

An Architecture for Component Evolution

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

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

Abstract—In an evolving object oriented system individual components may change. A system built out of such components needs to be able to use the most recent versions of these components no matter what their source.

This paper presents an architecture in which an application will select the most appropriate component version from a variety of sources, including network discovery. The architecture is based on the dynamic class loading mechanisms of Java. It succeeds where current techniques fail due to lack of interoperability and inability to adapt to dynamic environments.

The architecture allows an application to dynamically load components from a variety of sources including, local disk, remote service, and personalised techniques. It chooses the most appropriate component by using version control information. Security policies may be used to restrict the actions of components and the policies can be refined as new components are loaded. This ensures that an application can run with zero configuration but be continually updated in a safe manner. We have built a prototype system and show by example how this can be used by a home gateway to allow software upgrades with zero home configuration.

I. INTRODUCTION

Software in the field needs to be upgraded when new versions or bug fixes are made. The multiple versions of (for example) browsers that still exist on home computers demonstrates that this is a non-trivial problem. Current techniques used include .dll files and downloaded upgrades. For example, during initialisation some operating systems such as a Windows XP and Linux initiate searches for updated versions of software on a home site. In a distributed mobile or zero configuration environment these fail because of their reliance on a need for update location knowledge.

In a previous paper [1] we presented a system whereby a Java application could pick up code from its environment using an extended dynamic class loader. This was used to customise the application to its environment. For example, a tourist application would load code appropriate to its location. In this paper we extend this technique to address the problem of system evolution. We show that a class loader can be devised that will allow an application to find the latest version of a class from its network environment. This is a better way of achieving system evolution than previous techniques. The novel contribution of this paper is an architecture that allows component evolution via a variety of sources, including network discovery.

The technique is demonstrated for Java, but is applicable to

any runtime environment that supports dynamic loading of code. It is particularly suited for languages such as Java which support customisable class loaders, but could even be used by operating systems with suitable modifications to dynamic link loaders.

Based on security settings and version requirements an evolving system is able to “pick and choose” which version of the component is appropriate. Spoofed versions cannot be avoided, but are restricted in the damage they can do by use of security policies. The demonstration implementation of our architecture gives access to Java class files, where each class file represents a component, downloadable from an *unknown* system. We do not address the issue of how the updated classes are placed into a network environment, but possibilities such as home gateways are discussed in section VI.

Section II presents a brief outline of the concept of system evolution. Section III details the concerns associated with evolution. Section IV details how Java enables the creation of a prototype. Section V is a look at the implementation aspects of the architecture. Following in sections VI and VII is a detailed example scenario and analysis. We conclude in sections VIII and IX with related works, conclusion, and future works.

II. SYSTEM EVOLUTION

System alteration can be initiated and completed via distributed techniques, including the ability to upload remote classes or components [2]. Such structure changes may then allow systems to evolve according to their environment. The discovery based nature of distributed techniques, such as Jini [3], allow component discovery to be dynamic. Thus particular components may be unknown during initial implementation stages. All systems conforming to our middleware architecture will have the ability to evolve, even if they were not originally designed to evolve in this manner. The following sections detail several key attributes of our architecture; (1) no need to create objects in order to locate class files; (2) no need for hardwired addresses in order to locate class files; (3) an ability to find class files locally, on remote systems, or within local components of remote systems.

A. Remote Object Manipulation

Distributed techniques, such as Jini, RMI and CORBA [2], make use of remote object manipulation. These are ideal for remotely accessing server sections that exist for the use of

clients and the sharing of information. Its use solely as a means of evolution is impractical, as it would require interfaces for all for possible components. However by specifying the methods of a remote object, restrictions on processing performed by a client can be established.

In distributed systems that use a marshalling technique there is no passing of class bytes until an object has been discovered. The newly discovered object, for example, a marshalled object, then points the client system to the actual class bytes for initialisation [3]. This requires that an object be created first in order to be marshalled and that a copy or reference to this object is created in remote systems. If it is just a matter of finding the bytes defining a class, then it should not be necessary to create an initial object in order to do this.

B. Static Component Access

Web based programs do not use remote objects instead they generate programs based on local access to remote code. This gives them the ability to run program code that initially did not exist on the local system. Java applets, for example, use a specific classloader designed to direct clients back to a service located on a server. 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 [4]. Java's Web Start also allows a client to generate a program via web based links [5].

Although there is a lack of dynamism and location unawareness, the simplistic method in which the class files are obtained is conceptually applicable to an evolution technique. It is unfortunate however that a client must know the exact server location to locate class files.

