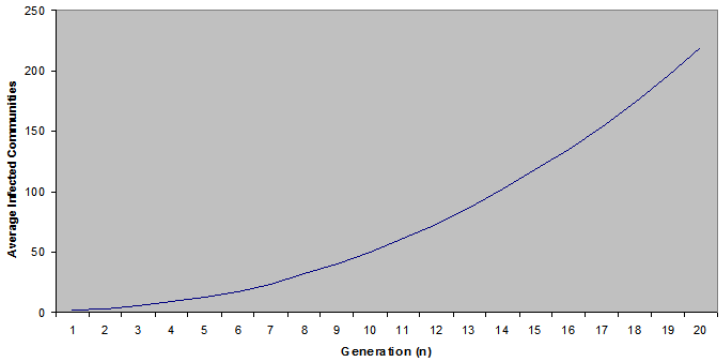


Figure 12.11: Triangle: Average Number of Infected Communities

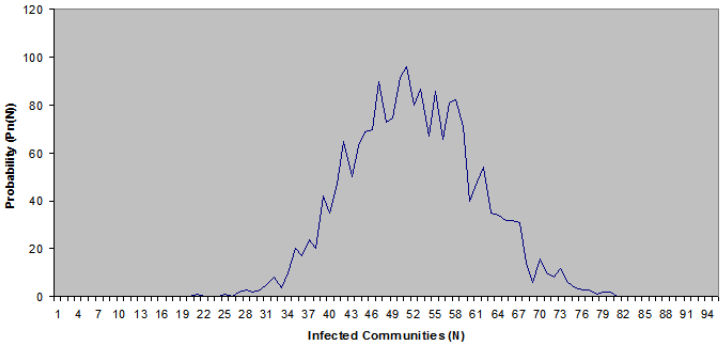


Figure 12.12: Triangle: Histogram Distribution at $n = 10$

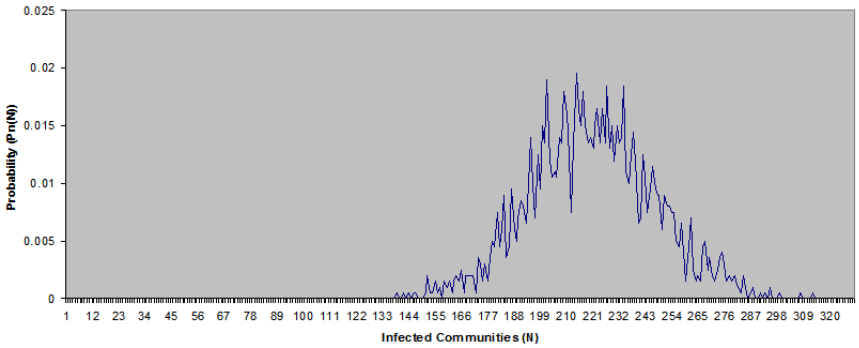


Figure 12.13: Triangle: Histogram Distribution at $n = 20$

The expected distribution ratio ($Ed(n)$) formula is:

$$Ed(n) = \frac{0.6098n^2 - 1.5291n + 4.3044}{3n^2 + 3n + 1}$$

This equation is plotted in Figure 12.14, asymptotically, as $n \to \infty$ $Ed(n) \sim 0.2032$.

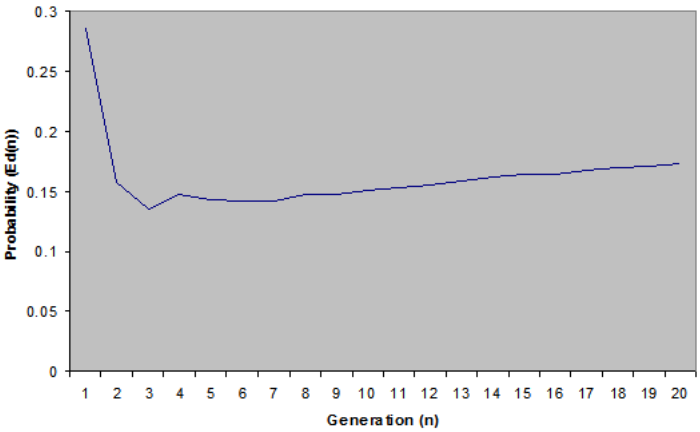


Figure 12.14: Triangle: Generation n Infected/Possible ($Ed(n)$)

12.5.3 Square Diagonal Model

The number of possibly infected nodes of the graph N at time n is:

$$N(n) = 4n^2 = (2n)^2$$

The medium distribution analysis, resulting for 2000 simulations, is provided in Figure 12.15. This approximately follows a curve of $0.847n^2 - 2.7705n + 6.1456$.

Distribution probability ($P_n(N)$) curve examples are provided in Figure 12.16 ($n = 10$) and Figure 12.17 ($n = 20$).

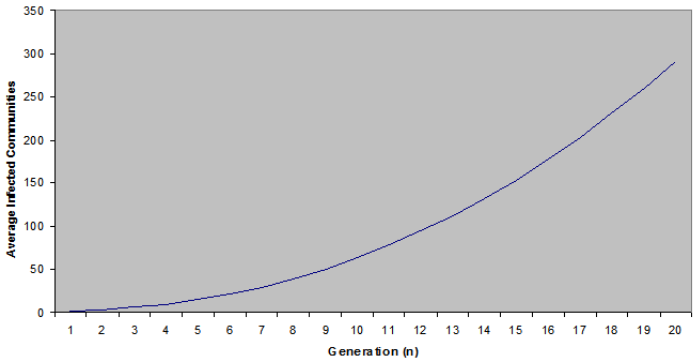


Figure 12.15: Square Diagonal: Average Number of Infected Communities

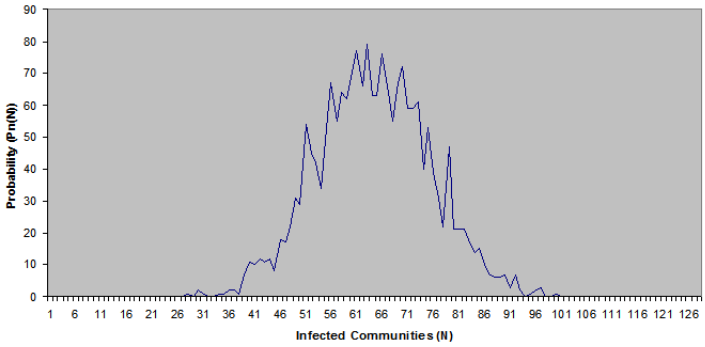


Figure 12.16: Square Diagonal: Histogram Distribution at $n = 10$

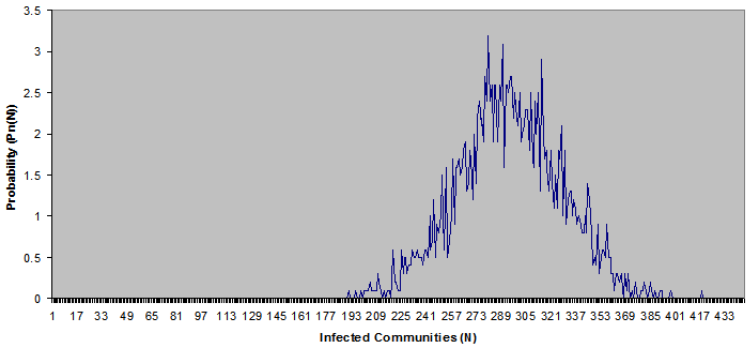


Figure 12.17: Square Diagonal: Histogram Distribution at $n = 20$

The expected distribution ratio ($Ed(n)$) formula is:

$$Ed(n) = \frac{0.847n^2 - 2.7705n + 6.1456}{(2n)^2}$$

This equation is plotted in Figure 12.18. Asymptotically, as $n \to \infty$ $Ed(n) \sim 0.4235$.

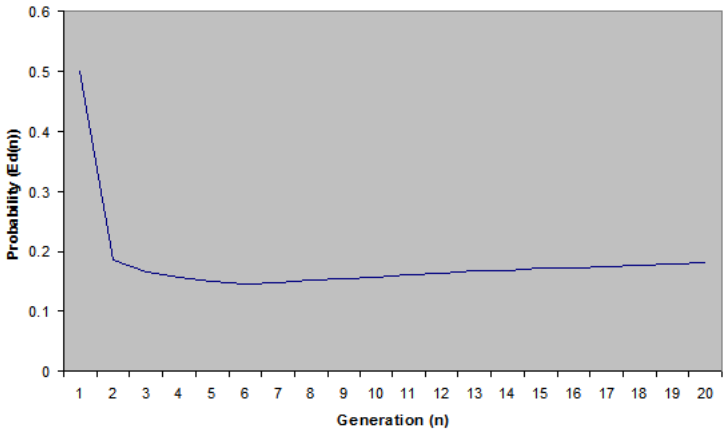


Figure 12.18: Square Diagonal: Generation n Infected/Possible ($Ed(n)$)

12.5.4 Hexagon Model

The number of possibly infected nodes of the graph N at time n is:

$$N(n) = 1.5n^2 + 1.5n + 1$$

The medium distribution analysis, resulting for 2000 simulations, is provided in Figure 12.19. This approximately follows a curve of $0.3734n^2 - 0.565n + 3.0026$.

Distribution probability ($P_n(N)$) curve examples are provided in Figure 12.20 ($n = 10$) and Figure 12.21 ($n = 20$).

