Class Loading Issues in Java™ RMI and Jini™ Network Technology

Michael Warres

Class Loading Issues in Java™ RMI and Jini™ Network Technology

Michael Warres

SMLI TR-2006-149 January 2006

Abstract:

Java class loading plays a key role in the Java Remote Method Invocation (Java RMI) and Jini architectures by enabling code mobility over the network. However, it has also saddled these architectures with a set of type compatibility and code downloading issues that commonly result in run-time errors and programmer confusion. This paper describes the Java RMI class loading model and examines its ramifications.

Sun Labs
16 Network Circle
Menlo Park, CA 94025

email address:
michael.warres@sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>.All technical reports are available online on our website, http://research.sun.com/techrep/.

Class Loading Issues in Java™ RMI and Jini™ Network Technology

Michael Warres
Sun Microsystems Laboratories
michael.warres@sun.com

Abstract

Java class loading plays a key role in the Java Remote Method Invocation (Java RMI) and Jini architectures by enabling code mobility over the network. However, it has also saddled these architectures with a set of type compatibility and code downloading issues that commonly result in run-time errors and programmer confusion. This paper describes the Java RMI class loading model and examines its ramifications.

1 Introduction

Class loading is a powerful and distinguishing feature of the Java platform. It enables Java processes to load application and system code at run time from a variety of sources, such as the local file system, a remote web server, or an in-memory buffer. It supports many diverse uses: remotely published Java applets, pluggable application containers, code instrumentation, on the fly software upgrades, and run-time code generation, among others.

Two interrelated architectures that heavily utilize class loading are Java Remote Method Invocation (Java RMI) and Jini network technology. Java RMI provides the ability to invoke methods on objects hosted within remote Java processes. Jini extends the Java RMI programming model to support a more full-fledged notion of distributed *services*, incorporating both client- and server-side computation, that can be discovered and used in an ad-hoc manner over the network. Both Java RMI and Jini fundamentally rely on passing Java objects between Java processes, where the executable code for a transmitted object may not be preinstalled in its destination. Supporting this form of object transfer requires dynamically loading the class definition for the object into the receiving process. Without this functionality, supplied by Java class loading, neither Java RMI nor Jini would be possible.

At the same time, class loading lies at the root of a set of abstruse type compatibility and code downloading issues that can significantly complicate development and deployment of Java RMI- and Jini-based systems, especially in cases involving multi-party service interactions. This document attempts to

enumerate and explain these problems. It is structured as follows. Section 2 gives background on class loading in Java. Section 3 describes the basic RMI class loading model, which is also employed by Jini. Section 4 examines the issues that follow from Java RMI and Jini's use of class loaders. Section 5 discusses *preferred classes*, a variant RMI class loading scheme that addresses some of the issues raised in the preceding section. Finally, Section 6 analyzes the overall impact of RMI class loading issues.

2 Class loading

This section gives a basic overview of Java class loading; for further details, refer to [1, 2, 3].

2.1 Basic model

Concretely, class loaders are instances of subclasses of the abstract `java.lang.ClassLoader` class. Their primary purpose is to obtain class definitions (i.e., code) for classes whose names they are passed.[1] This function is performed by each class loader's `loadClass` method, which takes as input a class name, and returns a `java.lang.Class` object representing the loaded class. Inside the class loader, this translates into several steps: first, the class loader checks if it has previously loaded a class of the given name, and if so, returns that class.[2] If no such class has been loaded yet, then the class loader obtains (through means specific to itself) a definition for the class, formatted as a Java class file. Finally, the class loader reifies the class by *defining* it—passing its definition to the Java™ Virtual Machine (JVM), which hands back a representative `Class` object.

Class loading is the initial step of a larger process through which a class is readied for execution. After a class is loaded, it must undergo *linking* and *initialization* before other classes can call or instantiate it. These steps are carried out by the JVM in between the defining of the class and the class's first use, though they may not take place immediately—the JVM may elect to perform them lazily, only as needed. Linking involves verifying the class definition's bytecodes for structural validity, and resolving symbolic references to other classes, which may in turn necessitate recursive loading of the referenced classes. Once linking is complete, the JVM initializes the class by invoking its static initializer method and any initializers for static fields, if present.

Class loading occurs initially during program startup, but may also be invoked throughout program execution—either explicitly, in response to requests from application or library code, or implicitly, to (lazily) resolve symbolic references. The JVM caches the results of `loadClass` invocations on class loaders,

[1] In this paper, the term "class" encompasses Java interfaces as well, unless noted otherwise.

[2] This step is expected, but not directly enforced. The JVM will, however, throw a `java.lang.LinkageError` if a class loader resolves a given class name to more than one class over time.

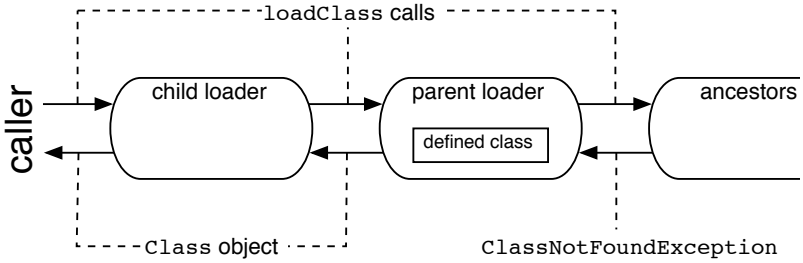

Figure 1: Class loader delegation

both for performance, and to guard against inconsistent return values, which could otherwise compromise type system integrity [4].

Each Java process uses multiple class loaders. A class loader typically obtains class definitions from a fixed set of sources, such as a URL, or the enclosing process's class path. Class loaders cooperate to load application and system classes through hierarchical delegation—each class loader holds a reference to a *parent* loader that it calls upon when asked to load a class unknown to itself.[3] For reasons explained later, delegation is generally tried first—most class loaders will only attempt to define classes if delegation to their parents fails.

A natural consequence of class loader delegation is that classes defined in a parent loader are visible through class resolution to its child loaders, and may be referenced by classes defined in those loaders. Thus, class relationships may span class loader boundaries: for example, a class C_1 defined in class loader L_1 may subclass from class C_2 defined in class loader L_2, provided that L_2 is an ancestor (i.e., direct or indirect parent) loader of L_1.

Because loading a given class may involve multiple class loaders, some additional terminology is useful to distinguish their roles. As established in [2], the *initiating loader* of a class is the class loader that is originally requested to load the class, even if the request is ultimately fulfilled by another class loader through delegation. The *defining loader* of a class is the class loader that actually obtains its definition and defines it. We will sometimes describe a class using the common notation $\langle C, L \rangle$, where C is the name of the class, and L is its defining loader.

2.2 Well-known class loaders

The following class loaders are present in each Java process running JDK™ 1.2 or later [5]:

[3] It is technically possible for a class loader to delegate to multiple "parent" loaders, though doing so is unorthodox, and increases the risk of violating loader constraints (described in [2]). Such a class loader, however, must still designate one of its delegation targets as the nominal parent loader to be returned by the `ClassLoader.getParent` method.

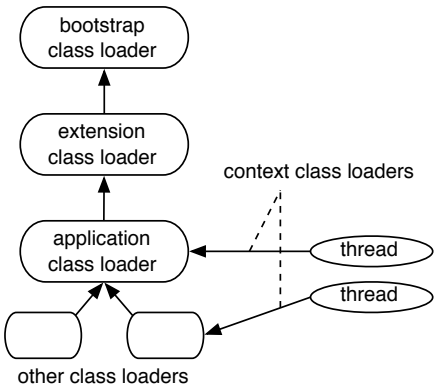

Figure 2: Well-known class loaders

Bootstrap class loader. This class loader loads system classes, such as those belonging to the various `java.*` packages. Where and how these classes are actually stored may vary between JVM implementations. This class loader is also known as the *primordial* or *null class loader*.[4]

Extension class loader. This class loader loads classes from Java™ Archive (JAR) files [6] present in the standard extensions directory [7] for the Java process.

Application class loader. This class loader loads classes from within directories or JAR files specified by the class path of the Java process. As this is the default class loader returned by the `ClassLoader.getSystemClassLoader` method, it is sometimes called the *system class loader*.

These three class loaders are arranged such that the application class loader delegates to the extension class loader, which in turn delegates to the bootstrap class loader, as pictured in figure 2. As a result of this hierarchy, system classes are visible (via class resolution) to extension classes, and both system and extension classes are visible to application classes.

