Dynamic Class Loading in the Lutris EAS Services Architecture

Lutris Technologies, Inc.

August 2001

A White Paper From Lutris Technologies, Inc.

1200 Pacific Avenue Suite 300, Santa Cruz, CA USA 95060

http://www.lutris.com (831) 471.9753

Table of Contents

Introduction

The class loading scheme in the EAS Services Architecture makes it easy to upgrade and compose applications and systems. This may sound ordinary, but it is unique and stems from the ability of EAS 4 to dynamically compose class loader delegation. This article explains how EAS Services Architecture class loading scheme works, in fairly dense technical detail. (This is a "hard hat area", in the words of one reviewer.) To set your eyes on the prize, I start by describing the service programming model for both client and service programmers. Then I summarize the requirements and implementation of Java class loading, and go on to show, step by step, how the EAS class loading scheme works, and how it differs from ordinary class loading schemes. This part takes some patience as I unwrap different layers of the problem and our solutions. Finally, I describe how to write services to maximize your upgrade possibilities, and suggest that such services are the most composable. If you get through all that here, in a future article we'll talk about how the class loading scheme works in the deployment of WAR and EAR applications.

You don't need to know this to use and enjoy EAS 4. But I think it is interesting, and it distinguishes EAS 4 from other J2EE application servers. If you are a developer, I hope to show you that services are easy to use and write. If you are a designer, I hope to suggest that the services programming model is superior in many cases to either EJB or XML architectures. If you are evaluating EAS 4 as an enterprise platform, I hope to present one of the technologies underlying the composability of EAS services, which will make EAS 4 more extensible, secure, usable, and reliable than the competition. While EAS 4 supports the J2EE- or XML-based programming models, you are also free to innovate with an architecture that brings the benefits of interface-disciplined programming up from the level of the object to the level of semantic services.

A Web of Services - Web Services and beyond

Much excitement now surrounds around the ideas of "Web Services" and "Service-based architectures." In these systems, components interoperate via XML data adapters and use XML to present meta-data about their facilities so they can be assembled into applications. While EAS 4 supports these architectures with SOAP services and XMLC, the underlying EAS Services Architecture itself has the same goals but uses a different approach: it delivers the same level of composability, but uses tightly-bound Java invocation interfaces for security and efficiency. The EAS class loading scheme makes this possible and makes EAS 4 unique among J2EE application servers. The scheme raises Java encapsulation from the level of an object to that of a service, providing a much richer and more powerful integration than XML or any fixed component model.

But the first benefit most people notice about our class loading scheme is not composability but upgradability. On most Java application servers, to upgrade the server you have to bring it down completely, and to upgrade an application component you have to stop the entire application, undeploy it, and redeploy entirely. This is because Java class loading requirements make it difficult to partition class binding scope; most vendors have used simple class loader hierarchies which require all associated components to be deployed or undeployed at once. As a

result, most lapses in server or application availability stem not from failures but from upgrades. On EAS 4, you can upgrade the implementation of services at run time even while maintaining existing clients.

This works through two mechanisms. First, when clients look up services using JNDI, the necessary class binding scope is automatically made visible to the client. Clients need not have library jar files in their deployment archives. Second, EAS 4 makes it possible to separate service implementation classes from the export/interface classes. Clients can continue to hold objects of export/interface classes while implementation classes are replaced, which permits old and new service objects to interoperate but prohibits clients from interfering with implementation classes. Together these mechanisms permit clients to always get exactly the classes they need (and no more!).

The services programmer's model

The programmer's model for service clients is natural: If you want a service, just look it up at run-time by name in JNDI. If it's a service that returns a proxy target, you can use the target like any other client object. If it's a library service than returns no target, you need not do a run-time lookup; you can add library lookup's to your deployment descriptors and never touch the code. Either way, after you look it up, you will be able to see all the service export/interface classes - classes determined to be correct at build time, because you compiled client code against the export classes of the service, ensuring that the class scope provisioning will be sufficient.

The programmer's model for service writers is natural, too. In the enhydra-services.xml descriptor, you segregate your implementation classes from your interface/export classes. If you are publishing a library, that's all there is - no coding required. If you are providing target service proxies, you write a factory for the targets. EAS 4 takes care of the rest.

You need not understand how the JNDI lookup of a service has the important side-effect of making visible the export classes required for the service target to be used, or that our class loaders are arranged in a federated scheme rather than a hierarchy, permitting local sharing without interfering with the global space. But understanding it is half the fun, and gives you another dimension for comparing application servers. The story starts back in the days when the language wasn't even called "Java" yet and enterprise-class computing was not foremost in the minds of the designers...

