A Dynamic, Discovery Based, Remote Class Loading Structure.

Adrian Ryan
School of Network Computing
Monash University
Melbourne, Victoria, Australia
email: adrian.ryan@infotech.monash.edu.au

Jan Newmarch
School of Network Computing
Monash University
Melbourne, Victoria, Australia
email: jan.newmarch@infotech.monash.edu.au

ABSTRACT
Remotely accessible services enable current day systems to gather information, generate programs, access server data and interoperate throughout environments. Such techniques may be associated with the web, service discovery, and remote object based technologies. However, there is no technique which incorporates all the positive aspects of each of the currently used methods.
This paper presents a remote class loading structure that expands the reach of a Java Virtual Machine (JVM) to beyond its host system. By interlinking remote classloaders within an environment the structure allows dynamic access, via advertisement and discovery, to each system's accessible class files.

KEY WORDS

Remote Class Loading, Dynamic Class Discovery, Software Engineering Applications.

1 Introduction

Java class bytes are normally loaded into a JVM through the use of a class loader [1]. This can be achieved by using Java's standard bootstrap loader or a specialised loader such as URLClassLoader, AppletClassLoader(s) or RMIClassLoader [2].
Specialised classloaders may instruct a program to gather java class bytes from abnormal places or in an irregular fashion. For example, the AppletClassLoader is able to load a java class from a specified url location [2].
We categorise the discovery and utilisation of remotely accessed classes into two technique types, direct downloading (Web based applets and scripts) and Remote Objects (distributed techniques such as RMI, JINI and CORBA). Each method for the gathering of class information has been developed for specific reasons and uses. Remote objects aid in distributed system connectivity, whereas web based techniques are designed to enable thin minimally connected clients. However, we find that neither technique is able to construct an application from class files that the application does not have nor know it will need. We aimed to develop a technique where a system is able to generate an entire program never having to know where most of the files are located. A tourist application, a home system's self configuration and interlinking, and zero configuration within environments are examples of applications that would benefit from such a technique. One scenario we envisage is a tourist system which allows users to run a destination specific application via the same program. Tourist destinations are able to develop unrestricted informative applications that are accessible for users to implement temporarily. The design is such that an application never physically resides on the users device, start up and runtime changes can progress through the use of distributed class files.
This paper introduces a specific class loading structure which allows dynamic loading through the use of a distributed methodology. In conjunction with discovery techniques we have developed a specialised classloader that can be used to eliminate the need for classes to exist on the same machine as the host program. Our use of a discovery based class loading structure provides access to remote classloaders which can be discovered and used 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 classloader. Furthermore, all Java classes are potentially accessible as there is no unique programming structure required in class development. The remote class loading structure can be used to create a dynamic environment where interacting systems, through the use of their classloaders, can access all classes available to each system.
The first section of this paper details the techniques currently used for remote class access, discussing how each concept was used to design the remote class loading structure. The remote class loading structure is then examined in the second section. The third section covers the testing of the class loading structure using several examples. Finally, we consider further uses and finalise by detailing our observations, future work and conclusions.

2 Distributed Classloading

Changes to a program's structure can be initiated and completed via distributed techniques, including the ability to upload remote classes or objects [3]. Such structure changes may then allow systems to alter their behaviour according to their interacting network.
OSGi [4] based systems give devices, mainly home based

devices, the opportunity to access specific details from within their environment. This enables them to configure their internal structure giving themselves access to other systems and information within the network, or outside it. For the changes to occur, details in the form of files or attributes must be located in an accessible section of a system for them to be found [4].
In such an environment access to the remote files and objects involve program design and construction adhering to the specific standards of the technique used.