One other commonly used class loader bears mention: the *context class loader*. This is not a distinct class loader instance, but rather a mutable thread-local variable provided by the `java.lang.Thread` class. The context class loader for a given thread represents a suggested class loader for code running in that thread to use to load classes or resources. It is typically consulted by library code that needs to explicitly load classes on behalf of the current thread. For example, a utility method might perform (roughly) the following operations to

[4]The bootstrap class loader used to also be called the system class loader (for example, by [1]), a term that now, somewhat confusingly, refers to the application class loader.

resolve a class name:[5]

```
Thread thread = Thread.currentThread();
ClassLoader ccl = thread.getContextClassLoader();
Class resolved = Class.forName(className, true, ccl);
```

The context class loader is necessary because library code cannot generally infer the appropriate class loader to use to load classes for the current thread. Library code cannot rely on its own class loader to resolve all class names it encounters, since the library code may be defined in an ancestor class loader, such as the bootstrap or extension class loader, from which the desired classes (perhaps belonging to the application) are not visible. The application class loader is not always sufficient for loading these classes, either: for instance, if the thread is executing a downloaded applet, then it may need to resolve applet classes defined in a separate, applet-specific class loader. Because the classes to be resolved, in the general case, may be defined in an arbitrary class loader unknown to library code, this class loader must be supplied explicitly. The context class loader presents a standard means to convey this value.

2.3 Other roles of class loaders

Class loaders not only provide class definitions to the JVM, but also determine various aspects of the classes they define:

Type identity. Class loaders create new namespaces for types. The compile-time type system of the Java programming language is name-based only: each class or interface name unambiguously designates a single type. The Java run-time environment, however, uses a type system with an added dimension, in which type identity is based not only on the name of a class, but also on its defining class loader. Thus, if class loaders L_1 and L_2 each define a class named C, then the two classes constitute separate types even if the class files on which they are based are identical—in other words, $\langle C, L_1 \rangle = \langle C, L_2 \rangle$ iff $L_1 = L_2$. Attempting to assign an instance of $\langle C, L_1 \rangle$ to a variable of type $\langle C, L_2 \rangle$ if L_1 and L_2 differ will cause a `java.lang.ClassCastException` to be thrown.[6]

Note that although $\langle C, L_1 \rangle$ and $\langle C, L_2 \rangle$ are separate types, they can still interact through supertypes they hold in common. For example, if both types implement the same interface $\langle I, L_{parent} \rangle$, then instances of either type can be assigned to variables or fields of type $\langle I, L_{parent} \rangle$. However, any common supertype must constitute a single type—if $\langle C, L_1 \rangle$ and $\langle C, L_2 \rangle$ were to implement $\langle I, L_1 \rangle$ and $\langle I, L_2 \rangle$, respectively, then their type incompatibility would simply extend a level higher.

[5] Realistically, the utility method would probably nest the `getContextClassLoader` invocation within a call to `java.security.AccessController.doPrivileged`, so as not to require callers to have permission to access the context class loader.

[6] Type conflicts of this sort rarely arise in standalone applications, since all classes used by such an application are usually defined along a single, non-branching delegation chain of class loaders, leaving no opportunity for duplicate definitions.

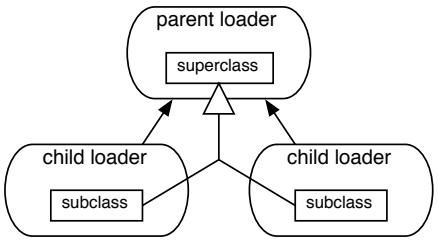

Figure 3: Type compatibility through a common supertype

The namespace-defining property of class loaders is sometimes used to provide a cheap approximation of subprocesses—this was in fact one of the original motivations for tying types to class loaders. As classes defined in different class loaders constitute separate types, they accordingly carry separate copies of class state (i.e., static field values). Therefore, classes defined in a class loader L_1 are insulated from actions taken by classes defined in another class loader L_2, unless those actions affect state visible to both sets of classes (for example, state belonging to classes defined in a third class loader L_{parent} that is an ancestor of both L_1 and L_2), or classes in L_1 are actively exposed to the classes in L_2 (perhaps by passing references through commonly visible state).

Life span. In the Java run-time environment, classes, like objects, can be garbage collected when no longer referenced. An extra condition applies, though: classes can be garbage collected only when their defining class loaders are eligible for garbage collection as well [3]. Since each class strongly references its defining class loader, this implies that the set of classes defined by a class loader becomes eligible for garbage collection as a unit—an outstanding reference to any one of the classes will keep the entire set alive. This rule is present to ensure a consistent view of state between classes. Consider two classes C_1 and C_2 defined in the same class loader—if C_1 were permitted to be garbage collected while C_2 remained loaded, then a subsequent use of C_1 from C_2 (or elsewhere) would force C_1 to be reloaded and reinitialized, losing any state previously visible to C_2.

Security. The Java security model uses permission objects to control access to protected resources and capabilities. For example, reading a local file requires that the caller possess in its set of granted permissions an appropriate `java.io.FilePermission`. The set of permissions granted to a given class is determined in part by its defining class loader—specifically, the class loader chooses a *protection domain* for the class when defining it, which describes the security-relevant attributes of the class, such as the location from which it was loaded, and any certificates used to sign it. This protection domain is later mapped by the security policy of the enclosing Java process to a set of granted permissions. The defining class loader may also assign a base set of *static per-*

missions to each protection domain it creates; these permissions are granted to the protection domain regardless of the security policy in effect.

The combination of functions served by class loaders neatly fits code downloading scenarios, such as applets running in a web browser, where there is minimal interaction and dependence between code loaded from different remote sources. Simply by defining groupings of downloaded code in separate class loaders, an application can ensure that the different sets of code cannot interfere with one another or clash over class names, and can be unloaded independently. Class loaders are also useful for automatically granting permissions to downloaded code based on the code's origin. For example, through their defining class loaders, applets can be granted security permissions permitting them to connect back to the hosts from which they were loaded.

3 RMI class loading

Java RMI provides the ability for Java processes to invoke methods on *remote objects* hosted by other Java processes. A remote object is represented within a client process by a *proxy object*, which is responsible for translating local method invocations made on it by the client into call data sent to and received from the remote object's hosting process. A key differentiator between Java RMI and other remote procedure call systems is its support for passing full-fledged objects as the arguments and return values of remote calls, including objects for which class definitions are not available in advance in the receiving process. Java RMI achieves this through clever use of class loading: classes for received objects are resolved in class loaders that can fetch the class definitions from remote locations specified by the sender [8].

RMI class loading is also employed by Jini. Jini is a service-oriented architecture built on top of the RMI programming model that adds features such as ad-hoc discovery, remote events, transactions, and leasing [9]. Jini systems are composed of *services*—components that offer functionality to clients elsewhere on the network, as well as to each other. Like remote objects in Java RMI, Jini™ services are represented within client processes by proxy objects, and may send or receive Java objects in the course of performing remote operations—a Jini service that performs all operations remotely may in fact be implemented simply as a remote object. Unlike remote objects, Jini services can utilize proxy objects that perform client-side computation as well as (or instead of) remote communication, and are not constrained to using a Java RMI implementation to transmit data. Even when sending objects over other channels, though, Jini clients and services must still be able to remotely download class definitions for the objects, which they accomplish using RMI class loading, independently of the rest of the RMI infrastructure.

3.1 Object marshalling

Java RMI passes objects between Java processes using an augmented form of Java object serialization. Java object serialization enables objects to be *serialized* (flattened) to, and *deserialized* (reconstituted) from, a sequence of bytes; this byte sequence, called a *serialization stream*, can then be directed over the network or stored persistently [10]. Code for the objects is not directly included in the serialization stream. Rather, the serialized form of an object consists of its serializable state (by default, the values of its fields, except for those declared `transient` or `static`) along with *class descriptors*, which identify the object's class and its superclasses by name, but do not contain their class definitions.[7] To deserialize the object, the class descriptor representing its concrete class is mapped to an actual class in the deserializing process, that class is instantiated, and the resulting object instance is initialized based on its state read from the stream.

Basic object serialization by itself does not entirely fulfill the needs of Java RMI, since its serialization streams do not provide information about locations from where classes can be remotely loaded. Java RMI plugs this gap by extending the standard serialization format (as well as the utility classes that produce and consume serialization streams) to include alongside each class descriptor a *codebase annotation*: a list of URLs indicating sources, called *codebases*, from which a definition for the class, and other classes referenced by it, can be downloaded. To distinguish them from standard serialization streams, these extended serialization streams are commonly called *marshalling streams*, and the acts of serializing and deserializing objects with codebase annotations are referred to as *marshalling* and *unmarshalling*, respectively.