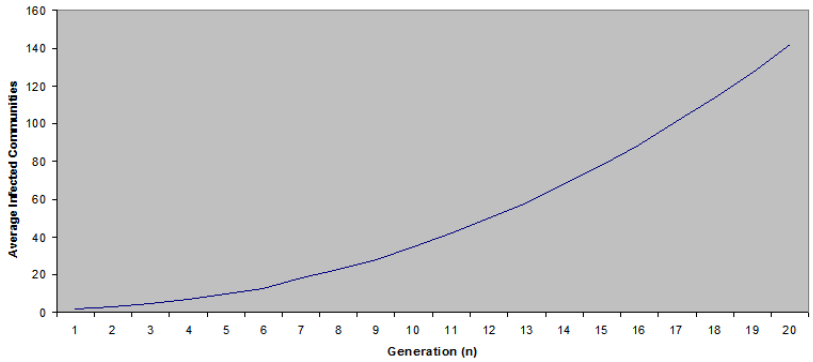


Figure 12.19: Hexagon: Average Number of Infected Communities

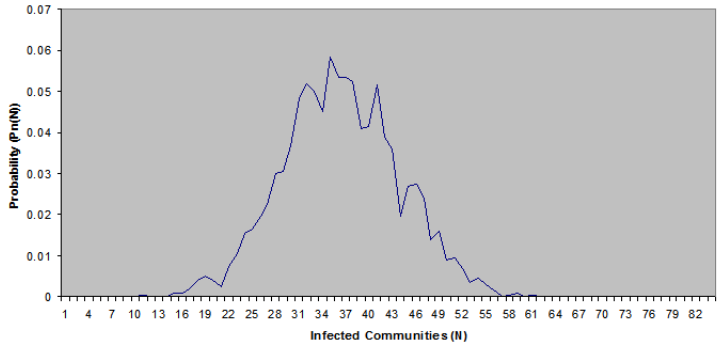


Figure 12.20: Hexagon: Histogram Distribution at $n = 10$

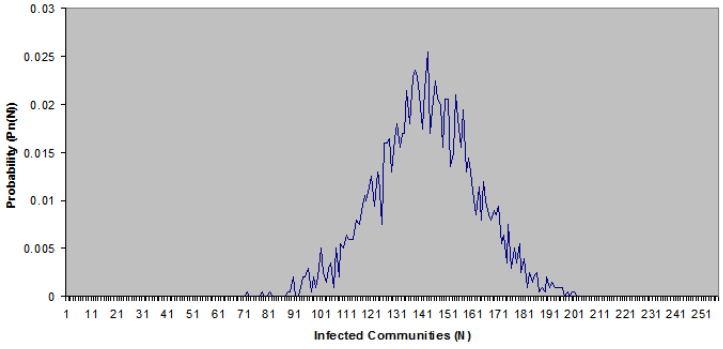


Figure 12.21: Hexagon: Histogram Distribution at $n = 20$

The expected distribution ratio ($Ed(n)$) formula is:

$$Ed(n) = \frac{0.3734n^2 - 0.565n + 3.0026}{1.5n^2 + 1.5n + 1}$$

This equation is plotted in Figure 12.22, asymptotically, as $n \to \infty$ $Ed(n) \sim 0.2489$.

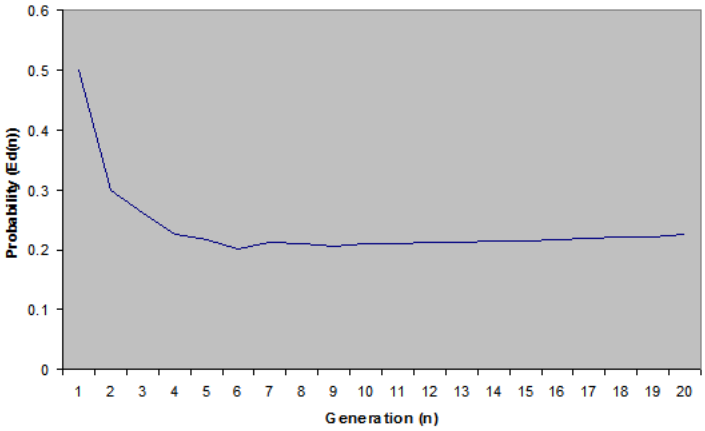


Figure 12.22: Hexagon: Generation n Infected/Possible ($Ed(n)$)

12.6 Limited Complex Community

All aspects of the previous modeling hold true for limited communities. The exception is that once the number of contactable communities reaches the community maximum, it remains there. As a result of this, the infection rate will reach an epidemic level. We will now provide results for this analysis.

We limit our total communities to a wrap around grid of 10×10 nodes, 100 communities, for all models. Figure 12.23 shows all possible paths for both the square model (red paths only) and the square diagonal model (both blue and red paths), Figure 12.24 shows the triangle model and Figure 12.25 shows the hexagon model.

An epidemic model is based on the spread of gypsy agents according to the maximum number of possible contactable communities.