How we got here: Java class loading requirements

Requirements for Java

To realize Sun's vision of vast forests of embedded devices connected to a network, a primary design objective of the Java VM was the safe network download and execution of platform-independent bytecode. This leads to the VM's requirements for loading classes and to the implementation, which are sometimes summarized as one requirement, for run-time typing:

Feature	Requirement	Implementation
extensible	To permit bytecode to be loaded from anywhere, the loading system must be extensible.	ClassLoader superclass delegates to user subclasses to load bytecode.
lazy resolution	To respect network latencies, the VM has to defer binding until actually necessary.	VM defers loading & binding until dereferencing variables for fields and method parameters and result values.
multiple namespaces	To permit bytecode to be loaded at any time, the VM has to support multiple namespaces to avoid irremediable name collisions as new bytecode is loaded	each class loader defines a name space
irrevocable binding	To permit the VM to optimize within and across classes, binding once made must be irrevocable	VM enforcement of binding invariants
run-time typing	To be safe, the type system has to be enforced at run time, so the VM must be able at run time to resolve symbolic references within code, binding them to available binary code	All of the above and more...

The key things to remember from this are (a) lazy resolution means you may not know until well into run time that you have a class-binding problem; (b) the ClassLoader is a template class which establishes a standard algorithm and set of behaviors for loading classes that must be followed to get correct behavior. For these reasons, you must take care when writing your own class loaders, as we did.

Implementing class loading requirements for Java

Extensible: The class loading process is normally initiated by the VM which delegates the loading to a ClassLoader. The loading process involves loading bytecode for the classes, converting to any internal binary form, and resolving, which is primarily linking symbolic names to actual binaries (also called "binding"). The ClassLoader class defines the template algorithm in three methods, some of which subclasses override in order to modify the loading behavior. (I'll show with the full signatures but indented to show the order of invocation):

```
1.  public Class loadClass(String name..) //   top-level method
2.          public final Class findLoadedClass(String name) // find cached Class
3.          public Class findClass(String name..) // look for bytecode
4.                  protected final Class defineClass(String name, byte[]...) // convert to
class
5.                  protected final Class resolveClass(Class c) // link class
```

Basically, a class loader is supposed to check its cache first, and then try to find the class in its resources (files or URLs). If found, load in the bytecode and delegate to the superclass to define and resolve the class. Resolving the class may involve recursively loading more classes. Subclasses may implement any of the non-private, public or protected methods. Implementors mainly override `findClass(String name)` in order to access custom resources, but I'll discuss below the significance of overriding other methods.

Lazy resolution: When does the VM initiate loading? Basically, loading a class causes its parent to be loaded, but otherwise no class is loaded unless by dereference of a non-null reference. Classes may also be loaded explicitly. Thus, a class is loaded (a) explicitly by calling `ClassLoader.loadClass(..)` or `Class.forName(String...)`; or (b) implicitly by the VM, (i) when dereferencing a variable or (ii) when resolving the parent of a class. To load a class, the VM calls `loadClass(..)` on the ClassLoader of the currently executing class. Many programmers are confused by lazy resolution because you can have incompatible classes in a system and not get ClassCastException until much later, when a non-null reference is passed to another class.

Multiple namespaces: Each class loader defines a unique name space. Many Java programmers don't realize that the identity of a class is determined by the combination of the fully-qualified name of the class and its class loader. You can get ClassCastException when the two types have the same name; to understand how this is enforced, you have to understand that the VM maintains a set of binding invariants for each class.

Irrevocable binding and binding invariants: Given multiple namespaces, there can be more than one instance of Class representing a given type in a VM. There are many reasons to require that, once classes bind with each other, the binding be irrevocable, e.g., to permit the VM to make certain cross-class binary optimizations.[1] As a practical matter, irrevocable means (a) a class loader always returns the same Class when asked to load a given class name; and (b) no class may link with two other Class instances having the same name.

The ClassLoader cache mechanism implements both (not surprisingly, in final methods). First, all class loaders are supposed to check the cache before attempting to load a class. Second, the system maintains a set of rules (the binding invariants) which prevent any second class instance from being used when a first class instance has already been linked with (the third) class instance. For example, if class CA has been linked with the first class CB, the system adds an invariant rule for any field of type B in class CA that references thereto must be resolved to class CB. The system adds rules for all fields, method parameters, or method results of type B. These rules are added when a class is added to the cache or linked to another class. Whenever you get a ClassCastException from passing in an object of the correct type (but loaded from a different class loader than the same type class linked earlier), it is because one of these invariants was violated.

Most programmers experience the Java class loading only as failures - ClassNotFoundException, ClassCastException, etc. You now recognize one of the interesting cases of ClassCastException, and what it takes for there to be a ClassNotFoundException: the class loader was unable to find the class in the cache or in its resources.

But consider yourself a designer of an application server. Not only are classes distinct if they are in different name spaces, but also Java's memory management takes most of the control away from you. No class or class loader will be garbage-collected at least until all instances of those classes are not reachable. How would you implement undeployment and redeployment? Most have followed Sun's lead in using class loader hierarchies. This enables them to share classes, but it effectively makes the entire hierarchy into a single deployment unit. To understand how

[1] In J2SE 1.4, the debugging architecture (JPDA) permits you to replace a class at runtime. This is consonant with the irrevocable binding rule because you cannot replace it with another class in the running system, previously bound to third classes.