Web Start contains checking mechanisms to analyse versions via jar files [5]. This allows it to check the version of an application each time a user implements it. However, Web Start still has the restriction of using a URL type link. Although it is able to recreate previously run applications, it must still know exactly where to get the application from.

C. Client Internal Access

The evolution of a system is greatly aided by the discovery of components from within a network. However, an architecture based on discovery must be able to discover local as well as remote components. This becomes further apparent when the architecture is middleware based. In this situation each system has the ability to supply others with their version of a component. To return the component, or even look and see if they contain it, a system must have the ability to search within itself. Internal sections within reach of a system should include the JVM class structure, classpath components, and other areas specific to that system, for example, codebase servers.

III. EVOLUTION CONCERNS

The discovery of unknown components presents the possibility of the intrusion of malicious components during the

evolution of a system. Our architecture must therefore contain a means for determining the difference between wanted and unwanted components.

A. Component Versions

Version checks enable systems to evolve according to new updates available within reach of a system. This is helpful in the removal of errors and the addition of extra features to a system. Furthermore, as the evolution of components is an internal linking process, a change in version of one component can lead to the need for a new component, or the update of another. The evolution process as a whole will then allow systems to adapt or evolve based on the components found within the environment.

Version manipulation within Java systems is generally achieved through use of a jar file version attribute, as used in for example Web Start [5]. Our architecture is based on the discovery of other compatible systems, and therefore component discovery is not necessarily confined to jar files. The architectures classloading semantics search through the JVM, including the classpath and personalised loading techniques, making it unrealistic to restrict component discovery to jar files. However as most java applications are initially within a jar file the version attribute can be assigned and gathered on the initial loading of system components. This makes version identification available during any type of component gathering. This includes:

- Loading via classpath: As the program is initially contained in a jar file(s) when each class is discovered its associated version is also obtainable. A system is therefore able to easily gather this information and attach it to any requests for components.
- Loading via JVM: At times it may be best to load a component directly from the source systems JVM. This system has obviously loaded the component at some stage itself storing and linking the version number. This version number is then easily accessed if a system ever wishes to make use of the source systems component.
- Personalised Loading: As a target system has the ability to load classes in its own unique manner, the architecture must force them to provide a version number. Most conceivable personalised loading techniques, for example URL based class loading or internal search class loading, make use of jar files. It is therefore possible, as the architecture is based on an initial common classloader to force all mechanisms to provide the version number of the initial jar file.

During the execution of a target system the continual upkeep and referencing of the component versions is essential to their evolution. Moreover, this is also an essential component in the security aspects of the architecture.

B. Security Settings

During the evolutionary alteration of a system all discovered components may be included and executed at some period of time. Therefore 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 includes possibility of complete system deletion.

Our architecture takes the same approach to security that is used within many Java distributed techniques [2], [6]. Security policies are implemented that clearly define the degree of access a client has to a system. As this architecture's system evolution is based on the retrieving of components from unknown sources, its version mechanisms play a vital role in aiding security concerns. The coexistence of version controls and access security form the basis of our implementation of security measures for the architecture. The version aspects control sequential evolution requirements. The security mechanisms control the degree of access a system will allow another system to gain.

The architecture is open to many different types of attacks that may be used to gain illicit access to a system. For example, harmful code may be masked under a class name that is assumed safe by a system.

The complete discussion of the security aspects of the architecture is large and thus beyond the scope of this paper. It will therefore be addressed and analysed in a future publication.

C. Context of Change

The context awareness capabilities of a system may be used to determine how they react to environmental variables. The architecture presented in this paper is not a context aware technique. There is therefore a need to determine what alterations should be based on. As the evolution of a system can be deciphered as a progressive change in a system, it is reasonable to assume that changes may occur in an incremental manner. That is, components should have corresponding version numbers enabling the system to determine whether a change should take place or not.

However, there are many different reasons for change, and therefore, there may be cases where version based change is not applicable. For instance, a component may wish to evolve in accordance with the version that is commonly used within a specific environment, be it higher or lower. These such cases may rely on particular settings within the versioning and profiling of the architecture, and should be selective to the implementing system.

IV. ENABLING JAVA TECHNOLOGIES

Our architecture needs to locate class files without prior knowledge of their location address, be it local or remote, in a safe manner. We have implemented the architecture using Java. The following sections detail the techniques used to enable the architecture to achieve the above needs.

A. Initialising an 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 (RCLS) [1]. The RCLS can be used as the underlying loading mechanism for any Java based

program. This gives the architecture even greater flexibility and useability as there is no limitation on the type of program that can gain from its techniques.

The architecture is able to control the entire loading structure of the system. It specifies that all classes must be loaded through the same remote classloader.