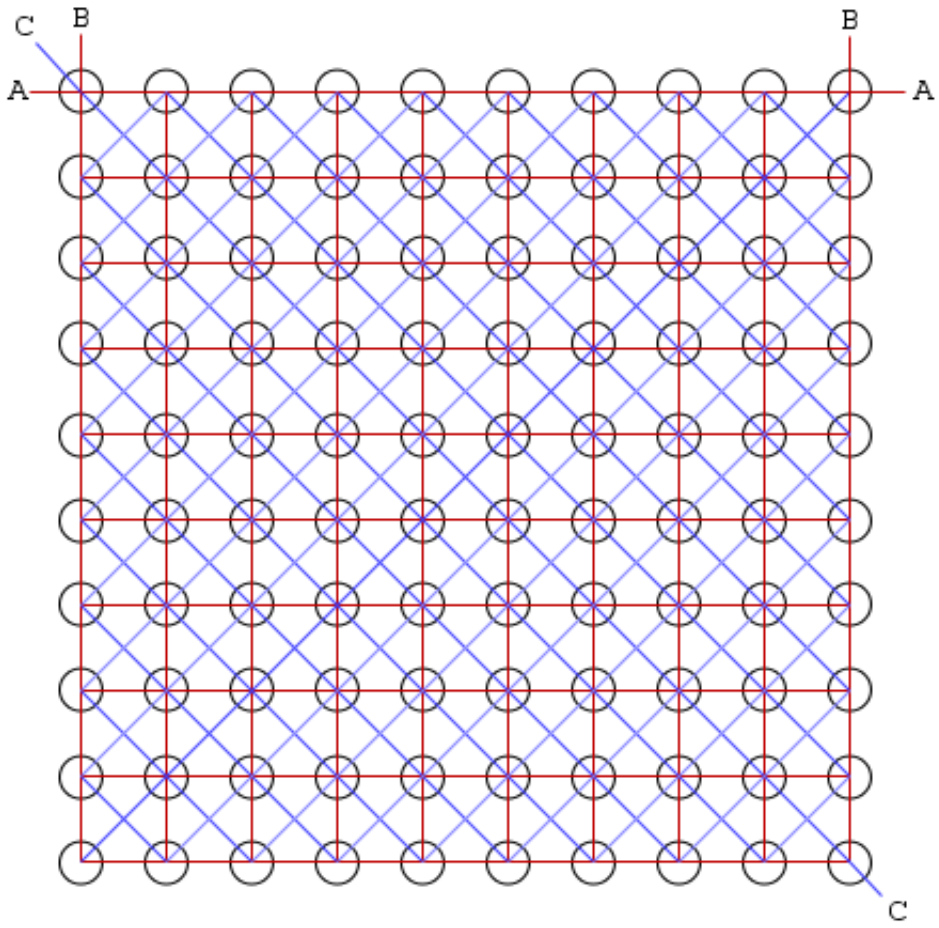


Figure 12.23: Limited Community Model - Square and Diagonal

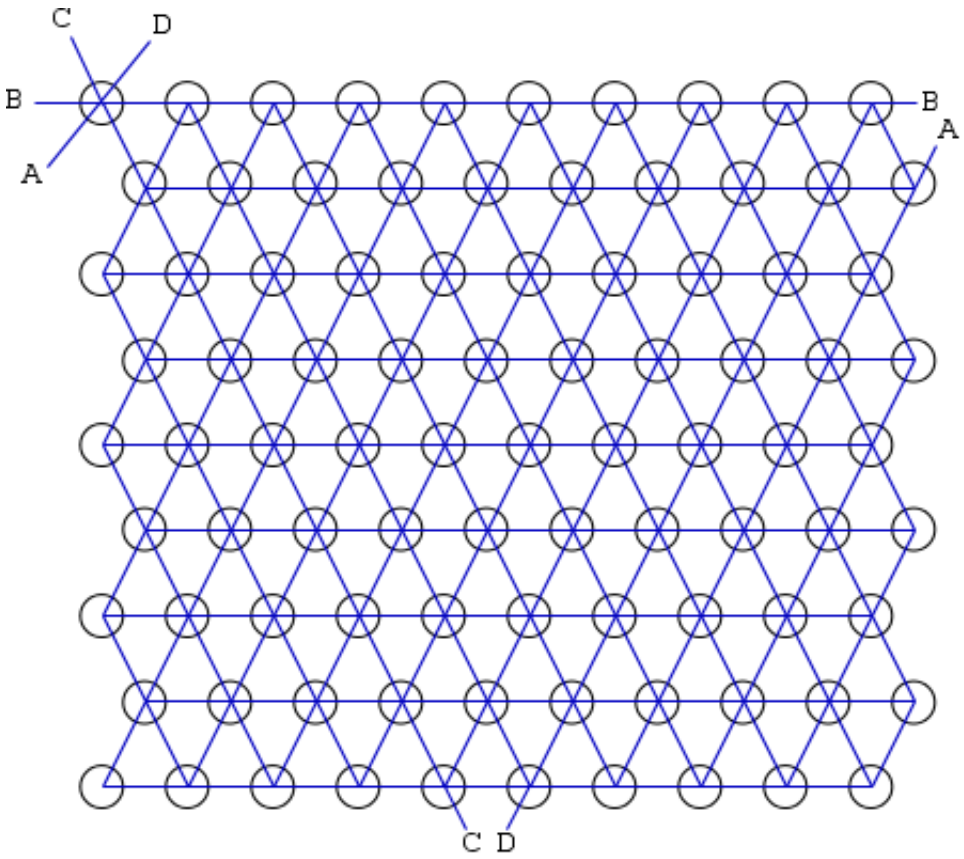


Figure 12.24: Limited Community Model - Triangle

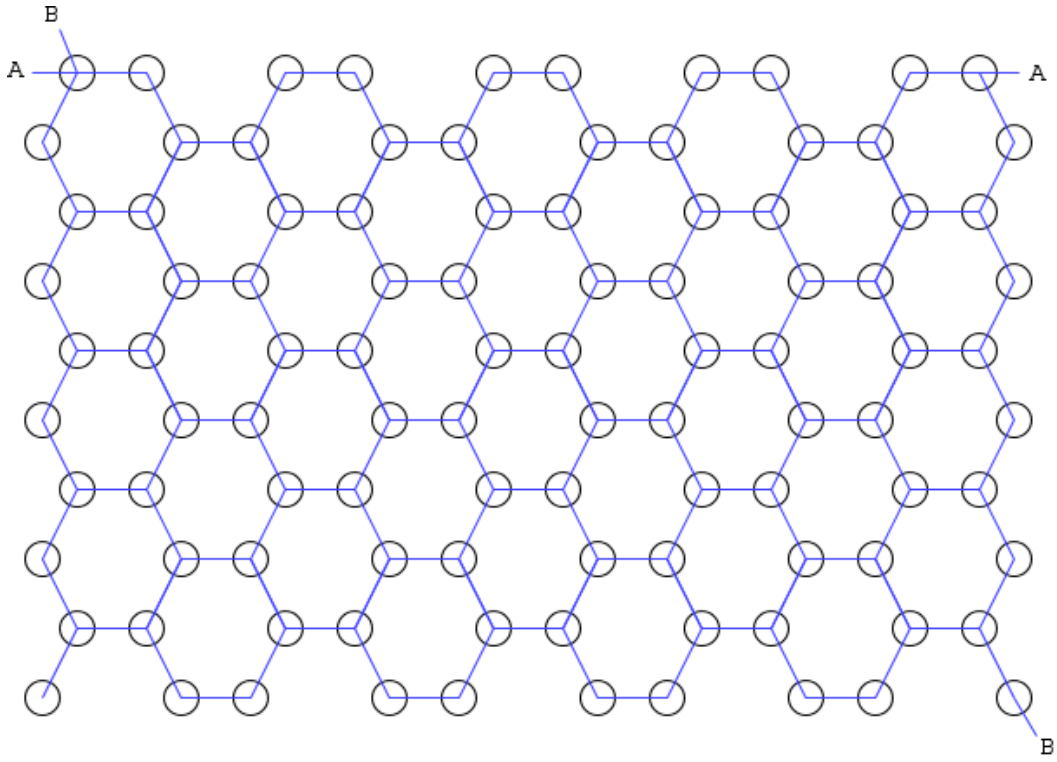


Figure 12.25: Limited Community Model - Hexagon

The results from our simulation are presented in two graphs. Figure 12.26 shows the epidemic growth line according to the total number of communities while Figure 12.27 shows the epidemic growth in relation to the number of contactable communities according to each generation.

From both graphs, we can determine that an epidemic level will be reached in all models. All complex models produce a similar epidemic curve. And, the rate at which the epidemic is reached is determined by the branching factor of the model. On analysis, it is seen that, once all communities are contactable (100 communities) the gradient to the epidemic level increases rapidly. This is common to all non-curable infection models.

12.7 Final Analysis

We presented several different models for the movements of gypsy agents. In each model, using simulation analysis, we have determined the following:

- The average infected communities according to each generation step.
  The subsequent analysis from this, finds the rate at which the gypsy agents spread a system component throughout communities.
- A histogram distribution according to a generation (two examples were provided for each model).
  The subsequent analysis from this, determines how predictable the number of infected communities is after a selected generation. The analysis showed that, in all models, as the generations increase, the distribution of infected community counts increases. However, for most generations, it is obvious that the peak of probability is equal to its mean point. This analysis, gives greater meaning to the next measurement.
- An expected distribution ratio,

$$\frac{Infected\ Mean\ at\ a\ Generation}{Possible\ Communities\ at\ a\ Generation}$$

  The subsequent analysis from this, indicates that, at each generation, the probability exists that an epidemic will be reached. This analysis showed, for all models, that during low generations the generation epidemic levels

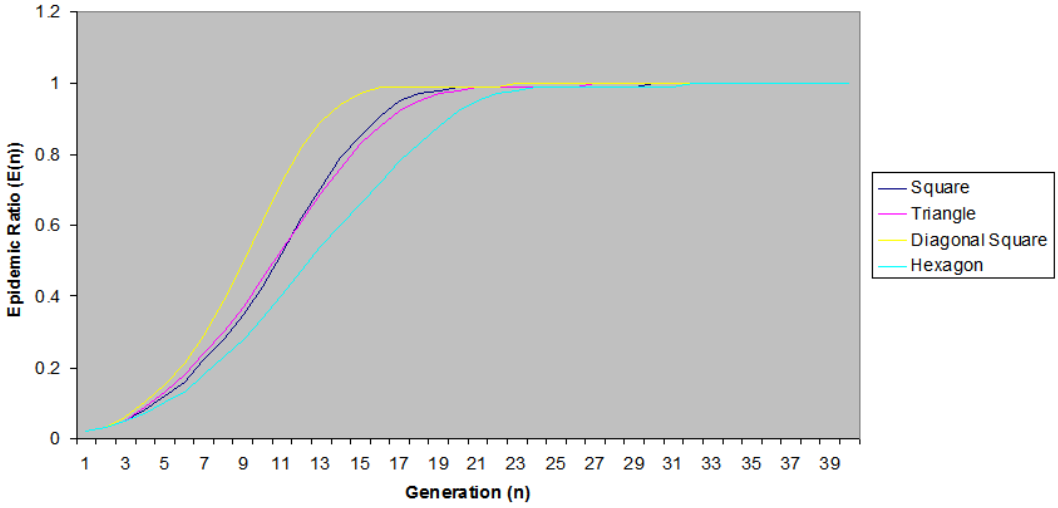


Figure 12.26: Epidemic Growth According to Total Communities

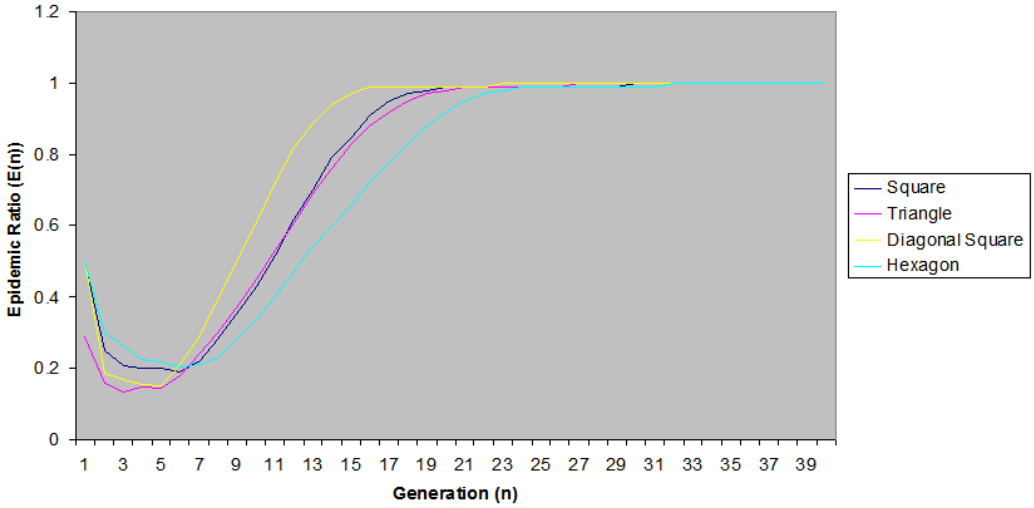


Figure 12.27: Epidemic Growth According to Generation Contactable Communities

were high in comparison to greater generations. We concluded from this that all graphs, leveled to a consistent epidemic rate.

- An analysis of limited communities (only provided for complex models). This analysis was provided to give an insight into the capabilities of gypsy agents for infecting all communities in a limited population. We limited the population to 100 communities. The results of this analysis were interesting. Each model was able to reach epidemic states, however, the rate of the epidemic growth is obviously related to the branching factor of the model.