Lutris has worked around that problem, see how our class loader scheme achieves the same goal of sharing classes without pulling everything into a single deployment hierarchy.

Bridging space: using delegation to share common class loaders

Hierarchies and federations: sharing classes to bridge disparate name spaces

The goal of supporting multiple name spaces is not only to keep them distinct, but also to permit them to work together. The only way (short of data-based communications) is for two classes from different name spaces to be able to share classes loaded from a common class loader.

In the VM Sun established a delegation model that enforces a hierarchical scheme, whereby child class loaders can share common classes in the parent class loader. This is how (parent) system classes are shared with multiple (child) class loaders. In this scheme, a parent is set on ClassLoader construction, to ensure constancy of the delegate ordering, which is required to ensure that successive `loadClass(..)` calls always return the same class.

In the EAS Services Architecture the team at Lutris has added a delegation model that enforces a federated scheme, whereby both the client and the service class delegate to the export/interface class loader of the service. This is how services classes are shared with multiple clients. These new delegates can be added dynamically at any time. Our variant of delegation is deterministic and preserves all the requirements and semantics of class loading, but permits class loader schemes to be assembled dynamically.

Implementing the hierarchical delegation model

When implementing a ClassLoader, the "upward delegation" rule is that when any ClassLoader is asked to load a class, it first delegates to its parent, and continues only if the parent is unable to load the class. This means that if any class is ever loaded by a common parent, it will be the class used when resolving other classes. This prevents binding errors and enforces the primacy of system classes because the ordering is determined to privilege system class loaders.

Standard algorithm: With upward delegation the template algorithm changes to insert the delegation in `loadClass(..)` (here represented without full signatures and in text form for):

```
6.  loadClass(..)
7.              {check cache}
8.              parent.loadClass(..)
9.              findClass(..)
10.                         check files/URL's
11.                         ask parent to resolve/bind
12.                         {installs in cache}
13.             ...
```

Each class loader checks the cache, checks the parent, and only then tries to load the bytecode itself, handing the result back to the parent class for resolution. The system ClassLoader will attempt to resolve what's required at load time (super classes, etc.) and will add the invariants (re: method and field references) when the class is added to the cache.

So, given this, what classes will the VM load when trying to resolve a reference in a class? This is sometimes called the "visibility" or "binding scope" of the class. For the ClassLoader class under the hierarchical delegation scheme, it is:

```
14. the ability of a given class to bind other classes, as determined (recursively) by the
    parent class loaders and by a class loader's own sources of bytecode
```

You can compare class loading schemes in terms of the visibility they afford to classes. You'll find the EAS Services Architecture provides more sources, each of which has a finer granularity. (This granularity is a big contributor to the composability of services.) Our scheme and the visibility are more complex than the standard Sun scheme, but I hope you'll agree the expressive power is worth it.

Implementing the federated delegation model

The EnhydraClassLoader has a delegate list which it uses after delegating to the parent. ClassLoader's can be added to the end of the list at run time, but nothing is ever removed from the list. Once a class is loaded from a particular class loader, that loader will always be the first one capable of loading that class which is asked to load it - just as guaranteed by parent delegates. Consider the revised algorithm, with added steps in bold:

```
15. loadClass(..)
16.             {check cache}
17.             parent.loadClass
18.             findClass(..)
19.                 {check cache}
20.                 for each delegate: delegate.findClass
21.                 findClassSelf(..)
22.                         check files/URL's
23.                         ask parent to resolve/bind
24.                         {installs in cache}
25.             ...
```

First, we created a method findClassSelf which does what findClass used to. Subclasses of the EnhydraClassLoader override this in order to implement local resources, etc. This permits us to use findClass for recursive delegate calls.

You will note that the delegation happens in and to `findClass(..)`, not `loadClass(..)`. It's *in* `findClass(..)` rather than `loadClass(..)` because parents do not need to be searched during delegation.[2] It's *to* `findClass(..)` because delegating to `loadClass(..)` would change the semantics of `findClass(..)` to add a parent.`loadClass(..)`. But because the delegation happens in `findClass(..)`, we have the duplicate cache check for the initiating class loader, at a minimal performance penalty. In this way the EnhydraClassLoader can act both as a delegator (going through the delegate list) and a delegate (checking the cache in findClass).

[2] All EnhydraClassLoaders on delegation lists have our "top" class loader as their parent. Class loading is always initiated by a loadClass(), not findClass() (which isn't public). Since loadClass() guarantees that the parent chain is always invoked, and since the parent is the same, subsequent parent delegation can be skipped by calling findClass().