B. Java Classloading

Java's class loading is optimally designed to aid in the dynamic linking and loading of class files [7]. 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 [7]. The hierarchical structure that is produced when classes are loaded into a JVM allow the redefining of an applications loading techniques. As classes are located, loaded and linked via a classloader any subsequent classes will be loaded via the same class loader [7]. In practice this allows program designers to load an initial class through a specialised classloader, thus loading the entire system through this class loader. If the same system then loads another different classloader the additional classloader will also be loaded through the first classloader. If the second classloader fails then all class processing will be delegated to its parent classloader and so forth [7]. When applied to a middleware concept a target system is run via a consistent class loading mechanism.

C. 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. A programmer is able to alter the structure of these methods to achieve a specific task, redefine how the loading structure intercommunicates and operates [7].

This allows classloaders to be defined that can discover other classloaders, and so extends the reach of classes that can be loaded. When a class is required, it can be downloaded from any classloader that has been discovered. The definition of specific methods is then used to maintain the separation between a classloaders normal and remote operations.

D. Component Discovery

The RCLS is the underlining discovery class loading. Ignoring all security or versioning the RCLS is able to load any component from unknown sources [1].

E. Interlinking the Virtual Machine

There are now several methods to tap into a systems virtual machine running state [8]. Traditionally Java gives us the use of reflection. Even through its usefulness is limited, when used along with class loading techniques it gives an ability to load and watch classes. This technique has been used to load classes through the architecture into a JVM. This is achieved in a very similar fashion to [1]. The architectures initial class is used to load and link the initialising class of a target system.

This forces all following classes to be loaded through the architecture.

F. Security

Java's Security Manager and Policy details secure many distributed techniques. It can be applied to Jini, RMI and Applets applications through use of the security API [6]. As our architecture utilises many characteristics of a distributed technique it is able to apply security measures in much the same way.

The policy details are used to determine the access to a system and its associated files. This is important due to the discovery nature of its remote class loading capabilities.

The accessible nature of the architecture deems it extremely susceptible to many unwanted attacks.

Although the full details of security are too large for this paper as a general guide the use of policy details within the architecture gives certain systems access to specific files, and specific files access to certain sections of the system. For example, a system is able to gather classes remotely and blindly, yet does allow the generation of new class loaders or the alteration of policy files.

G. Version Controls

Jar file manifest details provide a means of versioning. Our architecture makes use of this feature to determine the version of discovered components.

Generally the manifest is used to describe the version and other specific details, such as vendor, of an entire package. The use of multiple names within the manifest allow it to describe the version of a specific class. Our architecture exploits this by allowing the use of general version details and specific class version details. General version details apply to all classes *except* those signed with a specific version detail.

For example: `Duck.jar` contains 4 java class files (`duck.Huey`, `duck.Dewey`, `duck.Louie` and `duck.Donald`). The status of each file is determined by the manifest file `Duck.mf` (Figure 1). In `Duck.jar` the

```

Manifest-Version: 1.0
Name: duck
Specification-Title: Ducks
Specification-Version: 2.0.0.01
Specification-Vendor: Adrian Ryan
Implementation-Title: Duck1Melb
Implementation-Version: duck1.0078
Implementation-Vendor: Ryan

Name: duck.Donald
Specification-Title: Ducks
Specification-Version: 2.1.0.02
Specification-Vendor: Adrian Ryan
Implementation-Title: Duck1Melb
Implementation-Version: duck1.0078
Implementation-Vendor: Ryan

```

Fig. 1. `Duck.mf`

version number (we use `Specification-Version`) for all class files is `2.0.0.01`. Except for `duck.Donald` which has its

own version specification, `2.1.0.02`.

This feature allows the architecture to determine the specific version of each class. It will then store it ready for retrieval when discovered by a separate system.

V. IMPLEMENTATION OVERVIEW

The following is an overview of the architecture's implementation, design and construction.

A. The Architecture

Figure 2 represents the internal associations and processing links in detail showing how the architecture's discovery of components link within its structure. It details the communication, flexibility and internal constructs of the architecture. The pathways of internal communication are indicated by bridges and gaps that interlink each component together. These links allow the architecture to control the processing path of the targeted JVM (shown at the bottom of figure 2) and thus determine its reasoning for class initialisation.

Combining the computational power of Java and the dynamic adaptability of the RCLS [1] gives the architecture its base structure. However the extra flexibility given to a system by the architecture opens it for risk of unwanted intrusion. The architecture deals with this through the use of version controls and security measures. The structure for security purposes deems that all external loading, with the exception of the sending of components to other systems, must be achieved through security and version controls. This gives confidence to the architecture in terms of safety from malicious, unwanted, components. The architecture's middleware design allows Java applications to share files, data, and design structures whilst also maintaining that system designers have total control over their work.