Out of all our analyses, the limited communities presented the most interesting results. This analysis presented gypsy agents in the same context as epidemiology. We show that each complex model will reach an epidemic given a period of time, and that this time is related to the branching factor. In fact, we can deduce from all our results that, as with epidemiology, the branching factor of a model determines its rate of spread and its progression toward an epidemic. The results of our work are ordered according to this in Table 12.1.

Model	**Branches**	**Generation Epidemic Reached** (100 Communities)
Square Diagonal	8	23
Triangle	6	27
Square	4	30
Hexagon	3	32

Table 12.1: Complex Models Ordered According to Final Results

The interesting result from this is that, the epidemic will be reached according to the branching factor. This is clearly evident from Table 12.1. What can be derived from this is that, the higher branching factors will reach an epidemic more quickly, even if their initial spread rate in low. Therefore, the more branches that exist among the communities, the more effective the gypsy agents are at spreading system components.

Reaching an epidemic in this manner is typical of simple epidemiology models.

Chapter 13

Conclusion

Since the rise of pervasive computing, it has become more apparent that everyday life is a relevant aspect of technological developments. Of particular interest are community gatherings. Communities of people result in gatherings of systems. In this research project, we set out to discover what unique situations such community gatherings present to systems. Our objective was to determine how, and why, communities from the perspective of systems, could be useful. We found that current research into similar concepts is overly concerned with the context of a location, or the type of system in the location. Generally missing from the research is the capabilities that other systems located in the community could share. This led us to the concept of systems adapting to communities, and the definition of a dynamic community. A dynamic community allows for systems adaptation and community adaptation.

To enable us to develop dynamic communities, we created the DUPE framework using system adaptation. The framework was designed as a middleware framework for heterogeneous system based on our initial analysis on the requirements of dynamic communities. Our initial analysis focused on the aspects of dynamic communities, dynamic system updating (see Chapter 2) and distributed cooperation (see Chapter 3). We discovered that dynamic updating techniques for languages, such as Java, are present in many forms, and that these techniques are able to execute standardised applications. We also discovered that distributed cooperation has the ability to enable remote resources to contain system code. From this, we designed a community cooperation language for dynamic communities by allowing systems to share code resources, as Java byte code, using distributed communication (see Chapter 4). This technique allowed for dynamic

community interaction. This was then used to design the DUPE framework (see Chapters 5 and 6). We label a community resulting from the DUPE framework, a DUPE community.

Our work with DUPE communities, in turn, instigated further new ideas. The most important of these rose from an analysis into how mobile systems would use these communities. For example, a mobile system passing through a community could either present the community members with new constructs, or it could adapt to the resources available in community. We identified this capability as a physical mobile agent and termed it a 'gypsy agent' (initial concepts were presented in Section 4.4 and expanded in Section 6.6). This unique concept was a serendipitous outcome from the development of our DUPE framework. Along with these gypsy agents, and the other contributions presented in Chapter 1, several further contributions to the field arose from DUPE communities (see Section 6.6). In brief, these contributions are identified as:

- **System Alteration Deposits:** the planned inclusion of a system component within a dynamic community for system evolution.
- **Context Specific Functioning:** the use of system components as location specific context.
- **Evolution Transfer:** the use of dynamic communities and mobile adaptable systems for the transfer of system components. This contribution forms the basis for the concept of physical mobile agents, gypsy agents.

However, during our analysis of DUPE communities, we also discovered that DUPE communities, and the interaction of gypsy agents, presented new safety issues to systems that needed to be addressed (see Chapter 7). These security issues caused limitations to the interaction within a DUPE community. The main security limitation is technical. This limitation was a result of problems associated with class loader redefinition. In addressing this issue, we determined that some DUPE framework implementations are likely to lose some flexibility of system adaptation as a result of a security requirement. We did develop one technique to account for this limitation, however, unfortunately, the technique is only applicable to that specific DUPE implementation. We note this limitation as a limitation of the framework but not of the concept of dynamic communities. Nevertheless, we recommend changes to the JVM that may allow the security of the framework to remain whilst giving DUPE systems more flexibility in their community interaction. Further to this limitation, we identified that the general

trust issues related to certificate chains remain for the DUPE framework. We note that although the use of certificates does provide some level of trust, they do not provide indisputable trust. Therefore, the changes to the security and trust elements of DUPE and are anticipated as future work. Other than this limitation the framework design complied with all requirements of dynamic community adaptation.

To further assess the contributions from this thesis (see Chapter 1 for complete thesis contribution details), three DUPE framework implementations were developed; these were DUPE Lite, DUPE 5.0 and DUPE JPDA (see Chapters 8, 9 and 10 respectively). Each implementation was unique. DUPE Lite is a limited implementation that is designed for standard JVMs, DUPE 5.0 uses the new J2SE 5.0 specification, and DUPE JPDA was designed using the J2SE 1.4 HotSpot JVM. Both the latter implementations are fully compatible DUPE middleware. We analysed the properties of our implementations in order to assess the framework and its contributions to the field (see Chapter 11). And, as the implementations were individual, we were also able to assess the heterogeneity of DUPE communities. This was an important aspect of our original goal. Our analysis of DUPE communities was comprehensive. We used situation tests to show that DUPE systems produced dynamic communities, and to demonstrate that the conceptual contributions of the work were present. We also assessed the overhead of the DUPE implementations. During this analysis we saw that in most implementations there is a trade-off between extended community interaction and overhead; for example, DUPE JPDA was able to achieve more tasks in a DUPE community, however, it required more system resources than, for example, DUPE 5.0. From the perspective of DUPE communities, this analysis was included to determine the technical implications of DUPE community interaction. The results from this section of our work were as predicted.

Gypsy agents required a different approach to determining their contribution to the field. We designed simulations on the movements of gypsy agents throughout dynamic communities and modeled the rate at which they spread a system component (see Chapter 12). This was achieved using random graph modeling. This approach allowed us to draw similarities between gypsy agents and epidemics, traveling salesmen and random walks. This work was particularly interesting. We were able to show, on a large scale, a benefit of dynamic communities that originally, we had not contemplated. In fact, we perceive that, along with the concept of dynamic communities, the concept of gypsy agents is a unique contribution to knowledge in the field. However, this area needs more theoretical work.

Nevertheless, the concept has opened up a new area of research which, if pursued, is likely to further enhance the applicability of our work to contemporary society.

13.1 Future Research

A further result of our research is that we are motivated to continue to conceptualise new ideas associated with dynamic communities, the DUPE framework and gypsy agents. Generally, our future endeavors will relate to extensions to the interaction within dynamic communities, including security enhancements, a further analysis of gypsy agents, and the combination of our work with other fields of research.

We now identify the most interesting future work concepts that are related to this thesis. These concepts are research areas based on problems found during the testing of the DUPE implementations, or ideas that formed during our research, but were outside the requirements of this thesis.

DUPE Community Interaction

Currently, other than for DUPE Lite, system interaction in DUPE communities involves fully compatible DUPE systems. Further analysis and development of restricted DUPE implementations would provide an insight into the extent of the use of framework; for example, a limited DUPE implementation for mobile phones. This possible area of future work arose from our discussions on the general features of DUPE Lite (see Section 8.2).

Along with the above work, it is seen that testing the DUPE framework using more diverse applications is applicable to our work. Further, analysis of the type of systems that are best applied as DUPE systems is likely to discover new ways in which the concept of dynamic communities can be applied. During this thesis the focus was on allowing *any* system to make use of dynamic communities. We perceive that if systems are developed with dynamic communities in mind, then their use of community interaction constructs could be enhanced, resulting in advanced concepts; for example, the tourist application we used during our analysis in Section 11.8.1.

Multiple Community Memberships

It is possible to have multiple DUPE communities in the same location. Therefore, a DUPE system could possibly interact in more than one community at any given time. The possibilities of this concept have not been analysed in this thesis. It is anticipated that using DUPE communities in this manner would be beneficial to macro-community analysis. This concept was introduced in Section 5.6.1.

Security Improvements

We clearly identified security limitations of the DUPE framework in Section 7.2. These limitations remain for all our DUPE implementation, except DUPE JPDA (for full details on DUPE JPDA security capabilities see Section 7.2.5 and Chapter 10). We identify the solutions to this as future work. We discussed several recommendations for future work that may solve the problem in Section 7.2.6. Currently, none of these recommendations are being pursued.

Furthermore, we also discussed in Section 7.2.7 the need for further research into DUPE's use of certificates as a trust mechanism. We note that the use of certificates may be seen as an inadequate level of trust. We identify this area as future work in regards to the DUPE framework and as a research area in itself.

Advanced Gypsy Agent Modeling

The movements of gypsy agents, and their use for software transfer is an entirely new concept. Although we provided substantial analysis of this in Chapter 12, further work is applicable. Of particular interest, is further modeling of gypsy agents to determine the full extent of their relationship to epidemics. We visualise that the analysis should include concepts of epidemics in terms of gypsy agents and dynamic communities. The following are examples.

- **Cured Systems**: a DUPE system, more specifically a gypsy agent, which initially applies the software component only to remove it later and note it as unusable for future adaptation.
- **Immunization**: a DUPE systems whose adaptation and security settings are such that it will not adapt to the software component being carried by the gypsy agent(s).