This scheme enforces the rule that a class loader always returns the same class for a given type name, both for itself and on behalf of all the class loaders that it delegates to. What's new is that an EnhydraClassLoader might be unable to load a class at one point but almost immediately thereafter be able to load that same class, by virtue of a new delegate being added to the delegation list. (Sun has similar semantics in the URLClassLoader by virtue of permitting new URL's to be added after the class loader is constructed.)

With delegate lists, you now know a bit more about how a client comes to see the service export classes after looking up a service in JNDI: the export class loader is added to the delegate list for the client during the lookup. For classes defined by EnhydraClassLoaders, you understand visibility now as "...determined by the ordered list of parents, *delegates,* and bytecode sources..." This makes it possible for clients to get dynamic scope.

EnhydraClassLoader in the EAS Services Architecture

EnhydraClassLoader for the service provider

Not only clients but also service providers use EnhydraClassLoaders. The service implementation class loader is an EnhydraClassLoader with the service export class loader on its delegate list. Thus the export class loader is on the delegate list of both the client and the service. This is how classes are shared.

This section discusses the technical details of how class loaders are set up for services and clients. It talks about the setup for services and for clients, and how the export class loader is added to the client delegate list during the JNDI lookup.

This section should help for writing services and clients, especially when writing service deployment descriptors. Programmers new to EAS are sometimes confused by how some descriptor elements are close but distinct ("load-service" and "bound-library"; "export-jar" and "service-jar"). More generally, this section should give you a model of what's going on so you can reason your way around the class loader issues in the services architecture.

Static context: setting up the service class loader

A service is implemented in a TargetAccessPoint (TAP)[3] subclass which is instantiated by EAS at run time. EAS first sets up your service classloaders appropriately and uses them to load your TAP class and create a TAP instance. The class loader used to load/define the TAP class is the "implementation" class loader.[4] As you know from the loading rules, every object it defines will search for classes starting from itself, the implementation class loader, so you can think of it as

---

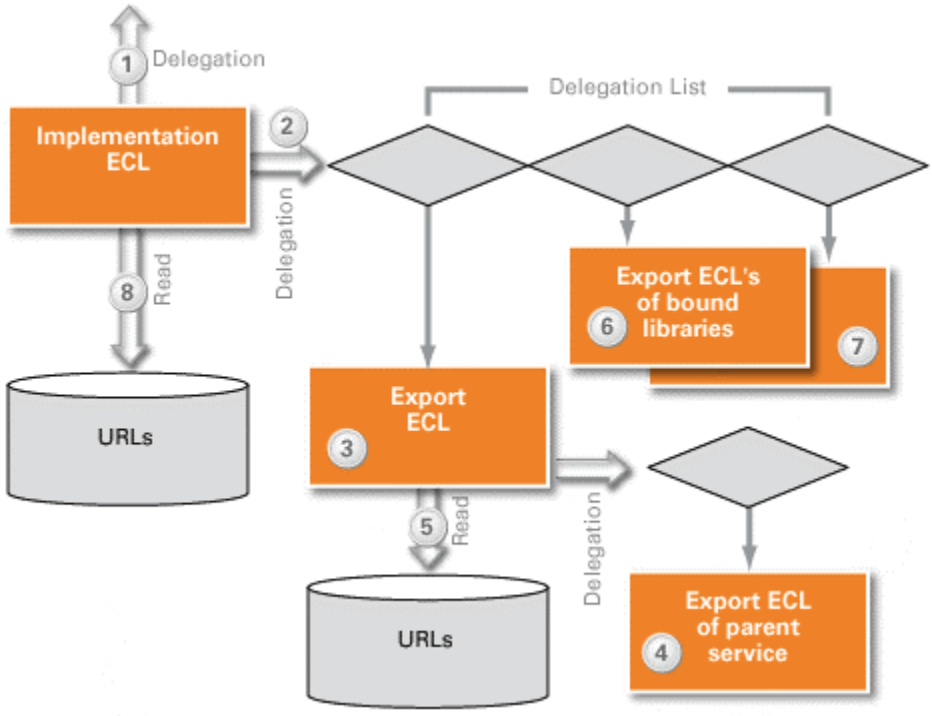
[3] The TargetAccessPoint (TAP) is so-called because it is the point where clients access service "targets", i.e., proxies. Your TAP class is essentially a factory for producing targets. When a client looks up the service in the namespace, there is no pre-existing target object waiting in storage; instead, there is a reference to your TAP factory that is invoked during the lookup to produce the target. The target is passed back to the client as a service proxy. In cases of service libraries, there is no target, but the class scope mechanisms are the same.

[4] The implementation class loader is sometimes called the "private class loader." Don't confuse that with any notion derived from Java's private access specifier. Also, the export class loader is sometimes called the "public class loader", or its classes referred to as the "service interfaces". Don't confuse that with any notion derived from Java's public access specifier or Java Interfaces, or with the more restricted notion of "service interface" element in the enhydra-service.xml.

the binding gateway for your service. Whatever classes you need should be accessible from the implementation class loader.