Note that when an object is unmarshalled, only the class descriptor and codebase annotation for its immediate, concrete class play a part in the resolution of that class; selection of the object's class then implies its superclasses. Thus, even though marshalling streams also include codebase annotations for superclasses of marshalled objects, these annotations are (for the most part) effectively ignored during unmarshalling.[8]

3.2 Class resolution

As part of unmarshalling, the RMI infrastructure must map class names encountered in the marshalling stream to classes to instantiate. This mapping is performed by the `java.rmi.server.RMIClassLoader` class. `RMIClassLoader` is not a class loader itself; rather, it is a class comprised of static methods that implement various parts of the RMI class loading mechanism, typically by internally invoking class loaders.

`RMIClassLoader` by default manages a set of class loaders, called *codebase*

[7] Strictly speaking, only class descriptors for serializable superclasses (i.e., those that implement the `java.io.Serializable` interface) are included in the serialization stream.

[8] The only exceptions to this are if a superclass itself serves as the concrete class of other objects in the stream, or if the stream contains a `Class` object representing the superclass.

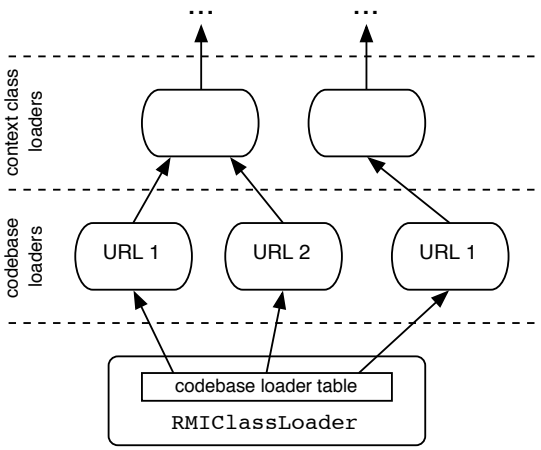

Figure 4: `RMIClassLoader` and codebase loaders

loaders, through which it loads classes (it can also be configured to employ other class loading schemes, as described in section 3.4). Each codebase loader is uniquely characterized by two values: a parent loader, to which it delegates, and a list of codebase URLs, from which it loads class definitions if delegation to the parent fails. Codebase loaders are stored in a table that indexes them based on these values. Figure 4 illustrates this structure.

The default class resolution algorithm used by `RMIClassLoader` is as follows: `RMIClassLoader` receives as input for each class loading request a class name and associated codebase annotation read from the marshalling stream, and, optionally, a class loader argument termed the *default loader*—by convention a class loader related to the calling context, such as the loader of the calling class itself.[9] To load the class, `RMIClassLoader` first tries the default loader, if specified. If the class is not found in the default loader, `RMIClassLoader` then consults its internal table to obtain the codebase loader that loads from the codebase annotation's URLs and has the calling thread's context class loader as its parent; if no such codebase loader exists yet, then one is created. Finally, `RMIClassLoader` calls on the codebase loader to load the class.

Effectively, `RMIClassLoader`'s default class loading algorithm prefers resolving class names to classes "belonging" to the caller, by first trying the caller-

[9]The exact value passed as the default loader varies depending on the Java RMI implementation used, and whether the class resolution occurs during remote method invocation or dispatch. For example, the standard RMI implementation provided by the Java 2 platform (also known as JRMP) uses the closest non-bootstrap class loader on the call stack, if one is present. Jini Extensible Remote Invocation, an alternate Java RMI implementation included in the Jini Technology Starter Kit, during invocation uses the defining class loader of the proxy object through which the remote invocation was made, and during dispatch uses the defining class loader of the exported remote object, unless explicitly directed otherwise.

provided default loader, rather than immediately using a codebase loader to load the given class. Furthermore, since each codebase loader delegates to its parent loader—a context class loader—before attempting to define classes itself, application classes visible in context class loaders are also favored over remotely loaded classes. Even when classes are loaded remotely, secondary classes referenced by them can resolve to local application classes, again due to the delegation of codebase loaders to context class loaders.

3.3 Codebase annotation

The RMI class loading mechanism is responsible for producing as well as consuming codebase annotations. To marshal an object, a sending process must emit a codebase annotation for the object's class, which requires identifying a set of URLs from which the class (and potentially other classes it references) can be downloaded. The task of mapping from a class to its codebase annotation is handled, along with class resolution, by the `RMIClassLoader` class.

In the RMI class loading model, the originating process of a (to be) downloaded class is responsible for establishing its codebase. From then on, the (logical) class ideally remains associated with that codebase, even as instances of the class are retransmitted from one process to another, allowing the same codebase to be reused by each receiver of the class. From the perspective of a single process, this translates roughly into the following rule: downloaded classes should be annotated with their remote sources, so as to "preserve" their codebases, whereas local classes should be assigned a codebase annotation associated with the process itself, since the process is serving as the originator of these classes.

To achieve this behavior, `RMIClassLoader` by default determines the codebase annotation for a class based on its defining class loader. If the defining loader is a codebase loader,[10] then the codebase annotation is the list of source URLs for that loader. Otherwise, the codebase annotation is taken from the `java.rmi.server.codebase` system property value of the sending process. Thus, codebase annotations are normally preserved by a causal chain: the incoming codebase annotation for a downloaded class determines the defining codebase loader of that class, which in turn serves as the source of the outgoing codebase annotation for the class. This sequence of events is depicted in figure 5.

Note that the model described above does not preserve codebase annotations for classes that are resolved locally, since the incoming codebase annotations for these classes are not tied to their defining loaders. This does not present a problem if all processes in a system load the same set of classes locally—even though the codebase annotation for a class in this set may change each time it is marshalled by a different process, no receiving process will ever depend on the value of the codebase annotation, since all processes can load the class locally.

[10] Or a `java.net.URLClassLoader`, not including the application class loader or its ancestor loaders.

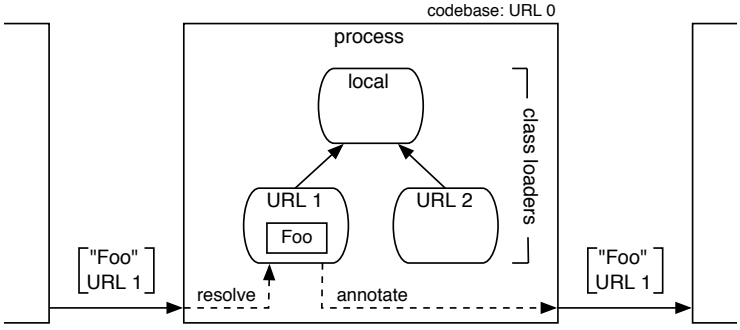

Figure 5: Class annotation

In systems composed of processes with non-uniform sets of locally available classes, however, codebase annotation loss can indeed cause errors; this issue is examined further in section 4.2.

3.4 Service provider interface

This section has focused on the default class loading scheme employed by the `RMIClassLoader` class and, by extension, the Java 2 Standard Edition (J2SE™) RMI implementation. In J2SE version 1.4, `RMIClassLoader` was made pluggable through the addition of a service provider interface, `java.rmi.server.RMIClassLoaderSpi`, enabling other class loading algorithms to be substituted in the form of alternate back-end implementations, called *providers*. One such provider is presented later, in section 5.

4 Issues

RMI class loading succeeds in enabling code mobility between Java processes. It is well suited to self-contained, one-to-one interactions between clients and servers. However, experience with RMI class loading, particularly within the scope of Jini, has shown that it does not deal as seamlessly with more complicated use cases, particularly those involving multi-party interactions. Type conflicts can result from mixing objects whose classes were loaded from different codebases. Relaying objects from one process to another can fail in certain cases due to codebase annotation loss. Changes to codebase content may fail to reach clients. Class loader delegation can lead to unwanted local class resolution. Lastly, codebase configuration and availability issues follow from the use of separate channels for transmitting code and data.