2.1 Remote Objects

Distributed techniques that use remote object manipulation, such as Jini, RMI and CORBA [3], are also constrained to their specific designs and techniques. For example, within Jini and RMI an object, class, or proxy service that is to be accessed remotely needs to implement java.rmi.Remote or java.io.Serializable somewhere within its structure [3]. Furthermore, it must also implement an interface that is known to both the server and the client.
This technique is ideal for remotely accessing sections of a server that exist for the use of clients and the sharing of information. By specifying the methods of a remote object through use of an interface, restrictions on the type of processing that may be performed by a client can be established.
Although remote objects aid in the connectivity of clients and services in distributed systems, there are several areas that are not clearly defined. For example, a client may wish to run a process using a remote object yet wishes the computation to be performed without the acknowledgement of a server. Or, connection with a server may not be the main purpose of the client. This may be a situation where the client does not wish to create an instance of a remote object, but only to create an instance of a class that is located on a separate system.
Some distributed systems use a marshalling technique where there is no actual passing of class bytes until an object has been discovered. This newly discovered object , in the form of a marshalled object (as is the case in Jini), then points the client system to the actual class byte code for it to be initialised [5]. Therefore, using such a technique we are unable to create an instance of a class without having a reference to an existing object on the server side.
However an advantage of remote objects is the use of proxy services, which is a remote object that allows a client access to server side processing. When advertising a proxy service we deal with a situation where the processing can be achieved on the server and results are passed back to the client. To achieve this link the proxy service object must still implement the specified common interface(s). Once again an interface that must be known to both server and client.
The ability for remote object applications to create instances of remotely based classes is derived from their access to stub files and interfaces. The creation of the class instance, by this we mean the initialisation and instantiation of the remotely based class, is still achieved through the JVM's classloader. However, in the case of RMI and Jini a RMIClassloader will handle the loading. The RMIClassloader is able to load classes using the codebase URL reference that is stored within a discovered Marshalled object. This information is then used by the classloader to find the needed class bytes and generate a class instance. The class instance is then merged with the serialised object information from within the Marshalled object to regenerate the remote object [5, 6].
This limitation within Jini's structure is generally not noted as a disadvantage. Jini based applications are usually designed to make use of its distributed flexibility and discovery techniques, not to manipulate the loading of remote class files.

2.2 Web Based Classloading

Web based programs do not use remote objects instead they are able to access remote code. This gives them the ability to load program code that does not exist on a client's system. Unlike distributed techniques they are able to load class files without the need for a linked object to a host. Within web based loading there is no use of remote objects as such, yet there is still a need to generate objects from classes that do not exist on a client's system. Java applets use a specific classloader designed to direct clients back to a service (such as a http server), or specified port/proxy, located on a server (usually this is the originating server of the applet). The client's classloader is then able to download class files as they are instantiated, generating local objects based on the remote class byte code [6]. This allows clients to run applications that do not exist on their systems. It also gives vendors control over versioning, security specifications, personalised attributes and upgrades. However there is a lack of dynamism and location unawareness which may disadvantage the client, particulary its classloader. A client must know the correct server location to contact if it is to locate any class files.
Current web based applications, for example applets, are not greatly effected by this limitation, as they only need a static address to find information. It may be assumed that a client has contacted the site for a specific reason and therefore should not require any redirection from this location. This allows the applet to assume the originating static address provides all the necessary classes for the application to execute correctly.
Much like applets, Java's Web Start allows a client to generate a program via web based links [7]. It runs within a web browser (however this is not always necessary) and is able to generate applications from scratch through a specified URL. Web Start was created to remove the confusion that surrounds the installation of multiple applications and their version control [7]. Although Web Start is similar to applets there are distinct differences.

1. Web Start handles much larger applications and is not confined to the browser.

2. Web Start uses caching to allow users to run applications when not connected to a network.

3. Web Start, when compared to applets, gives a greater flexibility to its programs allowing for greater control and creativity.

4. Web Start needs to be installed on a users machine before it is able to be used.

Web Start still has the restriction of using a URL type link, and although it is able to recreate previously run applications, initially it must still know exactly where to get the application from.
Web server type operations are unlikely to remain the same for any considerable amount of time. The evolution of wireless devices, networks, and communication techniques will generate a need for more unrestricted techniques of communication and interaction. The flexibility of systems, namely their discovery and adaptability techniques, may determine the useability of a device or system within any environment (wireless or not) that it is likely to encounter. This is not to say that static based concepts will be completely superseded nor that they cannot be utilised to construct the more adaptable techniques that are needed.

3 A Remote Classloader

Using concepts and techniques from both remote objects and web based classloading may enable systems to load classes from remotely accessed classloaders that they initially had no knowledge of. Such a structure would have no need for the most limiting aspects of both techniques, a *common interface* and a *URL reference*.
It would enable systems to share classes between themselves, whilst being able to find and use classes without prior knowledge of their whereabouts. Remotely accessible classloaders also enable centralised systems, similar to applets and Web Start based applications, to download the files from servers without requiring the application to directly specify a URL address. Users are able to run programs on their own systems even though only a few initial files need to be stored directly within its memory (This may be avoidable through an initial downloading technique not covered in the scope of this paper). This particular attribute will prove extremely useful for limited memory devices and embedded systems.