As an EnhydraClassLoader, the implementation class loader has a delegate list, and the first element on the delegate list is the "export" class loader. Classes will be defined from the export class loader before they would be defined by the implementation class loader, because a delegating class loader first defers to any delegates. Also on the delegate list are "bound" libraries. (A library is a service that presents only classes - no target proxy.) Bound libraries are services made available to the service implementation ECL only by putting the export class loaders for those services on the delegation list when the TAP implementation class loader is created. These typically represent classes your TAP classes require and are not visible from the export class loader. You add them by listing the services in bound-library elements of the enhydra-service.xml. These are all in the following diagram:

Diagram: service class loaders



You can see the numbers on the graph representing steps in the delegation order, so you can walk through a typical load operation:

	pseudo-code	explanation
0	((EnhydraClassLoader) privateECL).loadClass(..)	VM or explicit call to load class
1	{privateECL.}super.loadClass(..)	upwards delegation to parent
	{privateECL.}findClass(..)	privateECL.loadClass(..) calls privateECL.findClass(..)
2	{for each delegate, call findClass(..)}	privateECL.findClass(..) delegate loop
3	delegate[0].findClass(..)	exportECL checks cache...
	{for each delegate,}	...delegates
4	delegate[0].delegate[0].findClass(..)	... first delegate is export ECL of parent service
5	delegate[0].findClassSelf(..)	... then export ECL checking URL's
6	delegate[1].findClass(..)	boundLibrary1 checks cache, delegates, URL's
7	delegate[2].findClass(..)	boundLibrary2 checks cache, delegates, URL's
8	privateECL.findClassSelf(..)	privateECL checks URL's

The effect of this scheme is correct: classes will be loaded first from parents, then from cache, then delegates, and then from local URL's. Thus, service implementation loaders defer to their export loaders, and will load classes from there first. Further, specification class scope (in the enhydra-service.xml descriptor) will take precedence over run-time class scope (via naming lookup). This gives the service developer the ability to control visibility while permitting incremental visibility, e.g., during implementation upgrades.

Clients also have a delegate list, which after lookup includes the export class loader. Hence, after the lookup clients will similarly load the export classes first from the export class loader. This is how services and clients share common classes, and the ordering is why it works. Essentially, the export class loader acts as a local "system" class loader, taking precedence in binding for the community created by a service and its clients.

Upgrade: Now you can glimpse how EAS can upgrade the implementation without disconnecting clients. Clients hold references to classes in the export class loader, but the implementation class loader can be removed and replaced, adopting the old export class loader on its delegate list. This means existing clients can continue to use their old targets and also get new targets without getting ClassCastException's from the system's ClassLoader's cache-based invariant enforcement mechanism.

However, this glimpse is incomplete. Which client class loader gets the service export class loader on its delegate list? Objects the client holds may come from many class loaders, but any client object in any thread should be able to use the target and hence needs to resolve the export

classes. For you to understand the answer to that, I should explain how EAS uses the context class loader in implementing thread-based scoping for applications and services.

Dynamic context: setting up the client class loader

In Java 2, Sun implemented context class loaders to permit dynamic scoping. These follow the thread of execution as a backup class loader to the hierarchy available from the current object.[5] The EnhydraClassLoader delegates to the current context class loader as the last phase of the loadClass(..) process. We can thus again amend our definition of class visibility, which now we understand is "... determined by the current class loaders for the parent, the object and its delegates, *and the context*..." The complete algorithm is thus:

```
26. loadClass(..)
27.           {check cache}
28.           parent.loadClass
29.           findClass(..)
30.               {check cache}
31.               for each delegate: delegate.findClass
32.               findClassSelf(..)
33.                       check files/URL's
34.                       ask parent to resolve/bind
35.                       {installs in cache}
36.           ...
37.           contextClassLoader.findClass(..)
38.
```

This means that the EnhydraClassLoader is compliant with any class loader set as the context class loader. But the EnhydraClassLoader can and does act as the context class loader, providing a dynamic naming scope.

EAS creates a single EnhydraClassLoader to serve as the context class loader for each "client." Whenever that client thread does a JNDI lookup of service, our naming system adds the export class loader to the delegate list of the EnhydraClassLoader. Any object in the service or application will then be able to load classes from the looked-up service's export class loader.

To associate this single EnhydraClassLoader with a client, EAS uses its control over threading. Clients are either service threads or application threads; EAS manages applications and services primarily as thread aggregations. Any new Thread, whether created by a call to new Thread(..) or using our thread service, adopts the context class loader of the creating thread. When EAS starts applications or services, it sets up the application/service context class loader for the initial threads, and the same class loader instance follows any threads created by the service or the client. As a result, anything running in the thread of a service or application will be able to load classes from the export class loaders of any services looked-up from that service or application. In this sense, services are acting as clients to other services in the same way as application clients are.