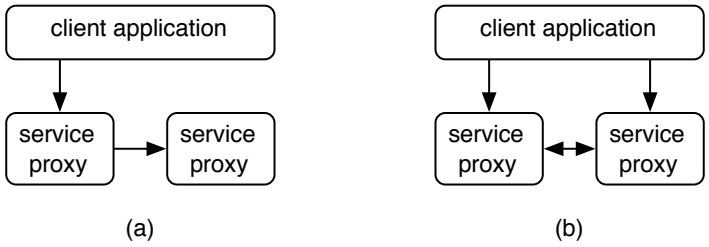

Figure 6: Type conflict scenarios: (a) service composition, (b) service orchestration

4.1 Type conflicts

As described in section 3.2, `RMIClassLoader`'s default behavior is to resolve classes using different codebase loaders, depending on the codebase annotation of each class, along with the context class loader present during unmarshalling. Recall that classes defined in different class loaders constitute distinct types. Therefore, classes loaded from different codebases (with the same context class loader in effect) are not directly compatible with one another—interaction can only occur through shared supertypes, defined in common ancestor loaders of the codebase loaders involved. Attempts to assign or cast directly between classes defined in sibling codebase loaders will fail, triggering `ClassCastException`s.

Some examples of more specific circumstances in which type conflicts can arise are listed below. For brevity, we use the notation $codebase(P)$ to indicate the codebase annotation associated with a process P (i.e., P's `java.rmi.server.codebase` system property value), and the notation $loader(A, L)$ to denote the codebase loader assigned to codebase annotation A under context class loader L; we further abbreviate this to $loader(A)$ in the common case where the context class loader is set to the application class loader (its default value). Unless otherwise noted, the context class loader for all threads involved in the examples is assumed to be the application class loader.

Service composition. A client process P_{client} uses service S_1, whose proxy object (instantiated within P_{client}) intends to internally use service S_2, invoking S_2 through an interface named I that S_2's proxy object implements. S_1 and S_2 are provided by separate server processes P_1 and P_2, such that $codebase(P_1)$ and $codebase(P_2)$ differ. Because the use of S_2 is a hidden implementation detail of S_1, none of S_2's classes, including I, are locally resolvable in P_{client}. Therefore, S_1's proxy object's use of I is bound to $\langle I, loader(codebase(P_1))\rangle$, whereas S_2's proxy object implements the interface $\langle I, loader(codebase(P_2))\rangle$. As a result, when S_1's proxy object attempts to use S_2's proxy object as a value of (S_1's notion of) type I, a `ClassCastException` is thrown.

Service orchestration. A client process P_{client} plugs together proxy objects for

services S_1 and S_2, provided by server processes P_1 and P_2 with different codebases. S_1 and S_2 try to interact using a class (or interface) named C that is not exposed in their public service interfaces, and hence is unknown to P_{client}. For example, C may be a specialized service interface that S_2's proxy object implements and S_1 understands, or C may be a concrete subclass of an object passed between S_1 and S_2 through an interface that expresses the value in terms of a publicly known (but more general) supertype. Then, by similar logic to the service composition case, S_1's proxy object binds C to $\langle C, loader(codebase(P_1))\rangle$, while S_2's proxy object views C as $\langle C, loader(codebase(P_2))\rangle$, ultimately resulting in a `ClassCastException`.

Codebase annotation changes. A client process P_{client} uses service S hosted by process P_{server} with $codebase(P_{server}) = A_1$. Classes belonging to S's proxy object are accordingly defined in $loader(A_1)$ within P_{client}. Later, P_{server} is reconfigured so that $codebase(P_{server}) = A_2$, while S's proxy object remains instantiated in P_{client}. Classes of objects that P_{client} subsequently receives from P_{server} may then, depending on the class and the context in which unmarshalling occurs,[11] be resolved in $loader(A_2)$, rendering them incompatible with the existing set of classes used by S's proxy. Such a codebase annotation change could be caused by a modification to the hostname or port of the codebase server, or, if using a content-sensitive URL scheme such as HTTPMD [11], could result simply from an update to the codebase content served.[12]

It should be noted that in certain situations, type conflicts can be averted through clever manipulation of the context class loader. For example, in the service composition case above, if S_1's proxy object were to set the context class loader to $loader(codebase(P_1))$ before unmarshalling S_2's proxy object, then $loader(codebase(P_2), loader(codebase(P_1)))$ would be used to initiate loading of classes for S_2's proxy object; through class loader delegation, S_2's proxy would then implement $\langle I, loader(codebase(P_1))\rangle$, the same interface seen by S_1's proxy. This technique, however, is not ideal: it is easy to bungle (the context class loader could be set to the wrong value, or not restored properly), requires additional permissions to get and set the context class loader value, and does not generalize to all cases of type conflicts.

4.2 Codebase annotation loss

While defining classes in different class loaders can lead to type conflicts, combining classes in the same class loader can result in a different problem—codebase annotation loss.

[11] Assuming default class resolution behavior, if the default loader passed to `RMIClassLoader` is $loader(A_1)$ (as is likely to be the case if S's proxy performs the unmarshalling) and the class in question is resolvable in this loader, then it will be resolved in $loader(A_1)$, retaining type compatibility with classes established under the previous codebase. Otherwise, it will be be resolved in $loader(A_2)$.

[12] Since JAR files embed a timestamp, even simple repackaging of a JAR file could cause codebase content to change.

Recall that in the standard RMI class loading scheme, classes are annotated based on their defining loaders, using the source URLs of the defining loader if the class was loaded remotely. Therefore, if the class of an object being unmarshalled is resolved to a class defined in a codebase loader, that class will "retain" its original codebase annotation when instances of it are marshalled elsewhere. If, however, the object's class happens to be resolved in a local class loader, such as the application class loader, then the class will be annotated with the value of the `java.rmi.server.codebase` system property in outgoing marshalling streams, effectively losing the original codebase annotation.

For example, suppose that process P_1 creates an instance O of a class named C, and that P_1 marshals O to another process P_2, which in turn marshals it to a third process P_3. As with the examples in the previous section, assume that the context class loaders of all threads involved are set to the application class loaders of their respective processes. Let C be resolvable in the application class loaders of both P_1 and P_2, but not in P_3. When P_1 marshals O to P_2, C is annotated with $codebase(P_1)$, since it was loaded locally in P_1. Within P_2, C is resolved locally despite its incoming annotation, since the codebase loader assigned to load it first delegates to the application class loader. As a result, when P_2 marshals O to P_3, C is annotated with $codebase(P_2)$ instead of $codebase(P_1)$, causing P_3 to attempt to load C in $loader(codebase(P_2))$ rather than $loader(codebase(P_1))$.

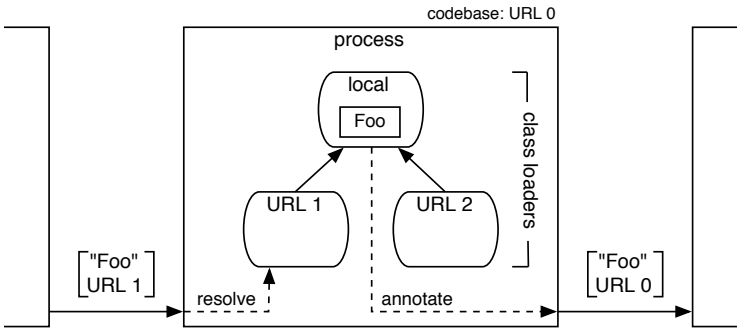

Figure 7: Codebase annotation loss

Codebase annotation loss is undesirable for a number of reasons. It often leads to class resolution failures: in the example above, even though P_2 resolves C locally, the sources indicated by $codebase(P_2)$ may not offer a class definition for C, in which case P_3 would be unable to resolve C. Even if the new codebase annotation happens to work, the codebase to which it refers may have a shorter lifetime than the original codebase and the object itself, setting the stage for future class resolution failures. Additionally, classes loaded from the old and new codebases would be type incompatible with one another: if P_1 were to also send O directly to P_3, then the directly and indirectly received copies of O

would be instances of $\langle C, loader(codebase(P_1))\rangle$ and $\langle C, loader(codebase(P_2))\rangle$, respectively. Finally, since permission grants to a class are often tied to its codebase, codebase annotation loss can cause a class to be afforded different rights than it would otherwise possess.

4.3 Codebase annotation mixing

Oftentimes, values sent between processes in Java RMI- and Jini-based systems are not single objects, but graphs of interconnected objects. For example, if the serializable state of an object O_1 includes references to objects O_2 and O_3, then marshalling O_1 has the effect of marshalling O_2 and O_3 along with it.