3.1 Java Classloading

Java's class loading is optimally designed to aid in the dynamic linking and loading of class files [1]. Whilst it allows files to be located, loaded, and linked during runtime the classloader also aids in the segregation and distribution of byte code within the JVM [1]. The dynamic nature of Java's classloading and its close connection with class byte code allows designers to use specialised classloaders to manipulate class files in unspecified ways. Furthermore, the hierarchial structure that is produced when classes are loaded into a JVM allow for the defining of an applications loading techniques. As classes are located, loaded and linked via a classloader, be it the bootstrap or not, any subsequent classes, or classes instantiated via an already loaded class, will be (if not previously loaded) loaded via the same class loader [1]. 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 classloader, thus loading the entire system through a unique class loading structure. Furthermore, if the same system then loads another different classloader, say for a specific section of its system, the additional classloader will then be loaded through the first classloader. If the second classloader fails then all class processing will be delegated to is parent classloader (the first classloader) and so forth [1].

3.2 Remote Access to Classloading Methods

The bootstrap classloader methods that are used to find, load, define, and initialise classes, work together forming the total loading structure of the JVM. The initial loading structure of the classloader, as detailed by Lindholm and Yellin [1], is relatively simple in architecture. Other methods may be called during the loading process, particularly in the case of a personalised classloader, yet all original methods will still be called by the JVM in the same manner and procedure. A programmer is able to alter the structure of these methods in order to achieve a specific task, redefining how the loading structure intercommunicates and operates.
By defining interfaces that must be implemented by a remote classloader we are able to determine a new set of procedures that are available. These additional methods sit in between the classloader and a gateway/proxy (as is discussed in section 4) enabling access to the classloader itself. This then allows a remote system to check for classes, loaded or unloaded, that exist within the reach of a classloader that has been *discovered.* They also allow all accessible classes (essentially anything within reach of the classloader) to be downloaded and used as needed. This leaves the discovered classloader to only retrieve and send the class byte code and not initialise it within its own JVM. Within this specification the definition of specific methods are used to maintain the separation between a classloaders normal operation and that which can be invoked remotely. The structure allows a classloader to look amongst its own designated classpath for files (or further if internally specified) and within its JVM to find any java classes that are requested. It will then transfer the class byte code, if available, to the requesting system. They do not however pass any byte code through their own virtual machine, this should not be achieved until the defineClass() (a bootstrap

classloader method) method is instantiated within its own structure [1].

3.3 Remote Discovery and Advertisement

A key to the remote class loading structure is its dynamic discovery of remote classloaders. It gives applications restricted access, through a distribution technique, to the actual inner sections of a remote JVM. This technique provides the structure with two of its crucial aspects.

1. The ad-hoc discovery of classloaders. There is no need for a system to know an actual address of the classloader they are going to use to gather class bytes.
2. The transfer of byte code through any remotely accessible classloader. Every class that is searched for by a JVM does not need to be located and loaded via the same system.

These unique attributes give the remote classloading structure the ability to create applications that are more flexible and adaptable within environments such as a wireless network. Yet it still maintains the direct and precise constructs that are needed for use within more concrete environments such as in a server/client situation.

4 Structure

Access to a remote classloader is not achieved physically through the actual classloader but through a remote proxy and gateway setup. If a system was to gain access to an actual remote classloader it may still enable the loading and unloading of classes within a system. Yet all classes loaded would need to be contained within that same system.
To enable remote loading from remote systems a proxy/gateway can be used. The proxy and gateway is able to establish a connection, via a remote object, between separate classloaders facilitating a search procedure to access files on the discovered system. Furthermore, the gateway within the structure may allow for a security extension and user defined file access.
Initial specifications for a remote classloader, although flexible, constrict it to being both a discovery and an advertiser. Therefore, each remote class loading structure will advertise itself as an available service, whilst also having the ability to discover other class loading services if needed (see figure 1). This allows the services to create a web of accessible information for all systems to access files from separate loaders.
However, this is left relatively flexible as there are no constraints that disallow the use of a “discover only” classloader. This extensibility has been introduced for further on-going development within resource constrained devices such as mobile phones and personal data assistants (PDAs). Such systems may be noticeably interrupted by the uploading of a file from within its system and thus may wish to selectively exclude some features.