- **Inner-Community Transfer**: the internal rate of infection transfer among DUPE members within a DUPE community.
- **Complete Analysis**: the combined analysis of inner-community transfer (as above) and the movements of gypsy agent among different communities.

A comprehensive analysis would take into account all the above scenarios, among others. It is likely that this analysis would draw further similarities to epidemics.

Extension to Other Programming Languages

The concept of dynamic communities is transferable to other languages. In particular, our specification of the dynamic community language can easily be applied using a different code format. Therefore, it would be possible to include aspects of dynamic communities into the design of a new programming language. This would create a standardised programming language that inherently allows applications to cooperate in dynamic community scenarios. As a result, the heterogeneity aspects of the concept would be expanded.

Research Combinations

During our early chapters, we identified techniques, such as AI and context awareness, as possible means of creating dynamic communities. And, although we concluded that these had short-comings for our research goals, the combination of them with the DUPE framework and dynamic communities is intriguing. We see that using context aware applications in combination with DUPE communities could allow systems to alter their execution in accordance with their current, selective, context, and according to the code available at that point in time. The advantages of a combination of dynamic communities with AI are more elusive; for example, dynamic communities provide a perfect situation for knowledge transfer among AI entities. We believe that this area of research is only limited by imagination.

References

[1] J. Andersson. A Deployment System for Pervasive Computing. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 262, Washington, DC, USA, 2000. IEEE Computer Society.

[2] J. Andersson and T.Ritrau. Dynamic Code Update in JDrums. In *Fourth International Software Architecture Workshop (ISAW'4) in Conjunction with ICSE'2000*, Limerick, Ireland, June 2000.

[3] J. Armstrong. Erlang — a Survey of the Language and its Industrial Applications. In *The 9th Exhibitions and Symposium on Industrial Applications of Prolog (INAP'96)*, pages 16–18, Hino, Tokyo, Japan, 1996.

[4] J. Armstrong. The development of Erlang. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 196–203, Amsterdam, The Netherlands, 1997. ACM Press.

[5] C. Austin. J2SE 1.5 in a Nutshell. Sun Microsystems, Feb 2004. http://java.sun.com/developer/technicalArticles/releases/j2se15/. Last accessed September 2004.

[6] C. Austin. J2SE 5.0 in a Nutshell. Sun Microsystems, May 2004. http://java.sun.com/developer/technicalArticles/releases/j2se15/. Last accessed September 2005.

[7] D. Ayed, N. Belhanafi, C. Taconet, and G. Bernard. Deployment of Component-based Applications on Top of a Context-aware Middleware. In *Mobile Computing Systems in Dynamic Environments Workshop, special session of the IASTED International Multi-Conference on Software Engineering SE2005*, Innsbruck, Austria, February 2005.

[8] R. Barták. Dynamic Global Constraints: A First View. In *ERCIM Workshop on Constraints*, Prague, Czech Republic, June 2001.

[9] P. Bellavista, A. Corradi, and E. Magistretti. REDMAN: A Decentralized Middleware Solution for Cooperative Replication in Dense MANETs. In *Third IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'05)*, Kauai Island, Hawaii, March 2005.

[10] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli. Context-Aware Middleware for Resource Management in the Wireless Internet. *IEEE Transactions on Software Engineering*, 29(12):1086–1099, December 2003.

[11] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli. Dynamic Binding in Mobile Applications: A Middleware Approach. *IEEE Internet Computing*, 7(2):34–42, 2003.

[12] R. Bialek and E. Jul. A Framework for Evolutionary, Dynamically Updatable, Component-Based Systems. In *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04)*, pages 326–331. IEEE Computer Society, 2004.

[13] Bluetooth. The Official Bluetooth Membership Site. https://www.bluetooth.org/, 2005. Last accessed July 2005.

[14] Bluetooth. The Official Bluetooth Website. http://www.bluetooth.com/, 2005. Last accessed July 2005.

[15] B. Bollobás. *Random Graphs*. Academic Press, London, UK, 1985.

[16] N. S. Borenstein. EMail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail. In *International Conference on Upper Layer Protocols, Architectures and Applications*, pages 389–402, Barcelona, Spain, June 1994.

[17] D. Brookshier, D. Govoni, N. Krishnan, and J. C. Soto. *JXTA: Java P2P Programming*. Sams, Indianapolis, IN, USA, 2002.

[18] L. Brown and D. Sahlin. Extending erlang for safe mobile code execution. *Information and Communication Security, Lecture Notes in Computer Science*, 1726:39–53, November 1999.

[19] R. Burioni and D. Cassi. Random walks on graphs: ideas, techniques and results. *Journal of Physics A: Mathematical and General*, 38(8):R45–R78, 2005.

[20] C. Campo, A. Marn, C. Garca-Rubio, and P. T. Breuer. Service Discovery in Multi-Agent Systems. In *Workshop on Ubiquitous Agents on embedded, wearable, and mobile devices*, Bologna, Italy, July 2002.

[21] CellPoint Connect AB. CellPoint Connect. http://www.cellpoint.com. Last accessed April 2006.

[22] D. Chess, C. Harrison, and A. Kershenbaum. Mobile Agents: Are They a Good Idea? Technical Report RC 19887 (December 21, 1994 - Declassified March 16, 1995), IBM T. J. Watson Research Center, Yorktown Heights, New York, 1994.

[23] S. Chiba. Javassist - A Reflection-based Programming Wizard for Java. In *Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, October 1998.

[24] S. Chiba. Load-Time Structural Reflection in Java. *Lecture Notes in Computer Science*, 1850:313–336, 2000.

[25] G. A. Cohen and J. S. Chase. An architecture for safe bytecode insertion. *Software - Practice and Experience*, 34:1–12, 2001.

[26] CollabNet, Inc. JXTA - Get Connected. http://www.jxta.org, 2003. Last accessed August 2005.

[27] F. Costa and P. Frasconi. Distributed community crawling. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 362–363, New York, NY, USA, 2004. ACM Press.

[28] CSIRO ICT Centre. ICT Centre - Robotics. http://www.ict.csiro.au/-robotics, September 2005. Last accessed October 2005.

[29] P. Danielsson and T. Hultén. JDrums. Java Distributed Run-time Update Management. Master's thesis, Växjö University, 2001.

[30] O. Davidyuk, J. Riekki, V.-M. Rautio, and J. Sun. Context-aware middleware for mobile multimedia applications. In *MUM '04: Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia*, pages 213–220, New York, NY, USA, 2004. ACM Press.

[31] A. Dey and G. Abowd. The Context Toolkit: Aiding the Development of ContextAware Applications. In *Workshop on Software Engineering for Wearable and Pervasive computing*, Limerick, Ireland, June 2000.

[32] A. Dey, J. Mankoff, G. Abowd, and S. Carter. Distributed mediation of ambiguous context in aware environments. In *Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 121–130, Paris, France, October 2002. ACM Press.

[33] S. Dorogovtsev and J. Mendes. *Evolution of Networks. From Biological Nets to the Internet and WWW*. Oxford University Press, New York, USA, 2003.

[34] J. Dowling. Coordinating Self-Adaptive Components for Emergent Distributed System Behaviour and Structure. In *eigth CaberNet Radicals Workshop*, Ajaccio, Corsica, October 2003.

[35] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*, pages 455–462, Washington, DC, USA, November 2004. IEEE Computer Society.

[36] W. Edwards. *Core Jini*. Prentice Hall, Upper Saddle River, NJ, USA, second edition, 2000.

[37] ej-technologies. ej-technologies: JProfiler. http://www.ej-technologies-.com/products/jprofiler/overview.html, 2005. Last accessed November 2005.

[38] erlang.org. Open-source Erlang. http://www.erlang.org. Last accessed April 2006.

[39] T. Fahringer. JavaSymphony: A System for Development of Locality-Oriented Distributed and Parallel Java Applications. In *IEEE International Conference on Cluster Computing*, Chemnitz, Germany, Nov-Dec 2000.

[40] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation (PLDI) 2003*, pages 1–11, New York, NY, USA, june 2003. ACM Press.

[41] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[42] L. Gong, G. Ellison, and M. Dageforde. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and ImplementationPermissions*. Addison-Wesley, Reading, MA, US, second edition, 2003.

[43] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, US, 1996.

[44] T. Gu, H. K. Pung, and D. Q. Zhang. A Service-Oriented Middleware for Building Context-Aware Services. *Journal of Network and Computer Applications (JNCA)*, 28(1):1–18, January 2005.

[45] R. Gupta, S. Talwar, and D. P. Agrawal. Jini Home Networking: A Step toward Pervasive Computing. *Computer*, 35(8):34–40, 2002.

[46] J. Gustavsson. JPatch.org. http://www.jpatch.org/, 2005. Last accessed August 2005.

[47] S. Hashman and S. Knudsen. The Application of Jini Technology to Enhance the Discovery of Mobile Services. Technical report, PsiNapic, Inc., December 2001.

[48] B. Hausman. Turbo Erlang: Approaching the Speed of C. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer, Dordrecht, 1994.

[49] M. Hicks. *Dynamic Software Updating*. PhD thesis, University of Pennsylvania, 2001. ACM 0-89791-88-6/97/05.

[50] M. Hicks, J. T. Moore, and S. Nettles. Dynamic Software Updating. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, June 2001.

[51] M. Hicks and S. M. Nettles. Dynamic Software Updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, September 2005.