Ideally, when a graph of objects is marshalled, the classes of the objects within are all assigned the same outgoing codebase annotation. This way, a single codebase loader will be used to resolve all of the classes during unmarshalling. If a marshalled graph of objects contains a mix of codebase annotations, though, then the classes will be handled by different codebase loaders upon unmarshalling, frequently causing type conflicts when (for example) the field type of one object resolves to a different type than that of the object to which it refers. Any hierarchical relationship between the defining loaders of the classes in the sending process is lost in the receiving process, where the different codebase annotations are mapped to sibling codebase loaders.

A common situation in which codebase mixing arises is when a process marshals an object graph containing instances of both local and downloaded classes. Suppose, for instance, that process P_1 constructs an object graph containing an instance of $\langle C_1, L_{app}\rangle$ with a field of type $\langle C_2, L_{app}\rangle$ that refers to an instance of $\langle C_3, loader(A)\rangle$, where C_2 is a supertype of C_3, L_{app} is the application class loader, and A is a codebase annotation other than $codebase(P_1)$. This object graph can exist in P_1 without violating type rules, since the type of the field is a supertype of its referent. When P_1 marshals the object graph to another process, P_2, C_1 is annotated with $codebase(P_1)$, whereas C_3 is annotated with A.[13] Consequently, P_2 resolves C_1 and C_3 using different codebase loaders; assuming that neither C_1, C_2, nor C_3 are locally resolvable within P_2, then C_1's field type resolves to $\langle C_2, loader(codebase(P_1))\rangle$, while C_3's supertype resolves to $\langle C_2, loader(A)\rangle$. This mismatch triggers a `ClassCastException` during unmarshalling, when P_2 attempts to assign C_1's field its value.

Codebase annotation loss can also lead to codebase annotation mixing. If a marshalled graph of objects with uniform codebase annotations is unmarshalled and then remarshalled by a process such that some, but not all, of the annotations are lost, then the remarshalled object graph will carry mixed codebase annotations. This could occur if the process performing the remarshalling had locally resolved some of the classes in the object graph, but downloaded others.

[13] C_2's annotation is immaterial, because C_2 is not the concrete class of any object in the graph.

4.4 Codebase content changes

Section 4.1 described how modifying the codebase annotation associated with a running service can trigger type conflicts. A different problem can result if the content of a service's codebase is updated without changing the codebase annotation that refers to it—client processes may continue to use outdated versions of classes loaded prior to the codebase change, despite the publishing of newer versions.

Codebase updates may fail to reach clients due to either of two reasons. First, if a client strongly references a codebase loader populated with outdated codebase contents (for example, if the client application still holds references to instances of classes defined in that loader), then the codebase loader will remain resident in `RMIClassLoader`'s codebase loader table,[14] and will continue to be consulted when loading classes from that codebase. Since the codebase loader must preserve all mappings it accumulates from class names to loaded classes, it cannot replace previously loaded classes with newly loaded versions. Even if the stale codebase loader is no longer strongly referenced by the client, it may be "resurrected" if a request to load a class from its associated codebase arrives before the codebase loader is actually garbage collected.

A second factor that can mask codebase updates from clients is JAR file caching. By default, the Java platform's standard "`jar:`" protocol handler caches JAR file contents based on their referring URL for the lifetime of the Java process. Therefore, if a client process rereads a JAR file from a codebase whose contents have changed since the client read it initially, the client will continue to see the JAR file's original contents.

Retention of outdated codebase content can be avoided by changing a codebase's URL each time its contents are modified, since this ensures that the updated codebase content is viewed as a new entity by both the client's codebase loader table and its JAR file cache.[15] However, as mentioned previously, changing codebase URLs can lead to type conflicts in certain cases.

4.5 Undesired local class resolution

The standard class loader delegation model calls for class loaders to delegate to their parent loaders before defining a class themselves. The rationale behind this is that defining classes further up the delegation hierarchy encourages type sharing, minimizing the number of classes to load and reducing the possibility of type conflicts. Consider the simple case of three class loaders, L_1, L_2, and L_{parent}, each capable of defining a class named C, where L_1 and L_2 are child loaders of L_{parent}. Suppose L_1 and L_2 are each asked to resolve C. If they delegate first, a single, shared class results, whereas if they delegate only as a

[14] Assuming the default `RMIClassLoader` provider or `PreferredClassProvider` (described in section 5) is used.

[15] In this respect, one side benefit of using content-sensitive URL schemes such as HTTPMD to refer to codebases is that codebase URLs change automatically as a result of codebase content updates.

fallback, they will define separate classes in themselves that are type incompatible with one another.

A consequence of standard class loader delegation, when applied to RMI class loading, is that the class of a marshalled object is always resolved locally when possible. Sometimes, this behavior is appropriate. However, there are also reasons for wanting to force class downloading, described below:

Implementation control. A marshalled object may depend on a specific implementation of its class, provided by the stream-specified codebase, or may wish to eschew faulty versions of the class known to exist in receiving processes.

Package access. Just as classes defined in different class loaders are considered distinct, so are the run-time packages to which they belong. If some classes in a (lexical) package are loaded locally while others are downloaded—perhaps due to differing notions in the sender and receiver of the set of classes in the package—then classes on opposing sides of the resulting class loader divide will be denied access to each other's package-private members.[16] Forcing all classes in a package to be downloaded avoids this possibility.

Type separation. The publisher of a class may wish to ensure that the class is kept distinct from local classes in receiving processes, both to prevent local code from accessing its private methods and fields, and to guarantee separate class-level state.

Security permissions. Forcing a class to be downloaded ensures that it will be granted the permissions assigned to its remote codebase. In practice, this is rarely a problem, since local classes are likely to be granted more extensive permissions than downloaded code.

4.6 Codebase configuration

In the RMI class loading model, code and data are transmitted separately—the marshalled form of an object contains a codebase URL rather than actual class definitions. This allows the receiver to load class definitions only when necessary, but comes at a cost: sending an object depends on the prior configuration of codebases for classes referenced in the object's marshalled form.

With the standard RMI class loading mechanism, a deployer specifies the codebase for a process by setting its `java.rmi.server.codebase` system property. Note that setting this property does not trigger any action to actually host class definitions at the codebase—it simply asserts a list of URLs with which to annotate locally loaded classes in outbound marshalling streams. Configuration of the codebase is a separate set of steps, often performed outside of the main

[16] Java's package sealing mechanism attempts to address scenarios such as this by limiting definition of all classes in a sealed package to the same class loader. However, package sealing cannot handle the case where a child class loader defines classes in a package before the ancestor class loader has a chance to do so, as described in JDK bug 4302406 (`http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4302406`).

process. A typical sequence involves first creating a JAR file containing class definitions to make available to remote processes, then starting a "class server" (e.g., an HTTP server) to serve the JAR file, and lastly launching the application process, specifying in its command line a URL for the class server as the `java.rmi.server.codebase` value.

Establishing the codebase in this manner is an inconvenient and error-prone process. The deployer may mistype or otherwise specify an incorrect value for the `java.rmi.server.codebase` system property. The codebase itself may contain the wrong content—it might omit needed class definitions, or provide out-of-date versions of them. The deployer might even neglect to host a codebase in the first place.

Confusion as to the function of codebases—particularly that codebase URLs are resolved remotely—also contributes to codebase misconfiguration. Deployers sometimes specify file URLs as the value of `java.rmi.server.codebase`, causing class resolution failures in remote processes because the file paths are only meaningful locally. Another common error is listing "localhost" as the hostname in a codebase URL, which directs the receiver of an object to attempt to downloaded needed classes from its own host, as opposed to that of the codebase.

Since codebases are chosen during deployment, they cannot be verified by the compiler; thus, successful compilation of an RMI- or Jini-based application provides little assurance in itself that transfer of objects between processes will actually succeed. Because the run-time failures resulting from codebase misconfiguration may not occur immediately, or may only appear during certain interactions—for example, if an object of a specific class is sent—even successful initial execution of an application is no guarantee against later problems.

Security must also be considered. A process may wish to ensure the confidentiality and/or data integrity of objects it sends and receives. Securing the channel over which marshalled data is transmitted is insufficient by itself, since part of each object—its code—may be sent separately. Therefore, codebases referenced by a secured marshalling stream must themselves be secured as well, in a manner consistent with the protections applied to the marshalled data—this can be achieved, for example, by using HTTPS or HTTPMD URLs (or even file URLs, if all participating processes share a trusted filesystem). This further complicates configuration, since security must be arranged in two places rather than one, through non-uniform mechanisms.