Note that this makes the term "client" somewhat ambiguous, in that it's used both for "the service-user" and "the identity associated with the CCL-context." For example, when

---

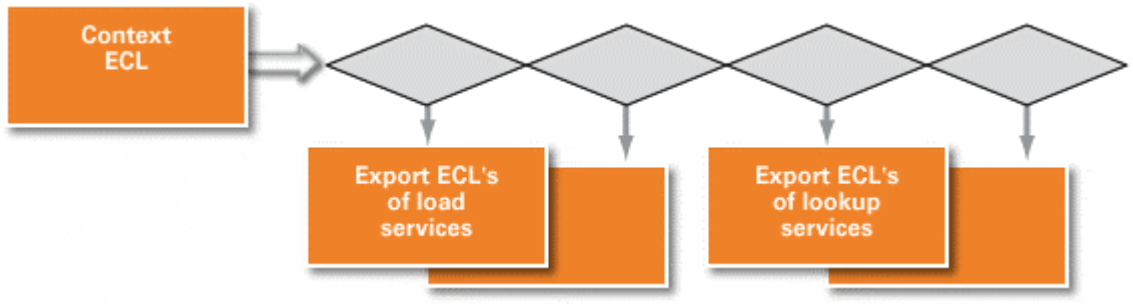
[5] This was needed to support situations where objects running in a thread required visibility into a broader scope than their own ClassLoader. This is necessary, e.g., for the RMI class loader running under the system class loader to be able to access client scope in order to deserialize classes.

application code running in an application thread uses a service target, the application is the client in both senses. But when that same service target in turn invokes another service target (perhaps in a way unknowable to the application client), then the invocation is running in the application thread, but one might consider the initial service target the client of the other service target.

Since we have an EnhydraClassLoader acting as the context class loader in every service and application thread, we have a mechanism for adding dynamic context: When any client (whether service or application) does a lookup in the naming system to another service, the naming system casts the current context class loader to an EnhydraClassLoader and adds the export class loader of the looked-up service to the delegation list of the context class loader, making it available throughout the service or application threads.

Just as you can specify bound libraries for the service class loader, you can specify services to be pre-loaded into the context class loader. The result is a delegate list with the class loaders specified by load-services followed by the export class loaders of the services that are looked up in JNDI programmatically.

Diagram: context class loader



When the EnhydraClassLoader is used as a delegate context class loader, it has no bytecode resources of its own. Further, it delegates to the same "top" class loader as others and is ordinarily invoked via `findClass(..)`, so it effectively does no parent delegation. This does not affect the correctness of the upward-delegation scheme because the context class loader is guaranteed to be the last class loader invoked, after which the "top" class loader (and its system parents) have already been invoked.

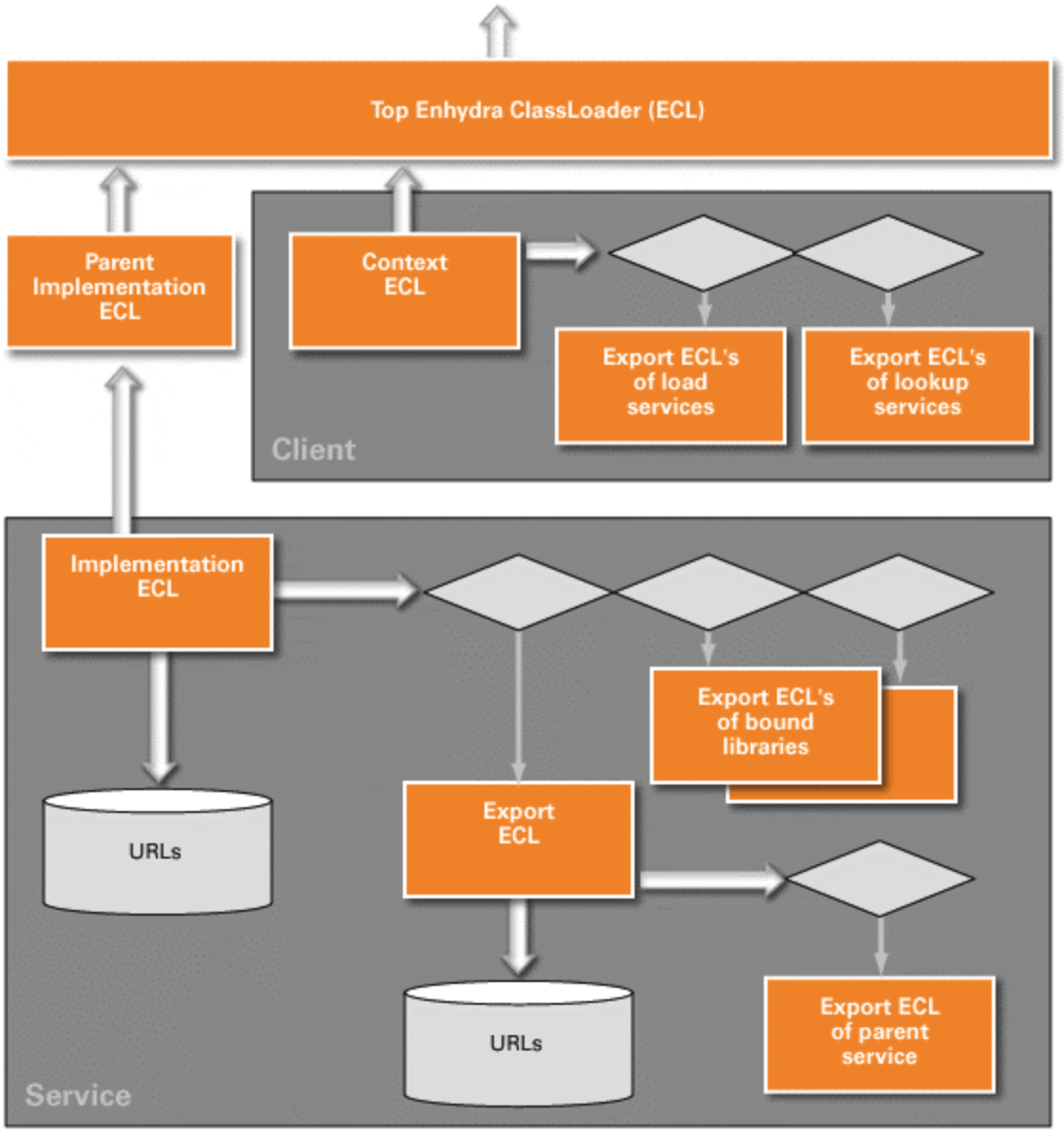
Since this context class loader is available in any client thread, the class loaders on the delegate list are available when executing in other objects. For example, when the client invokes a proxy target, the target will first resolve classes through its defining loader (the implementation class loader) and then through the context class loader.