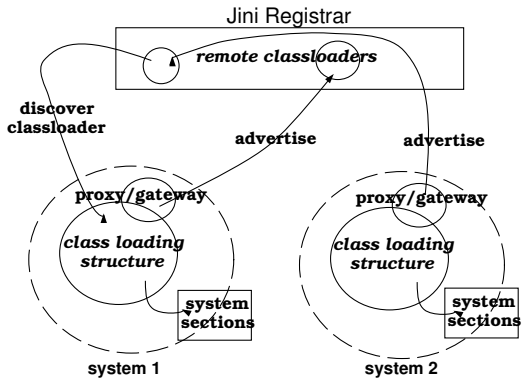

Figure 1. Discovery and Advertising of Remote Class Loaders

4.1 Initialising the Target Application

The design of Java’s class loading is flexible enough that it allows the creation of our versatile and adaptable remote class loading structure. A classloader is the basis for the loading and initialisation of Java programs. It is therefore reasonable to suggest that the remote class loading structure can be used as the underlying loading mechanism for any Java based program. This gives the remote class loading structure even greater flexibility and useability as there is no limitation to the type of program that can gain from its techniques.
To allow this our structure loads any start class of a program in the same manner as a JVM would. It searches the targeted start class for its main() method, and if found invokes it. The hierarchy design of the java loading structure then specifies that all the following classes derived from the initiating class will be loaded through the same classloader, this being the remote classloader. Meaning that all classes instantiated by any program that has been started using the remote class loading structure will also be loaded using the remote classloader.
This is not to say that a developer cannot produce their own classloader, for example one that overwrites the structure of the remote classloader. However, Java’s hierarchial class loading architecture means the unique classloader will revert back to the remote structure if, or when, it fails [1]. Therefore, the remote classloader may still be used as a backbone for the application.

5 Testing

To efficiently test the remote class loading structure, a test scenario and corresponding application was developed. Although the remote class loading structure is designed to be used by any Java program a specifically developed system would enable us to test its characteristics and determine its practicality within the scope of our research.

5.1 Test Case 1 - Dynamic Factorial

To initially test the remote class loading structure we needed to be able to determine the exact moment a class that does not reside on a host system is called. To do this we first used a simple Factorial Program. The Program consists of two classes, factorial.FactorialGui.class and factorial.Factorial.class
The time of the generation of the second class can be predetermined as it is based on a user triggered event from the GUI. This then allowed us to determine an exact moment when the remote class loading structure was being used.

5.2 Specifications

Class factorial.FactorialGui consists of AWT components including a Button and Textfield. The event triggered by a button press instantiates an object of factorial.Factorial, first calling the base constructor then calling the factoriate(int num) method. This will return the factorial value of the integer passed in as an argument. Once this method is completed the result is displayed within the Textfield of the GUI.
The same program was then executed on two separate machines, both using the remote class loading structure and connected to the same network. On one of the systems we left all files as standard, in running order, whilst on the other we deleted the factorial.Factorial.class file. The system will be unable to find this class locally, when it is requested from the Button Event, and the remote class loading structure will therefore have to look elsewhere for it.
We then observed that it located the second remote class loading structure (as advertised from the second system) and initiated a search for the factorial.Factorial class bytes within the reach of the second classloader's JVM. Remembering that factorial.Factorial only exists on the second system, as detailed earlier, we observed that the application found, read and sent the corresponding bytecode back to the searching classloader. The searching classloader received the bytes and was able to load and initialise them as normal.
There was no hard save of the class file to the receiving system, that is, factorial.Factorial is never actually saved to persistent storage on the searching system. Therefore, once we closed each system completely, we found that each system's file structure remaind unchanged from when we had originally set them up.
Upon restarting the searching system again, in the exact same way, we were again unable to find Factorial.class locally. However, on this occasion the second system was not running, and subsequently the first system could not find the missing class file. A ClassNotFoundException was thrown and appropriately handled via the remote class loading structure.

5.3 Results

As an initial test-bed the Factorial program demonstrated that the remote class loading structure was able to download and instantiate a class that it did not initially contain. The design of the remote class loading structure not only allowed the Factorial program to access and instantiate a class file that did not exist (factorial.Factorial), it also reflected the initial objectives of our research through its demonstrated flexibility and dynamic nature.
This test was an initial test and further development of a more *useful* example allowed us to view the useability and practicality of the technique. We also needed to test the affects of the loading of a class into system that had no prior knowledge that it was going to be loaded.

5.4 Test Case 2 - Tourist Information Service

Based on a simple tourist system we developed a small program that uses our class loading structure for a specific purpose. The system allows visitors to view a destination specific application using the same program. Furthermore, there are no restrictions to the type of application the tourist destination is able to provide.