A straightforward way to eliminate many of the sources of misconfiguration is to automate codebase deployment. For example, an RMI/Jini process launching utility might programmatically start an HTTP server, and construct from its host and port an appropriate `java.rmi.server.codebase` property value. Unfortunately, this approach isn't universally applicable. For maximum flexibility, the RMI class loading model does not limit the types of URLs that can be used for codebases, so long as they are understood by receivers: a codebase may be hosted on a different machine than its associated process(es), or may employ a non-standard URL scheme and protocol. These custom deployments cannot be handled by an automated mechanism. Furthermore, some aspects

of codebase deployment are less amenable to automation: determining proper JAR file contents requires advance knowledge of the types of objects to be marshalled at run time, and ensuring security is configured sensibly may depend on trust assumptions or other external factors that are not easily codified.

4.7 Codebase availability

Transmitting code and data separately can also limit the scope of marshalled objects. A marshalled object is only as available as the codebases on which it depends—if any of them are unreachable, then the object cannot be unmarshalled.

Long-lived objects pose a codebase availability problem. For example, an object may be marshalled to persistent storage, and not retrieved until years later. For retrieval to succeed, the codebase must still exist in the exact same location. Even if one is willing to commit the resources to maintain a codebase over time, it is difficult to determine where and to what degree such an effort is necessary: marshalled objects may have arbitrarily long lifetimes, and there is no built-in mechanism for tracking how many (if any) marshalled objects rely on a given codebase.

Another way in which a codebase may become less available than the objects depending on it is if its class server, or the network providing access to the class server, fails. Therefore, highly available marshalled objects require highly available codebases. Consider the case of a Jini service whose proxy object communicates remotely with a back-end server. A sensible strategy for reinforcing availability of the service is to eliminate the single point of failure represented by the back-end server, by deploying additional servers and equipping the proxy to switch from one to the other in case of failure. However, the codebase for the proxy object is also a single point of failure—if unavailable, the proxy object cannot be instantiated in the first place. Currently, the standard solution to this problem is to use a highly-available (clustered) web server for the codebase, which may be costly to administer and depend on custom DNS configuration.

A codebase may also be unavailable because it is unreachable. While objects may travel between processes, their codebases remain stationary. If an object travels outside the range of its codebase, then it can no longer be unmarshalled. This could occur if the codebase server resides within a private network protected by network address translation and/or a firewall, and objects dependent on the codebase migrate outside of the network. Because of this, deployers must decide up front what the domain of objects exported by a process will be, and select their codebase accordingly.

5 Preferred class loading

In response to several of the class loading issues described in section 4, version 2.0 of the Jini specifications introduced the notion of *preferred classes* [12]. A preferred class is a class that is loaded by a class loader without the loader first dele-

gating to its parent loader. Preferred class loading is supported by an alternative `RMIClassLoader` provider, `net.jini.loader.pref.PreferredClassProvider`, an implementation of which is included in versions 2.0 and higher of the Jini Technology Starter Kit.

`PreferredClassProvider` mimics the default `RMIClassLoader` provider in most respects: class loading is still handled by codebase loaders keyed on the codebase annotation of the class to load in conjunction with the context class loader of the invoking thread. `PreferredClassProvider` differs from the default provider in that the codebase loaders it creates delegate class loading requests to their parent loaders only when the class or resource to load is not preferred; also, the caller-associated default loader is consulted only for non-preferred classes.[17] Preferred classes for a given codebase URL list are designated by a *preferred list*—a resource (i.e., file or JAR file entry) located at a well-known position relative to the first codebase URL in the list.[18] The preferred list itself consists of a sequence of name expressions, each accompanied by a true or false value indicating whether classes or resources matching the name expression are to be preferred. If the preferred list is absent, then no classes or resources are preferred.

The following example illustrates the effects of preferred class loading. Consider a process P that unmarshals instances of classes named C_{pref} and C_{reg}, both with codebase annotation A (assume the context class loader is set to the application class loader throughout). C_{pref} is preferred and C_{reg} is not, as specified by the preferred list contained in their codebase. Then, because C_{reg} is not preferred, loading of it follows the default class resolution algorithm: through class loader delegation, it will be defined in the senior-most ancestor loader to which a definition is available, and thus defined in $loader(A)$ only if not resolvable locally. C_{pref}, on the other hand, will always be defined in $loader(A)$—its initiating loader—regardless of whether or not any ancestor loaders can load, or have already loaded, a class of that name.

5.1 Benefits

By offering codebases (and their deployers) more input into the class resolution process, the preferred class mechanism addresses the following class loading issues:

Codebase annotation loss. Recall that codebase annotation loss occurs when the class of an unmarshalled object is resolved locally, in a class loader whose associated codebase annotation differs from the incoming codebase annotation of the class. Marking a class as preferred prohibits such resolution. Thus, a preferred class cannot "lose" its codebase, since it will always be defined in a codebase

[17] While this statement summarizes their high-level intent, the exact rules governing use of the default loader are somewhat more complicated; a full specification is given in [12].

[18] If the URL refers to a JAR file, then the preferred list is the `META-INF/PREFERRED.LIST` entry within that JAR file; if the URL is a directory, then the preferred list is the file at the path `META-INF/PREFERRED.LIST` relative to the directory.

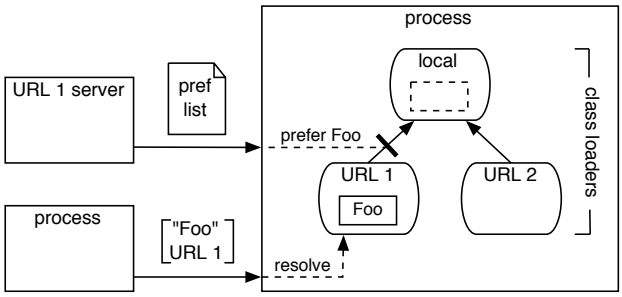

Figure 8: Preferred class loading

loader that preserves record of the codebase. Applied to the codebase annotation loss example from section 4.2, the class named C, now preferred, would be defined in $loader(codebase(P_1))$ within P_2, allowing C to retain its original codebase annotation of $codebase(P_1)$ in the marshalled data sent from P_2 to P_3.

Codebase annotation mixing. By preventing codebase annotation loss, the preferred class mechanism also avoids cases of codebase annotation mixing that result from it. Designating a group of classes from a given codebase as preferred ensures that a receiving process will define all of the classes in the same codebase loader, leading the classes to be assigned a uniform outgoing codebase annotation if remarshalled. However, if a process marshals an object graph it has explicitly assembled to contain instances of both local and downloaded classes, then the resulting marshalling stream will contain mixed codebases regardless of whether preferred class loading is in effect.

Undesired local class resolution. Preferred class loading directly addresses this issue by providing a way to disable local resolution of selected classes. Preferring a class ensures that the implementation of the class provided by its codebase will be used, and that the class will constitute a distinct type with separate state from any local class of the same name. Marking all classes in a package as preferred guards against package access errors by guaranteeing that the classes will be defined in the same class loader, and hence belong to the same run-time package.

5.2 Class boomerangs

For ease of explanation, the description thus far of preferred classes omits a detail of the class resolution algorithm necessary to handle cases in which a preferred class is loaded back into its (loosely speaking) originating process—a situation sometimes called a *class boomerang*.

Suppose a process P marshals an instance O of a local class $\langle C, L_{app} \rangle$ that

is marked as preferred in P's codebase, and that O (or another instance of the same class) eventually returns to and is unmarshalled by P—this could occur, for example, if a remote process were to echo O back to P, if P were to pass O in a remote call to a remote object hosted by P, or simply if P were to marshal and unmarshal O to/from a file. In O's marshalled form, C will be annotated with $codebase(P)$. Assume that the context class loader of the unmarshalling thread is L_{app}, the application class loader. If P were to follow the preferred class algorithm as currently outlined, it would resolve C in $loader(codebase(P))$, which would immediately define C without delegating to its parent loader L_{app}, since C is preferred. The resulting class $\langle C, loader(codebase(P))\rangle$ would be separate from (and type incompatible with) its original self, $\langle C, L_{app}\rangle$.