The rules for when to use load-service or bound-library are relatively simple. Use bound-library if you need classes to be visible from your TAP classes. Use load-service if you need classes to be visible in any of your threads, e.g., even when executing objects not created by the TAP implementation class loader. (I remember this by thinking of threads dynamically loading services, but libraries being bound to the defining class loaders.) If you need the same service for both, put it on both lists; the same class loader will be used in both places, so the classes will resolve correctly both for clients and when the target is calling back into the TAP.

All together now

Let's review the whole class loader scheme and elaborate a complete description of class visibility in EAS.

Diagram: EAS class loading scheme



Client context class loader: This delegates to the export class loaders of the "load-services" and any services looked-up. The "load" services are also specified in the enhydra-service.xml. As a delegate of the service's context class loader, their classes are visible in any thread of the service. Both services and applications have a context class loader. Whenever they look up a service (i.e., act like a client to the service), the export class loader of the service is added to the delegate

list if it is not already there.[6] Unlike most class loaders, this context class loader does not load any bytecode and effectively does no upward delegation, acting instead as a representative of the class loaders on the delegate list.

Implementation class loader: Like most class loaders, it has a parent and delegates upwards first in loadClass(..) calls. Normally, the parent will be the "top" EnhydraClassLoader, whose parent is the system class loader. However, if you specify a parent in the enhydra-service.xml descriptor, the parent will be the implementation class loader of the parent service. As an EnhydraClassLoader, it also has a delegate list, which contains first of all the export class loader. If you specified a parent, then the export class loader has the export class loader of the parent service first on its delegate list. The classes directly defined by both the implementation and the export class loader are specified in the enhydra-service.xml URL entries for the service-jar and export-jar elements.

On the implementation class loader delegate list, note the "bound-libraries." These are the export class loaders of services listed in the enhydra-service.xml bound-library elements, and they are added to the delegate list before the TAP is constructed. As delegates of the implementation class loader, they are visible to any object defined by it, no matter what thread the object is running in. Thus, if your target implementation depends on a service classes which it does not look up directly, then the required service should be listed as a bound library so its export classes are visible even when the target is running in a client thread, which may not have looked up the required service. Notice the difference between having a parent service and using a bound-library: bound libraries only see export classes of the desired services, while the parent permits your service to see anything visible through the parent's implementation class loader.

For services, the implementation class loader scheme defines the static binding scope. For both services and applications, the context class loader scheme defines the dynamic binding scope. The dynamic scope follows the application or service in any of their respective threads. (The static scope of applications is the subject of a later article.)

Writing services, mindful of binding scope

Class visibility in the Services Architecture

I can now compare the complete definition of class visibility in EAS with the common programmer's view and our initial definition. This derives from the combination of the revised algorithm above and the class loader layout.