[52] J. Hill. A software architecture supporting networked sensors. Master's thesis, UC Berkeley, December 2000.

[53] G. Hjálmtysson and R. S. Gray. Dynamic C++ classes - A lightweight mechanism to update code in a running program. In *Proceedings of the USENIX Annual Technical Conference*, pages 65–76, New Orleans, Louisiana, US, June 1998.

[54] IBM. Aglets. http://www.trl.ibm.com/aglets/, March 2002. Last accessed October 2005.

[55] IEEE. IEEE 802 General Information. http://www.ieee.org/, Nov 2004. Last accessed September 2005.

[56] Jini Community. Jini.org - The Community Resource for Jini Technology. http://www.jini.org, 2005. Last accessed November 2005.

[57] J. O. Kephart, D. M. Chess, and S. R. White. Computers and Epidemiology. *IEEE Spectrum*, 30(5):20–26, May 1993.

[58] G. Kiczales. AspectJ: Aspect-Oriented Programming in Java. In *Lecture Notes in Computer Science*, volume 2591, page 1, Jan 2003.

[59] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Lecture Notes in Computer Science*, volume 1241, page 220, Jan 1997.

[60] G. Kniesel. The JMangler Project. http://roots.iai.uni-bonn.de/research/jmangler/, 2005. Part of ROOTS: Research on Object-Oriented Technologies and Systems. Last accessed October 2005.

[61] G. Kniesel, P. Costanza, and M. Austermann. JMangler – A Framework for Load-Time Transformation of Java Class Files. In *1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, Florence, Italy, Nov 2001.

[62] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, Reading, Massachusetts, USA, August 1998.

[63] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 36–44, Vancouver, British Columbia, Canada, October 1998. ACM Press.

[64] C. Lin and T. Chen. Dynamic memory management for real-time embedded java chips. In *Seventh International Conference on Real-Time Computing Systems and Applications*, volume . 2000 Page(s):49 - 56, Dec 2000.

[65] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, US, second edition, 1999.

[66] H. Liu and M. Parashar. A Component-based Programming Framework for Autonomic Applications. In *the International Conference on Autnomic Computing (ICAC'04)*, pages 278 – 279, New York, NY, USA, May 2004. IEEE Computer Society Press.

[67] W. Lugmayr. *Gypsy: A Component-Oriented Mobile Agent System*. PhD thesis, Technical University of Vienna, Austria, October 1999.

[68] A. MacWilliams. Self-Extending Systems for Context-Aware Mobile Computing. In *Doctoral Symposium at the International Conference on Software Engineering*, Portland, Oregon, USA, May 2003.

[69] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *Proceedings of the Fourteenth European Conference on Object-Oriented Programming*, page 337, Sophia Antipolis and Cannes, France, June 2000.

[70] P. Månsson. Versatile services for mobile devices. Master's thesis, Lund University, Sweden, September 1999.

[71] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing Adaptive Software. *Computer*, 37(7):56–64, 2004.

[72] Members of the UPnP Forum. UPnP. http://www.upnp.org, 2004. Last accessed October 2005.

[73] T. Menzies. Absolute Prolog. http://www.csee.wvu.edu/∼menzies/pl-/goodasserts.html, Dec 2001. Last accessed September 2005.

[74] Microsoft Corporation. Microsoft .net. http://www.microsoft.com/net/, 2005. Last accessed May 2005.

[75] P. Mihailescu. *MAE: A Mobile Agent Environment for Resource Limited Devices*. PhD thesis, Monash University, 2003.

[76] K.-D. Moon, Y.-H. Lee, and C.-K. Kim. Context-Aware and Adaptive Universial Home Network Middleware for Pervasive Digital Home Environment. In *IEEE Consumer Communications and Networking Conference (CCNC2004)*, Las Vegas, NV, USA, January 2004.

[77] C. J. Murray. Bluetooth rools toward deployment on cars. http://www.-linuxdevices.com/news/NS5234339457.htm, August 2002. Last accessed July 2005.

[78] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Random graphs with arbitrary degree distributions and their applications. *Phys. Rev.*, 64(026118), 2001.

[79] J. Newmarch. *A Programmer's Guide to Jini Technology.* Apress, New York, USA, 2000.

[80] J. Newmarch. A Critique of Web Services. In *IADIS International Conference E-Commerce*, Lisbon, Portugal, December 2004.

[81] S. Oaks. *Java Security.* O'Reilly Media, Inc., Sebastopol, CA, USA, 2nd edition, 2001.

[82] Object Management Group, Inc. CORBA. http://www.corba.org/, 2005. Last accessed September 2005.

[83] K. Ohata, K. Maruhashi, M. Ito, S. Kishimoto, K. Ikuina, T. Hashigucgi, N. Takahashi, and S. Iwanaga. Wireless 1.25Gb/s Transceiver Module at 60GHz Band. *Solid-State Circuits Conference 2002. Digest of Technical Papers ISSCC. 2002, IEEE International*, 1:298 – 468, February 2002.

[84] OSGi Alliance. The OSGi Service Platform - Dynamic services for networked devices. http://www.osgi.org/, 2005. Last accessed March 2005.

[85] J. K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[86] J. K. Ousterhout, J. Y. Levy, and B. B. Welch. The Safe-Tcl security model. *Lecture Notes in Computer Science*, 1419, 1998.

[87] PsiNaptic Inc. JMatos. http://www.psinaptic.com/j_matos.jsp, 2004. Last accessed April 2005.

[88] K. Ragab, N. Kaji, Y. Horikoshi, H. Kuriyama, , and K. Mori. Autonomous decentralized community communication for information dissemination. *IEEE Internet Computing*, 8(3):29–36, May-Jun 2004.

[89] A. Ranganathan, J. Al-Muhtadi, S. Chetan, R. Campbell, and M. D. Mickunas. MiddleWhere: a middleware for location awareness in ubiquitous computing applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 397–416, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[90] C. W. Reynolds. Flocks, birds, and schools: a distributed behavioral model. *Computer Graphics*, 21:25–34, 1987.

[91] M. J. Rochkind. The Source Code Control System. In *Proceedings of the 1st National Conference on Software Engineering*, pages 37–43. IEEE Computer Society Press, 1975.

[92] A. Ryan and J. Newmarch. A Dynamic, Discovery Based, Remote Class Loading Structure. In *Proceedings of the Seventh Annual IASTED Conference on Software Engineering and Applications*, Marina Del Ray, CA, US, November 2003.

[93] B. Schilit, N. Adams, and R. Want. Context-Aware Computing Applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[94] C. Sørensen, M. Wu, T. Sivaharan, G. S. Blair, P. Okanda, A. Friday, and H. Duran-Limon. A context-aware middleware for applications in mobile ad hoc environments. In *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, pages 107–110, New York, NY, USA, 2004. ACM Press.

[95] A. Sommerer. *The Java Archive (JAR) File Format.* Sun Microsystems, Inc, Sept 1998. Available at: http://java.sun.com/developer/Books/-javaprogramming/JAR/index.html.

[96] Sun Microsystems. The Java Debug Interface (JDI) API. http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jdi/. Last accessed December 2005.

[97] Sun Microsystems. Dynamic code downloading using RMI. http://java.sun.com/j2se/1.3/docs/guide/rmi/codebase.html, 1999. Last accessed December 2005.

[98] Sun Microsystems. J2ME Building Blocks for Mobile Devices. Technical report, Sun Microsystems, Inc., Palo Alto, CA USA, May 2000.

[99] Sun Microsystems. Java Platform Debugger Architecture Overview. http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jdi/index.html, 2001. Last accessed December 2005.

[100] Sun Microsystems. Java RMI over IIOP Technology Documentation Home Page. http://java.sun.com/j2se/1.4.2/docs/guide/rmi-iiop/index.html, 2002. Last accessed February 2005.

[101] Sun Microsystems. Permissions in the Java 2 SDK. http://java.sun.-com/j2se/1.4.2/docs/guide/security/permissions.html, 2002. Last accessed September 2005.

[102] Sun Microsystems. Java 1.4.1 api. http://java.sun.com/j2se/1.4.1/docs/-api/, 2003. Last accessed September 2005.

[103] Sun Microsystems. The Java HotSpot Virtual Machine. Technical Report v1.4.0, Sun Microsystems, Inc., February 2003.

[104] Sun Microsystems. Java 2 SDK, Standard Edition Documentation Version 1.5.0 Beta 1. http://java.sun.com/j2se/1.5.0/doc/index.html, April 2004. Last accessed April 2004.

[105] Sun Microsystems. Java Remote Method Invocation (Java RMI). http://java.sun.com/products/jdk/rmi/, 2004. Last accessed October 2004.

[106] Sun Microsystems. JDK 5.0 Documentation. http://java.sun.com/j2se/-1.5.0/docs/, 2004. Last accessed October 2005.

[107] Sun Microsystems. CLDC HotSpot Implementation Virtual Machine. Technical report, Sun Microsystems, Inc., Santa Clara, CA USA, February 2005.

[108] Sun Microsystems. Core Java J2SE 5.0. http://java.sun.com/j2se/1.5.0/-index.jsp, 2005. Last accessed December 2005.

[109] Sun Microsystems. Desktop Java - Java Web Start Technology. http://-java.sun.com/products/javawebstart/, 2005. Last accessed October 2005.

[110] Sun Microsystems. Jini Network Technology - Specifications. http://www.sun.com/software/jini/specs/, 2005. Last accessed November 2005.