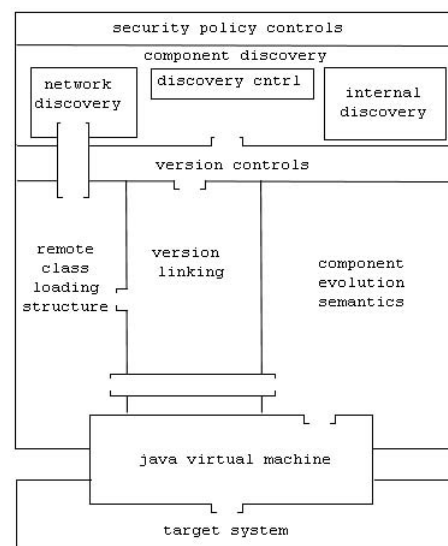


Fig. 2. Internal Structure

B. Remote Classloading

Using concepts derived from both remote objects and web based classloading, the architecture allows systems to load classes via unknown remotely accessed classloaders [1]. This enables system to share components, finding and using them without prior knowledge of their whereabouts. Remotely accessible components give systems a means to change according to their needs and the needs of the environment. This particular attribute will prove extremely useful for evolution within limited memory devices.

C. Remote Discovery and Advertisement

The architecture is designed so that each compatible system discovers and advertises. Therefore, each will advertise itself as an available service, whilst also having the ability to discover other services when desired in a peer-to-peer manner. A key to the architecture is the dynamic discovery of remote classloaders. This allows applications restricted access, through a distribution technique, to the actual inner sections of a remote JVM. The trigger for the discovery of these components comes from the need to instantiate the class within the JVM. It would be demanding for a system if a classloader search was initialised each time a new object was created. The dynamic classloader discovery technique provides the architecture with two of its crucial aspects.

- 1) The discovery of unknown components. Discovery techniques allow services to find each other without the need for address knowledge.
- 2) The transfer of byte code through any remotely accessible compatible system. Every component that is searched for by a JVM does not need to be located and loaded via the same system.

These unique attributes give an ability to create applications that are more flexible and adaptable within environments. Furthermore, the inclusion of specific version and security mechanisms detail the extent to which a system will evolve.

D. Limitations

All components must have been, at some stage during their discovery, found from within a Jar file. Moreover, this Jar file must have an associated Manifest file detailing the version status of all class files within it. Without this detail the version of components cannot be set and the architecture is unable to determine correct updates.

Jini requires several separate components to work [2]. Although all of these do not have to exist on the same device the need for them still restricts the architecture to certain devices and systems.

Performance is the most obvious limitation that the architecture adds to a system. When it searches, discovers and downloads a component there is a considerable lag period. This lag time is variable as its duration is in proportion to common distributed problems, such as speed of discovery, location of discovered service and download speed.

VI. EXAMPLE SCENARIO

In [1] we considered a tourist application example. This demonstrated the use of remote class loading concepts in a highly dynamic environment. However, this testing scenario is not complex enough to detail the current evolution nature of the architecture.

The following is a scenario which makes full use of the architectures nature.

A. Home Automation Service

Home automation services are made possible through the use of techniques such as, OSGi [9] and UPnP [10]. Each technique gives services access to external information, including the internet via a gateway. This example is based on the concept that each home service, including the gateway, is running the architecture and is able to evolve as the home system grows.

B. Specifications

A home automation system is designed to allow all services, in most cases these being appliances and devices, to interact and use the capabilities of others. Examples of its use include: phone call notification - a user is informed of a phone call via the television; device interaction - a user using the coffee machine will be informed if there is milk in the fridge or not. The evolution of these appliances, or systems, is static if not impossible. One lacking need is the ability to update appliances on the inclusion of a new system, or the upgrading of a systems current semantics.

Our architecture enables the evolution of systems by having the current version running on each system and in accordance with the home system provisions. The evolution of a system could be initialised via several different situations, these are detailed as.

- *Evolution via Addition of a New System.*

The object oriented (OO) nature of components deems them reusable. It is therefore not necessary that component based evolution be achieved via the same type of system. For example, a toaster and a coffee machine have nothing physically in common, yet their internal semantics are likely to have similar components. Each communicates with the gateway and the fridge. If coded appropriately, a toaster should be able to evolve several of its components through the addition of a new coffee machine.