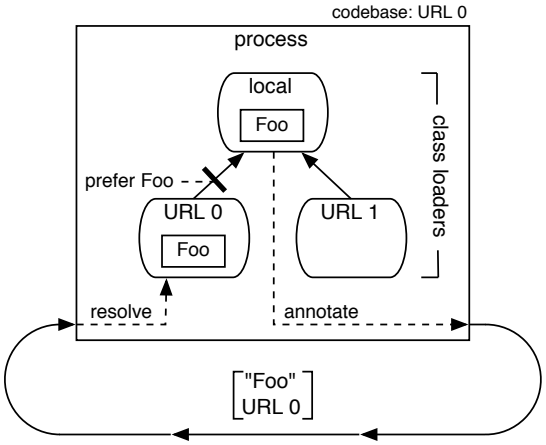

Figure 9: Naive (hypothetical) resolution of a preferred class boomerang

To deal this possibility, `PreferredClassProvider` institutes a special clause: if the codebase annotation of a preferred class to be resolved matches the outgoing codebase annotation associated with a class loader "from" the unmarshalling context, then that class loader is directly called upon to load the class. More precisely, if the class's codebase annotation matches the annotation associated with the default loader, then the default loader is first tried; if the annotation matches an annotation associated with the context class loader or one of its ancestor loaders, then the junior-most matching loader serves as the codebase loader. Thus, in the example above, L_{app} would be chosen as the codebase loader (if not also the default loader) to resolve C, since C's codebase annotation $codebase(P)$ matches the outgoing codebase annotation assigned to L_{app}. Consequently, C would resolve back to $\langle C, L_{app}\rangle$, preserving type consistency.

5.3 Preferred class selection

Effective use of preferred class loading requires careful selection of the set of classes to prefer. Preferring a class sacrifices type compatibility with local code, since it causes the class to be defined in a non-local class loader (ignoring class boomerangs). Therefore, classes appearing in public APIs should not be preferred, since doing so would render the API unusable for interaction between local and downloaded code. Type compatibility is not needed, however, for non-API "implementation" classes, which are not directly referenced by external code. Because of this, it makes sense to prefer these classes.

The decision to prefer a class also impacts resource consumption: if a class is preferred, then each process that resolves it (ignoring class boomerangs) must download and process its class definition, and allocate space for it, even if a class that is suitable in all aspects other than type identity has already been loaded locally.

The tradeoffs involved in preferring a class have subtle implications for API design. Designating public API classes as not preferred leaves them susceptible to codebase annotation loss. However, as noted in section 3.1, only codebase annotations for the direct classes of marshalled objects play a significant role during unmarshalling. Therefore, codebase annotation loss for interfaces and abstract classes is mostly harmless,[19] since they cannot be directly instantiated. For this reason, it is recommended that public APIs be defined purely in terms of abstract types (with all other classes marked preferred), as doing so limits the scope of codebase annotation loss to classes for which such loss has little impact.

5.4 Limitations

Despite its benefits, the preferred class mechanism leaves unsolved significant class loading issues affecting Java RMI and Jini:

Type conflicts. `PreferredClassProvider` adopts the same basic class loader structure as the default `RMIClassLoader` provider, creating separate class loaders in which to load classes from different codebases. Therefore, it is similarly prone to type conflicts between classes downloaded from different locations. It fares better than the default provider in one respect: by preventing codebase loss, preferred class loading also eliminates type conflicts artificially caused by codebase loss.

Unfortunately, `PreferredClassProvider` is actually more likely than the default `RMIClassLoader` provider to incur type conflicts in situations involving codebase changes. To see this, consider again the codebase change scenario presented in section 4.1: a client process P_{client} uses service S hosted by process P_{server}, whose initial codebase A_1 later changes to A_2 while P_{client} continues to use S's original proxy object. Suppose instances of an internal implementation class named C are passed back and forth between S's proxy and P_{server};

[19] Unless `Class` objects for such classes are sent between processes, which is rare in practice.

because C is not part of S's public API, it is marked as preferred. When S's proxy object receives an instance of C sent after the codebase change, C will be annotated with A_2. Under the default `RMIClassLoader` provider, C could still be resolved in the default loader (presumably set to $loader(A_1)$, since S's proxy object is the caller), avoiding a type conflict. With `PreferredClassProvider`, however, the default loader will not be consulted, since C is preferred, and does not qualify as a "boomeranged" class (i.e., C's codebase annotation does not match that associated with the default loader). As a result, C will resolve to $\langle C, loader(A_2)\rangle$, which clashes with the definition of C used by S's proxy, $\langle C, loader(A_1)\rangle$.

Codebase configuration and availability. Preferred class loading does not affect the interpretation or function of codebases themselves; it only changes the rules governing when they are used. Therefore, codebase configuration and availability remain factors to be considered. Codebase configuration actually becomes more complicated when preferred classes are used, since codebase deployers must perform the extra task of composing preferred lists for the classes to serve. As with the codebase contents themselves, these preferred lists cannot be statically verified for correctness. Preferred class loading is also slightly more sensitive to codebase failure, in that it requires the first element of the codebase URL list (from which the preferred list is loaded) to always be available, even if no classes are preferred.

An additional limitation of preferred class loading is that it depends on favorable API design and correct formulation of preferred lists in order to be effective. If an API needs to reference concrete classes, then those classes will not benefit from preferred class loading, since (to support type compatibility) they cannot be preferred. If a preferred list is improperly constructed, preferred class loading may end up increasing, rather than decreasing, the number of class loading-related failures in a system.

6 Discussion

The Java™ RMI and Jini architectures are compelling in large part because they enable programming of distributed systems using the standard Java object model. They do not rely on a separate interface definition language, with its own abstractions and limitations, to codify remote communication. Rather, distributed interactions are governed by the same Java language mechanisms and safeguards that apply to local operations.[20] Like objects, remote services are encapsulated: to interact with a service, a client need only know its interface, not how it is implemented, or what protocol it uses to communicate. The result is a programming model that in theory allows services to be reasoned about and used simply as Java objects.

[20] A clear distinction between local and remote interaction is maintained, however, by reflecting remote invocation failures as checked exceptions that must be handled or propagated by callers.

6.1 Impact

Class loading issues compromise the conceptual simplicity of the Java RMI and Jini architectures.

To the application developer, RMI class loading introduces restrictions and failure modes for downloaded code that are not expressed clearly through APIs or language constructs. A process cannot freely mix objects whose code was loaded from different locations, since they may not agree on common run-time types, resulting in unchecked `ClassCastException`s that application code is unlikely to be equipped to handle. Processes cannot transmit arbitrary (serializable) objects—codebases must be arranged in advance to host their class definitions, otherwise unmarshalling will fail. API design is affected as well: defining public APIs that reference concrete classes should be avoided, since it invites codebase loss, increasing the chances of unmarshalling failures and/or type conflicts.

Failures related to RMI class loading can occur even in applications without any obvious faults. Mixing of classes from different codebases may occur unbeknownst to an application, either because the application does not track the sources of objects, does not have visibility into third party code performing the mixing (which may itself have been loaded remotely), or because the mixing occurs inadvertently as the result of codebase loss in a separate process. Unmarshalling errors may also result from codebase loss or misconfiguration in another process.

Debugging, fixing, and avoiding problems caused by RMI class loading requires low-level knowledge of how RMI class loading works. To understand the causes of type conflicts and codebase loss, one must comprehend at a minimum the distinction between compile-time types and run-time types, the role of class loaders in defining type identity, class loader delegation, the function of the context class loader, the concept of codebases, and the relation of codebases to the codebase loaders created by the `RMIClassLoader` provider in effect. Addressing RMI class loading failures involves correspondingly arcane techniques, such as setting the context class loader to a different value, removing classes from a process's class path, or marking particular classes as preferred. To all but those already well-versed in RMI class loading, these methods do not correspond to any high level, intuitive notions of solutions. Encapsulation is diminished, since using a service requires knowledge of implementation—not of the service itself, but of the underlying mechanisms responsible for loading it.

Conceptually, RMI class loading presents a model of types that is difficult to reason about at a system-wide level. There is no global, fixed type hierarchy. Instead, relationships between classes may differ from process to process, depending on class loader formations assembled individually by each process at run time, whose shapes are influenced by variable factors such as the set of classes locally loadable by each process, context class loader settings, and the paths traveled by objects prior to receipt.

The dependence of RMI class loading on mechanisms and configuration external to the core application reduces ease of use, while increasing the likelihood

of class loading and codebase problems. JAR files must be packaged with correct contents and preferred class settings, and applications must be configured to export the proper codebase. Because these actions are performed during deployment, they cannot be verified through compile-time analysis. Furthermore, since semantic information about codebases—such as which classes should be made downloadable to other processes, or which types should be shared with code loaded from other sources—is not encoded in the application, there is no guaranteeably accurate metadata on which to base run-time checking. This places an extra burden on deployers: to fix configuration errors, they too must understand aspects of the RMI class loading model.