[111] Symbian Ltd. Symbian OS - The Mobile Operating System. http://www.-symbian.com, 2005. Last accessed July 2004.

[112] Tcl Developer Xchange. Tcl developer Xchange. http://www.tcl.tk, 2006. Last accessed April 2006.

[113] Tcl Developer XChange. Tcl/Java Project. http://tcljava.sourgeforce.net/-docs/website/index.html, 2006. Last accessed April 2006.

[114] The LEGO Group. Lego Mindstorms. http://mindstorms.lego.com, 2005. Last accessed July 2005.

[115] W. F. Tichy. RCS - A System for Version Control. *Software - Practice and Experience*, 15(7):637–654, July 1985.

[116] Y. Vandewoude and Y. Berbers. An Overview and Assessment of Dynamic Update Methods for Component-oriented Embedded Systems. In *The international Conference on Software Engineering Research and Practice*, Las Vegas, NV, USA, June 2002.

[117] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill Professional, 1999.

[118] P. Wallich. Mindstorms: Not Just a Kids Toy. *IEEE Spectrum*, 38(9):52–57, September 2001.

[119] G. Weiss, editor. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT Press, Cambridge, MA, USA, 1999.

[120] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart. An Architectural Approach to Autonomic Computing. In *International Conference on Autonomic Computing (ICAC'04)*, New York, NY, USA, May 2004.

[121] Wikipedia. Wikipedia, the free encyclopedia. http://www.wikipedia.com, 2005. Last accessed November 2005.

[122] World Wide Web Consortium (W3C). W3C World Wide Web Consortium - Leading the Web to Its Full Potential... http://www.w3.org/, April 2005. Last accessed May 2005.

[123] Y. Ye, Y. Yamamoto, and K. Kishida. Dynamic Community: A New Conceptual Framework for Supporting Knowledge Collaboration in Software Development. In *11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pages 472–481, Busan, Korea, Nov - Dec 2004.

[124] M. Zhong, K. Shen, and J. Seiferas. Non-uniform Random Membership Management in Peer-to-Peer Networks. In *IEEE Infocom*, Miami, FL, USA, March 2005.

[125] Ziff Davis Publishing Holdings Inc. Linux + Java turbocharge "Super8 Hemi" concept car. http://www.linuxdevices.com/news/-NS5234339457.htm, Janurary 2001.

[126] Zigbee Alliance. ZigBee Alliance - Wireless Control that Simply Works. http://www.zigbee.org/en/index.asp, 2006. Last accessed April 2006.

Appendix A

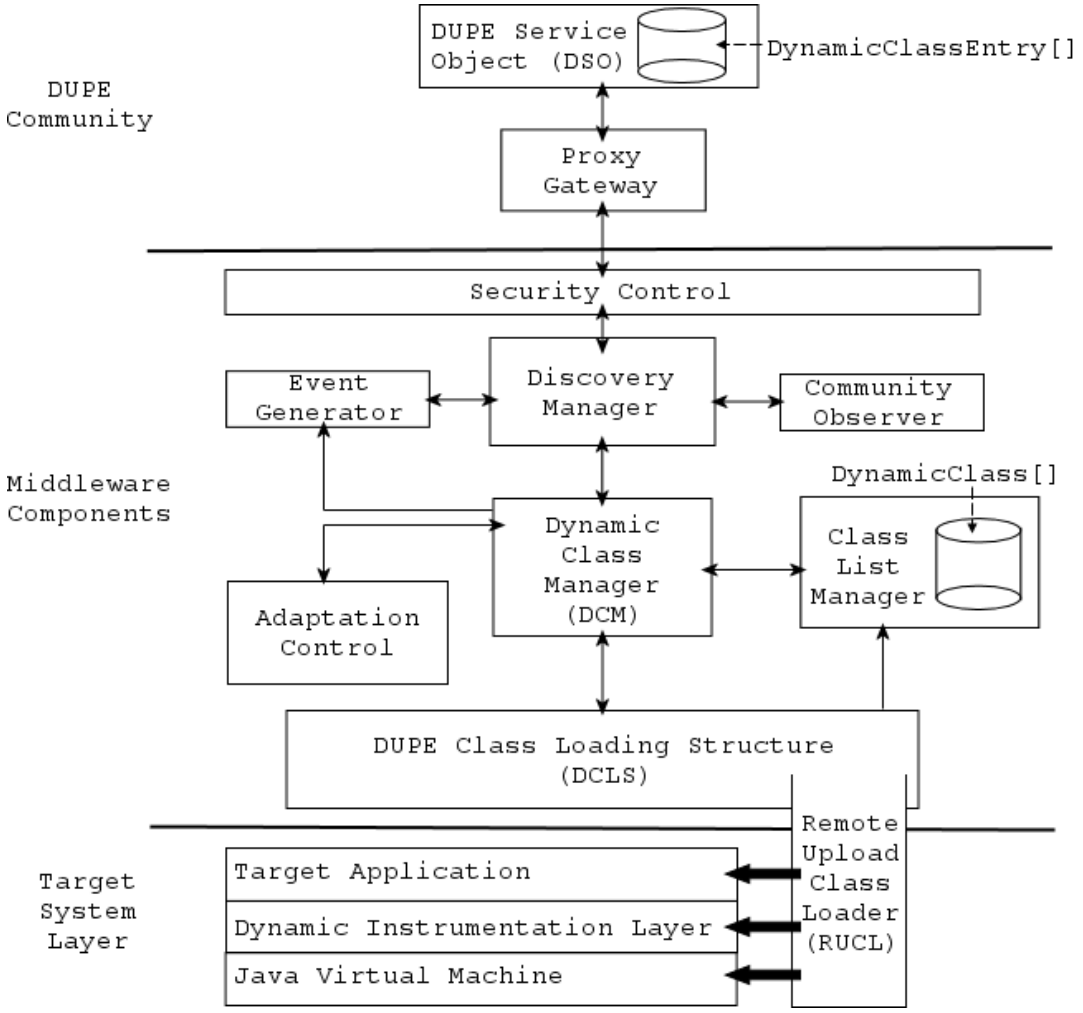
DUPE Specification

This appendix provides the high-level specification for implementations of the DUPE framework. The specification focuses mainly on the communication language of DUPE communities. The elements of the framework related dynamic redefinition are generally specific to the technique applied, consequently, only a broad specification of this section is defined. Specifically provided in this specification are:

- Requirements of a DUPE Compatible Middleware
- Requirements for DUPE System Adaptation
- Requirements for using a DUPE Service Object, including:
  - The specification of a LoaderType
  - The specification of a DynamicClassEntry
  - The specification of a DynamicClass
- DUPE Community Events
- Security Specification
- Adaptation Specification
- Configuration Files
- Target Application Requirements

Requirements of DUPE Compatible Middleware

There are several requirements that must be met for a DUPE implementation to be recognised as DUPE compatible. To incorporate all the requirements of the framework, the following is a suggested implementation design.



There are two levels of compatibility for these requirements.

Mandatory: a section of the framework which *must* be included for a DUPE compatible middleware.

Standard: a section of the framework which is generally included for a DUPE compatible middleware. If the section is excluded, the DUPE implementation can still interact as a member of a DUPE community, however, its

interaction is on a limited level. All such DUPE implementations are identified as limited DUPE middleware.

The requirements are listed below.

Section	Description	Inclusion
DUPE community discovery	The ability to locate and recognise a Jini Registrar as a DUPE community	Mandatory
DSO Understanding	The ability to understand the contents of a DSO.	Mandatory
DSO Creation	The ability to create and advertise a DSO object.	Standard
DSO Discovery	The ability to discover DSO objects within a DUPE community.	Standard
Load time dynamic code redefinition	The ability to alter the state of Java byte code during start up.	Standard
Runtime dynamic code redefinition	The ability to alter the state of Java byte code during runtime.	Standard
DUPE Event Recognition	The understanding of a DUPE community event and its contents.	Mandatory
DUPE Specification Understanding	The understanding of the contents of DUPE specification details associated with DynamicClass and DynamicClassEntry objects.	Mandatory
Certificate Recognition	Differentiating between an illicit or unwanted certificate.	Mandatory

Requirements for Dynamic System Adaptation

A fully compatible DUPE middleware can dynamically update the runtime code of a target system using DUPE community resources. This aspect of the specification results in system adaptation. However, this feature of the DUPE framework is not mandatory (although it is recommended for most implementations).

The specification of any dynamic system updating implementation will be unique. Consequently, an exact specification of this aspect of a DUPE implementation can not be given. However, all implementations should maintain the following during execution:

- Control the class loading of the entire target system during execution.
- Search for class details within a DUPE community when they can not be located locally, or in response to a DUPE event.
- Redefine class details based on the resources of a DUPE community[1].
- Maintain a dynamic list of current executing classes.
- Continuously link all class details to an associated DSO.