5.5 Specifications

The tourist application was designed using an initial start up class (tourist.Tourist.class) that then generates an instance of another class (tourist.TouristInit.class). Although, a Tourist object is the initial instantiating class it is not the most productive in operation, this function belongs to the tourist.TouristInit class. Therefore, tourist.Tourist.class (along with the remote class loading structure) is the only file that is needed on a users system. This class being extremely small in design would easily fit on a resource limited device, furthermore in this example the design is such that the rest of the program resides somewhere else within the network.
The Tourist system application has been designed to be used using a central server which is based at each tourist site. Although, this is not necessarily where each system will gather all their files from as different tourist destinations are able to write their own representation of the Tourist system. Whereas, users only need to use the same Tourist program in order for then to view each destination specific program. As a user starts the Tourist program the remote class loading structure will take over the loading of the classes. Due to this all classes can be found from either, a *central server* as setup by a tourist destination, or, a *separate user's system* that has utilised the program and advertised their own classloader. The system is based within a networked environment and is therefore able to access all network machines, including web servers and other file loading techniques. These may also be used by the designers of specific tourist systems to facilitate access to images and information specific to their program (for example

photographs of the tourist destination). Our test example Tourist program used this type of architecture to gather images that are loaded into remotely accessed classes.

5.6 Results

Similar to the initial testing results of the Factorial example, the tourist destination proved to work efficiently and correctly as intended whilst running within the remote class loading structure. The Tourist program was designed to test the flexibility, in terms of conventional programming techniques, file access, and event based changes, of the remote class loading structure. We found that the extensibility of the structure in terms of location discovery and multiple class downloading allowed the Tourist program to be used as intended by a client. This operation also included giving access to images and information from a location that was initially unknown to the client.
A target tourist program was designed that loaded multiple classes that were unknown to the client system, whilst also accessing remote picture files via a HTTP server. All of which proved successful.
The results obtained in testing the Tourist program showed that the technique can be applied to more meaningful applications. In this case the system worked much like a web page, yet allowed the user to not only view information but also generate an entire program.

6 Observations

The remote class loading structure's use of Jini 1.1 [5] gives it a dynamic method of accessing classes. While the internals of the classloader allow it to download files simply and efficiently. Some methods of downloading are reminiscent to those of a standard web browser, that generally follows these common steps.

1. Find and load and initial file (.class or .html).
2. Gather further information as is needed for the initial file to proceed (pictures, further classes/pages, documents etc).
3. Continue operation until user closes application, never actually writing anything to the users drive (cookies and caching can be excluded from browsers as these may not be considered mandatory for its operation).

It is in the combining of the computational power of Java with the dynamic adaptability of the remote class loading structure that gives this technique a unique difference. Yet as is often the case, with extra power and flexibility also comes a greater risk of attack. Currently there is no security measure that can, efficiently, stop a would-be attacker replacing a searched class (unavailable locally) with a spoofed version and thus interfering with a users system. The gateway section of the loading structure may be used to rectify this situation in the future, as this is one section that has been left open for further advancements of the structure.
The design of the remote class loading structure allows Java applications to share files, data, and design structures whilst also allowing the designers to maintain total control over their work. Furthermore, applications may become so diverse that they offer spontaneous, or selective, version updates creating user established environments.

7 Conclusion

Remote class loading via a dynamic discovery technique, such as Jini [3, 5], is a unique concept that gives access to Java class files within an environment.
The discovery of remote classloaders produces a peer-to-peer feel in accessing of the internals of their associating JVM's. Allowing any Java program to access, and give access to, Java class files within the environment.
The testing of our remote structure demonstrated its usefulness within a wireless network, extending the flexibility of targeted programs whilst handling multiple remote classloaders and devices. AccordinglyThe structure has proved successful As a basis for establishing a unique technique in obtaining a dynamic spontaneous link between systems.

7.1 Future Work

Continual research into aspects of the technique encapsulates areas such as, security, performance, multiple platforms and proxy manipulation. As discussed within this paper security issues may initially be a priority, focusing on the structure's flexibility and efficiency.
Further testing and analysis of the structure's negative effects on a discovered system are planned, however, at present these appear to be minimal.

References

[1] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification. second edition* (Reading: Addison-Wesley, 1999).

[2] Java 1.4.1 api. *http://java.sun.com/j2se/1.4.1/docs/api/*

[3] J. Newmarh. *A Programmer's Guide to Jini Technology.* (New York: Apress, 2000).

[4] OSGi - Open Services Gateway Initiative *www.osgi.org*

[5] W.K. Edwards. *Core Jini. second edition* (Upper Saddle River: Prentice Hall, 2000).

[6] Dynamic code downloading using RMI *http://java.sun.com/j2se/1.3/docs/guide/rmi/codebase.html*

[7] Java Web Start *http://java.sun.com/products/javawebstart/*