- *Evolution Triggered by the Vendor*

There are many cases where vendors release updates for device systems. These contain changes to a small number of certain system sections, or components. This evolution change is based on a version update. In home automation a vendor should be able to remotely place components on the home gateway for the systems within the house to discover and evolve (according to security and version constraints). Furthermore, as our architecture enables systems to use components found within themselves, within their normal system reach, or, via

discovery, a component could be placed anywhere in the home network and a system would behave accordingly. This includes a visiting, temporary, system providing the environment with new capabilities.

- *Linked System Evolution*

The previous two situations can be extended to achieve a complete home system evolution. In fact, it is likely that if a new version of a generic component, for example communication (such as new version of a communication module introducing a new network protocol), is placed somewhere within the home environment, one by one each system would discover it and evolve. Each separate discovery of the component need not be achieved via the same source system. All systems are able to receive and send components. Thus as the generic component is applied to each system the number of places it can be discovered from increases.

VII. ANALYSIS

The home automation system represents an application that would benefit from the architecture. It demonstrates several advantages in the alteration of system components, to the degree that evolution is, at times, unavoidable. Due to the combination of component OO methodologies and component evolution the flexibility of systems is greatly enhanced.

The *Evolution via Addition of a New System* scenario best depicts this. In this case the architecture allows components to be discovered, according to security and version controls, from systems that generally have nothing in common. We can see from this that the ability to discover components and use them for evolution is more flexible than initially perceived.

As mentioned previously the architectures additional performance time is difficult to measure. The running of the scenario showed a lag time is present. However any distributed technique involves considerable network overhead, our architecture is no exception.

VIII. RELATED WORKS

We presented a unique architecture to allow the evolution of components. However, there are several existing techniques which are similar in nature.

Columbia is a dynamic component upgrade technique [11]. It is designed as a middleware and is governed by a clearly defined Meta-Data (XML). Being continually dynamic Columbia is complex. We see that it is restricted by its security based properties and use of context variables. Furthermore, Columbia requires an XML parser in each device, which is heavyweight and requires the system installer to produce a complex configuration document.

Many operating systems achieve updates triggered 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. On the other hand, our system runs with minimal configuration and only requires version status in the jar files.

Version control techniques for file management, such as RCS

[12] and SCCS [13], 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.

The architecture presented within this paper is aimed at mobile devices. 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.

IX. CONCLUSION

Discovery based remote classloading is a unique concept that gives remote access to Java class files. The extension of such a technique enables component evolution and system adaptation. As demonstrated by the example scenario, the presented architecture is not only advanced in component evolution but applicable to every day situations. Furthermore, due to the architecture's structure the evolution of a component within an environment, such as a home network, is contagious. Through the discovery of compatible systems and access to their semantics, the architecture provides a means to evolve systems based on new versions of components. Components which may be discovered from within themselves or from within the environment.

A. Future Work

System evolution is a complex task. Although this architecture aids in the discovery of components whilst providing security measures we have not analysed the concept of runtime component evolution. The elements of such a technique are a large topic and further work needs to be achieved in this area.

REFERENCES

- [1] 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.
- [2] J. Newmarch, *A Programmer's Guide to Jini Technology*. New York, USA: Apress, 2000.
- [3] W.K. Edwards, *Core Jini*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2000.
- [4] "Dynamic code downloading using rmi," accessed April 2004. [Online]. Available: <http://java.sun.com/j2se/1.3/docs/guide/rmi/codebase.html>
- [5] "Java web start," accessed April 2004. [Online]. Available: <http://java.sun.com/products/javawebstart/>
- [6] L. Gong and G. Ellison, *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*, 2nd ed. Reading, MA, US: Addison-Wesley, 2003.
- [7] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Reading, MA, US: Addison-Wesley, 1999.
- [8] "Core java j2se 5.0," accessed Oct 2004. [Online]. Available: <http://java.sun.com/j2se/1.5.0/index.jsp>
- [9] "Osgi - open services gateway initiative." [Online]. Available: www.osgi.org
- [10] "Upnp," <http://www.upnp.org>. [Online]. Available: <http://www.upnp.org>
- [11] R. M. Paolo Bellavista, Antonio Corradi and C. Stefanelli, "Dynamic binding in mobile applications: A middleware approach," *IEEE Internet Computing*, vol. 7, no. 2, pp. 34–42, 2003.
- [12] W. F. Tichy, "RCS — a system for version control," *Software — Practice and Experience*, vol. 15, no. 7, pp. 637–654, July 1985.
- [13] M. J. Rochkind, "The Source Code Control System," in *Proceedings of the 1st National Conference on Software Engineering*. IEEE Computer Society Press, 1975, pp. 37–43.