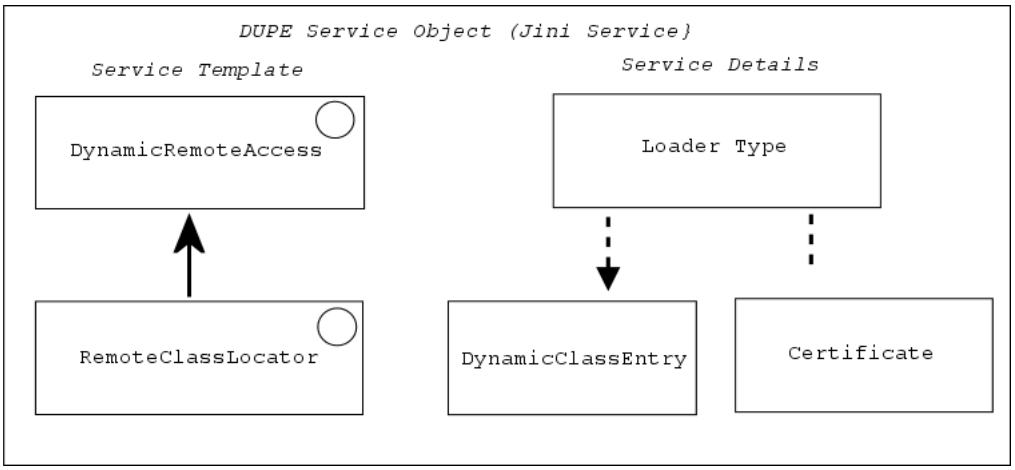
This element of the framework specification should be maintained within the DUPE Class Loading Structure, the Remote Upload Class Loader, the Dynamic Class Manager and the Class List Manager.

Requirements of a DUPE Service Object

The most exact elements of the DUPE specification language specification are contained within the DUPE Service Object (DSO). The specification of the DSO must be followed exactly to ensure interaction in a DUPE community is correct.

The following classes and interfaces are requirements of the DSO:

- `dupe.discovery.DynamicRemoteAccess.java (interface)`
- `dupe.discovery.RemoteClassLocator.java (interface)`
- `dupe.discovery.LoaderType.java`
- `dupe.dicsovery.DynamicClassEntry`
- `java.security.cert.Certificate`



The structure of the DSO as a Jini Service is provided above.

The specification of `java.security.cert.Certificate` is located in the Java Security Specification [42]. The specification of the other classes and interfaces will now be detailed.

DynamicRemoteAccess

The DynamicRemoteAccess interface determines the basic requirements of the DSO. This interface provides DSO with the methods used for gathering remote class details.
The `dupe.discovery.DynamicRemoteAccess.java` interface is as follows.

```
public interface DynamicRemoteAccess extends java.io.Serializable, RemoteEventListener{
  public DynamicClass generateDynamicClass(String className)
        throws ClassNotFoundException, RemoteException;
  public void notify(RemoteEvent evt) throws RemoteExceotion, UnknownException;
}
```

RemoteClassLocator

The RemoteClassLocator is the main interface of the DSO. This interface should be used for DSO discovery and advertising. However, it essentially provides only

[1]All class updates should be according to security and adaptation settings.

the methods of its super interface, DynamicRemoteAccess. The extra methods provided by this interface will generally be empty for implementation as they are unnecessary at this stage. It is likely that the extra methods will be removed in future changes to the specification.
The `dupe.discovery.RemoteClassLocator.java` interface is as follows.

```
public interface RemoteClassLocator extends DynamicRemoteAccess {
  public Class loadClassBytes(byte[] classData, String name) throws java.rmi.RemoteException;
}
```

LoaderType

The LoaderType class is the Jini details class of a DSO. This class contains an array of DynamicClassEntry objects that represent the resources that can be accessed via the associated DSO. The LoaderType API is now provided.

`public class LoaderType implements net.jini.core.entry.Entry`[2]		
Name	Type	Description
`hostName`	`String`	The name of the DSO's server system. May be set as `null`.
`allowInternalSearch`	`boolean`	Indicates if the DSO's server system allows internal searching for uninitiated resources.
`classes`	`DynamicClassEntry[]`	An array DynamicClassEntry objects representing the resources known to be available on the DSO's server system.

DynamicClassEntry

The transfer of class resources throughout a DUPE community is achieved using `dupe.common.DynamicClass` objects. The specification of this class is below.

The adaptation attributes in a DynamicClassEntry must be stored correctly. This is to make sure that all DUPE systems can understand them.

public class DynamicClassEntry implements Serializable		
Name	Type	Description
`className`	`String`	The reference name for the class file resource.
`attributes`	`String[][]`	The details of all attributes that have been assigned to the class resource.
`certificates`	`java.security.-cert.Certificate[]`	An array of certificates that have been assigned to this class resource.

DynamicClass

After a DynamicClassEntry is designated as useful to a DUPE system, it can request the complete details of the associated class. This achieved using the `generateDynamicClass` method of the DSO.The return type of this method call is a `dupe.common.DynamicClass` object. The API for this class is provided.

public class DynamicClass implements Serializable		
Name	Type	Description
`classBytes`	`byte[]`	The class bytes of the Java class which the objects represents.
`name`	`String`	The class name of the Java class which the objects represents.
`attributes`	`String[][]`	The adaptation attributes as assigned at the initial compilation of the class.
`certificates`	`java.-security.cert.-Certificate[]`	Certificates that have been assigned to the class.

DUPE Community Events

All DUPE community events are in the form of Jini events. Specifically, they are all an instance of `net.jini.core.lookup.ServiceEvent`. For the specification on how to discover and listen for Jini events see the Jini Specification [110].

According the Jini specification, in a DUPE community the DSO (see previous section) should be used for discovery and advertising purposes.

The following details how the Jini events are interpreted as DUPE community events:

Name	Description
<TRANSISITION_NOMATCH_MATCH>	New member into the community or an update of current member.
<TRANSISITION_MATCH_NOMATCH>	Exited member.
<TRANSISITION_MATCH_MATCH>	Change in current member, but not necessarily an update of DUPE details.

The DUPE information of the event is found as a DSO representation. A DUPE community event is generated by a DUPE middleware according to the execution of the target application. This element of the specification should be maintained in the Discovery Manager, the Event Generator and the Community Observer.

Security Specification

There are two elements of the security specification which are standardised for all DUPE communication. All other trust aspects of security involve Certificate recognition. The Certificates are located either within the Jar file for the details of Certificates), the DSO (Section A) or the DynamicClass (Section A). For the complete specification on understanding Java Certificates see the Java Security Specification [42].

The most important part of the DUPE Security Specification is its Permission sets. There must be two sets of Permissions:

- a Permission set governing the DUPE middleware, and
- a Permission set governing the target application.

However, if possible, the Permission set for the target application should be split further into two, making three Permission sets:

- a Permission set governing the DUPE middleware,
- a Permission set governing local classes of the target application, and
- a Permission set governing downloaded classes of the target application.

An implementation with this type of permission setup is safer.

Moreover, the following Permission sets should only be allowed cautiously. Unless absolutely necessary, they should remain locked to a target application.

- `java.security.AllPermission`
- `java.security.SecurityPermission`
- `java.io.FilePermission`
- `java.lang.reflect.ReflectPermission`
- `java.util.PropertyPermission`
- `java.lang.RuntimePermission`, specifically:
  - `RuntimePermission.createClassLoader`
  - `RuntimePermission.setContextClassLoader`
  - `RuntimePermission.setSecurityManager`
  - `RuntimePermission.createSecurityManager`
  - `RuntimePermission.exitVM`
  - `RuntimePermission.getProtectionDomain`
  - `RuntimePermission.accessDeclaredMembers`

The security of a DUPE system should be maintained within the Security Manager.

Adaptation Specification

The adaptation of a DUPE system is based on the Adaptation Details of DUPE resources. These details are in both the DynamicClassEntry and the DynamicClass objects that are associated with a resource. Other settings may be added to some resources, however, the specification only states that those shown below are necessary.

Attribute	Description
`Name`	Full class name of package name.
`Specification-Title`	The Specification Title for the associated class.
`Specification-Version`	The Specification Version for the associated class.
`Specification-Vendor`	The Specification Vendor for the associated class.
`Implementation-Title`	The Implementation Title for the associated class.
`Implementation-Version`	The Implementation Version for the associated class.
`Implementation-Vendor`	The Implementation Vendor for the associated class.

The adaptation of a DUPE system is a result of DUPE community events found by the Community Observer. Determining if an event is useful should be achieved within the Adaptation Control.

Configuration Files

The configuration files follow standard Policy file structure. Using this technique results in a simplification of configuration and also provides a format for the configuration files. An example of an the adaptation configuration file is now provided.

Configuration files can be redesigned according to the requirements of a DUPE implementation.

```
#
#Wed Nov 10 15:42:01 EST 2005
SPECIFICATION_VERSION_CHECK=true
SPECIFICATION_VENDOR_CHECK=false
IMPLEMENTATION_TITLE_CHECK=false
IMPLEMENTATION_VENDOR_CHECK=false
IMPLEMENTATION_VERSION_CHECK=true
SPECIFICATION_TITLE_CHECK=false
```

Target Application Requirements

There are several compilation and distribution requirements that are necessary for DUPE community interaction, that *must* be applied to any target application of a DUPE system. These are:

1. the application must be complied using `javac` with the `-g` option
2. the application must be contained in a signed Jar file
3. the Jar file must have an associated Manifest file containing all adaptation information, and
4. the Jar file must be signed with a minimum of one certificate.

There are no special coding requirements that must be met for the application of a DUPE system. However, as a result of the security constraints of the DUPE specification, there are aspects of standard programming that are not recommended. For example, an application wishing to create its own class loader may not execute correctly in some DUPE systems. This is a result of the DUPE framework's security specification.