Pitfalls aside, even a properly developed and deployed Jini or Java RMI application may be artificially limited by the RMI class loading model. Codebases are a straightforward example: marshalled objects cannot extend beyond the reach of their codebase, across time (if the codebase ceases to exist), space (if communication to the codebase is not possible), and failures (if the codebase crashes). One might attempt to work around this by reannotating classes with new codebases, at the expense of probable type conflicts later on if the newly annotated code happens to interact with its original copy.

A less obvious limitation is that downloaded code can only interact along avenues of compatibility established, and therefore anticipated, at load-time. If two classes downloaded from different codebases wish to share types that are not resolvable locally in the host process, then the codebase loader of one must delegate to the codebase loader of the other. This decision must be made when the classes are defined—if proper delegation is not established at load-time, perhaps because the full set of run-time interactions between downloaded classes cannot be predicted, then the classes will not be able to interact through common types.

6.2 Impediments to solutions

Architecturally, RMI class loading issues are less straightforward to solve than they may appear.

Attempts to fix RMI class loading's type compatibility problems are hamstrung by the multi-function nature of class loaders. Intuitively, one might imagine eliminating the possibility of type conflicts by defining all downloaded classes in a single, unified class loader.[21] The immediate benefit of this approach is that it would flatten the type namespace for downloaded code, making it resemble more closely the compile-time type system. However, defining all downloaded classes in the same class loader would mean that none could be garbage collected until all were unreferenced, since class loaders determine class lifespan. Because class loaders also serve as record of the locations from which classes were loaded, using a single class loader would complicate tracking of codebases.[22] Finally, imposing a flat namespace on downloaded code would

[21] Informally, this approach has come to be called the "big happy class loader" model.

[22] One might instead refer to the `java.security.CodeSource` associated with a class to determine its codebase annotation, though this would only allow a single codebase per class

mean that only a single class definition, copy of class state, and set of security permissions could be bound to a given class name, which could be problematic in situations where multiple remote parties (perhaps trusted to differing degrees) each wish to provide their own copy of a class, and to be insulated from the actions of competing versions.

Codebase issues also defy easy solutions [13]. It is tempting to suggest that marshalling streams should include class definitions inline, rather than referencing them indirectly through codebase annotations. However, bundling class definitions for annotated classes alone would be insufficient—the marshalling stream would also need to contain definitions for any classes transitively referenced by them that receiving processes could not locally resolve. Receiving processes might also require resource files and metadata that are associated with classes in the stream, but not part of their class definitions. Including the full set of necessary class definitions, resource files, and metadata in each marshalling stream would be prohibitively wasteful in the common cases where instances of a given class or set of classes are repeatedly sent from one process to another, or written to storage in marshalled form.

There are other complications as well: with class definitions passed inline, there would be no codebase information available on which to base permission grants (though one might argue that the transience of codebase identity makes it a non-ideal foundation for security in the first place). For an object passed through a third party, the receiver would have to trust that the intermediary did not tamper with the code. Obtaining the class definitions to include in the marshalling stream would also present a technical snag: class loaders are not required to make accessible the raw class definitions of the classes they load (even though most currently do).

An alternate scheme would be to retain on-demand loading of class definitions, but always load from the immediate sender of the object, effectively rehosting the codebase with each transmission hop. This would require revisiting the way in which RMI employs class loaders, however: given the current class loader structure, in which codebase loader instances are tied to codebases, altering the codebase of a class would cause it to be defined in a different codebase loader, again leading to type incompatibility with other versions of itself.

6.3 Future directions

There are no readily apparent improvements within the bounds of the current RMI class loading model that address its remaining issues. This suggests that comprehensive solutions must be found elsewhere, by reconsidering RMI class loading at a more fundamental level.

At least two approaches are possible. One is to expose type compatibility, code downloading, and other remote class loading considerations in the Java RMI and Jini programming models.[23] For example, to marshal an object, an

name to be remembered.

[23]This approach was suggested by Peter Jones.

application may be asked to specify, through library calls or source code annotations, values controlling or describing how the object's code is sent. Similarly, unmarshalling an object might require application input on how the types and classes of the object should fit into the receiving environment. Code location information (or, in some situations, the code itself) could be represented explicitly as data members of an envelope object wrapped around each marshalled object. One could argue that extra mechanisms such as these deserve a place in the programming model because remote class loading, much like remote communication in general, cannot be made fully transparent to applications [14]. An advantage of this approach is that it could be implemented on top of the Java platform as it exists today, at the expense of extra (perhaps necessary) complication for developers.

Another approach is to revise the class loading mechanism itself. RMI class loading effectively establishes a distributed type system, whose characteristics are dictated to a significant degree by particulars of class loading in its current form. One might instead work in the opposite direction, starting with requirements and desired properties of a distributed type system, and deriving from those the underlying mechanisms needed to achieve them. A likely first step in formulating the type system would be to pry apart the multiple functions of class loaders, considering type compatibility, class identity, class lifetime, subprocess isolation, security permissions, and code location as distinct issues that may warrant separate mechanisms.[24]

Such a type system might also provide an opportunity to tackle other more general mobile code issues. Versioning is one: in distributed systems that are not centrally deployed, or that stay running over long periods of time, different versions of types and code inevitably emerge. A type system geared towards distributed interactions would ideally provide a vocabulary for identifying different versions of entities, and describing the relationships between them.

Platform issues are another consideration. Mobile code may depend on aspects of the environment into which it is loaded, such as certain local APIs or resources. Most Java RMI- and Jini-based systems currently handle these issues through out-of-band agreement on a minimum set of classes and facilities that downloaded code can assume to be present in each receiving process. A distributed type system might offer a way to describe the external dependencies of a class or module, allowing mobile code and objects to be safely matched with processes capable of supporting them.

Acknowledgments

Thanks to Tim Blackman, Jane Loizeaux, Victor Luchangco, Bob Scheifler, Jim Waldo, and Ann Wollrath for helpful feedback on earlier drafts of this paper.

[24] For example, subprocess isolation could be accomplished through the *Isolate* mechanism proposed by Java™ Specification Request (JSR) 121 [15, 16].

References

[1] Sheng Liang and Gilad 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, 1998.

[2] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*, chapter 5. Addison-Wesley, 1999.

[3] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*, chapter 12. Addison-Wesley, 2000.

[4] Vijay Saraswat. Java is not type-safe. `http://matrix.research.att.com/vj/bug.html`, August 1997.

[5] Li Gong, Gary Ellison, and Mary Dageforde. *Inside Java 2 Platform Security, Second Edition*, chapter 4. Addison-Wesley, 2003.

[6] Sun Microsystems. Java 2 Standard Edition JAR File Specification. `http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html`, 1999.

[7] Sun Microsystems. Java 2 Standard Edition Extension Mechanism Architecture. `http://java.sun.com/j2se/1.4.2/docs/guide/extensions/spec.html`, 1999.

[8] Sun Microsystems. Java Remote Method Invocation Specification. `http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC.html`, 2002.

[9] Jim Waldo et al. *The Jini Specifications, Second Edition*. Addison-Wesley, 2000.

[10] Sun Microsystems. Java Object Serialization Specification. `http://java.sun.com/j2se/1.4/docs/guide/serialization/spec/serialTOC.html`, 2002.

[11] Sun Microsystems. API Documentation for the `net.jini.url.httpmd` Package. `http://java.sun.com/products/jini/2.0/doc/api/net/jini/url/httpmd/package-summary.html`, 2003.

[12] Sun Microsystems. API Documentation for the `net.jini.loader.pref` Package. `http://java.sun.com/products/jini/2.0/doc/api/net/jini/loader/pref/package-summary.html`, 2003.

[13] Robert Scheifler. "Re: Code download concerns". Posting to the `rmi-users@java.sun.com` mailing list, June 2001.

[14] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A Note on Distributed Computing. Technical Report TR-94-29, Sun Microsystems, November 1994.

[15] Sun Microsystems. Java Specification Request 121: Application Isolation API Specification. `http://http://www.jcp.org/en/jsr/detail?id=121`, 2005.

[16] Grzegorz Czajkowski and Laurent Daynès. Multitasking without Compromise: a Virtual Machine Evolution. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 125–138, 2001.

About the author

Michael Warres is a researcher at Sun Microsystems Laboratories, focusing on large-scale distributed systems for managing sensor data. Previously, he was a member of the Jini network technology development team, where he worked on discovery protocols, security infrastructure, Java RMI, and object serialization. He holds Sc.B. and Sc.M. degrees in Computer Science from Brown University.