Common view	`System classes and all the classes in the class path`
Initial definition	`the ability of a given class to bind other classes, as determined by the parent class loaders and by sources of bytecode`
EAS definition	`the ability of the currently-executing class to bind other classes, as determined by` `(a) the current contents of the Class cache;`

definition	(b) the result of its parent class loader's loadClass(..) call, including any recursive loading; (c) the result of any delegate class loader findClass(..) calls, taken in delegate order (for the service implementation class loader, the list of the export class loader and any bound-libraries); (d) the set of bytecode resources available to this class loader and the ability of the superclass to define the class from any such bytecode; and (e) the result of a call to the current context class loader associated with the thread of the currently-executing service or application (i) of loadClass(..) if not an EnhydraClassLoader, or otherwise (ii) of findClass(..), which will in turn call findClass(..) on the list of export class loaders of the specified pre-load services as well as any services looked-up from Threads created by the current service or application.

Most programmers do not need to know this definition. To use a service, you need only follow the rule that to use a service, you need to have looked it up in JNDI at one point in the lifecycle of the application or service. To write a service, you need only know to deploy using enhydra-service.xml, which requires you segregate export classes from implementation classes, and where you can identify dependency (implementation or export) on other services.

Debugging

In the event that you do write services and get ClassCastException or ClassNotFoundException, you will recognize the elements of the volumnious debugging options available. Two that I recommend:

1. In the multiserver script are variables controlling the logging of the class loader scheme for given events. For example, you can log the entire class loader layout whenever there is a ClassNotFoundException.
2. In the LMC you can view attributes for each services listing the classloaders. Each class loader also lists the URLs registered with it.

In any case, as a service architect, you might want to take care in where classes are defined, to give yourself the most flexibility in upgrading your service later.

Upgradability

Services can be undeployed completely, including the export class loader, so you can always replace everything. But if you change the export class loader, existing clients will not be able to use the newly-deployed version of the export classes. Any client class bound to a class in the export class loader will be unable to bind with a class of the same type in the new export class loader. Thus, to maintain existing clients, you should retain the export class loader when undeploying and adopt it when deploying.[7]

In order to upgrade the most possible of your classes, you might want to include only the minimum required classes in your export class loader: whatever is publicly reachable to the

[7] Before you undeploy a service, you can choose whether to remove or retain the export class loader using the KeepExportClassLoader attribute in the ServiceManager MBean. If a class loader has been retained, on redeployment it will be adopted. This mechanism will likely change.]

client from the target interface.[8] What's publicly reachable is the set of classes traversed by traversing the public fields and the return and parameter types of public methods of the service target interface class and any class so traversed.[9] Any other references made by the target object to references will be resolved initially by the service implementation class loader, even when running in client threads.[10] In the "Resource" section below is a link to a small program that calculates the set of export classes for you.

You can probably imagine the caveats to the rule for minimum-required classes. For example, if your old targets interoperate with classes from the new targets or the new TAP, then those classes also belong in the export class loader. Even though the classes are not seen by the client and can be loaded without being in the export class loader, they must be the same Class instance in order to interoperate or you will get a ClassCastException. For example, in the case of a target that acts as a proxy for a callback to the TAP, on undeployment you would replace the old reference with a balking or no-op implementation, and on redeployment you would replace that reference with the new operative reference. You can do this only if the callback class is defined in the export class loader (or some other common class loader, as Runnable is defined by the system class loader). But for most service writers, the minimum-required rule is easy to calculate and suffices.

Composability

Why not put all the classes in the export class loader? Having to bring down the application to upgrade it does not seem like a burden; most people are used to it now, and upgrades can be planned during slow periods or when cover is available within a server cluster. This may be true, but it misses the real benefits of the class loader scheme: increased design granularity and stronger semantic encapsulation of services.

System composability is related to component modularity, but it measures the ease with which modules can be integrated, including assembly, validation, management, and evolution. It is a property of systems, not of modules, and spans the lifecycle of the system and any system it evolves to or coordinates with. Upgradability contributes to composability because it eliminates downtime for interface-compatible changes. Composability contributes to upgradability because the units for upgrade can be smaller, rather than having to upgrade entire systems or applications. Many people are attracted to the EAS Services Architecture because it solves their upgrade

---

[8] "Service interface" here is a technical term meaning the classes of the target object specified in the enhydra-service.xml service-interface element. For ordinary, unwrapped targets this should be the class of the target that your TargetAccessPoint factory returns from createTarget(..). For wrapped targets, the dynamic proxy wrapper will have this class and delegate to the actual target returned from createTarget(..). With wrappers, you can specify multiple Interface classes in order to wrap multiple interfaces around a single target.

[9] This definition of minimal assumes your client is not deserializing target objects and is not in the same package or a subclass of the service target interface. If your client has more than public access to the target, you will have to include anything reachable from that access level. If your client deserializes target objects, you would have to modify the definition to include anything the target object requires for deserialization, i.e., the transitive closure of the set of target and target parent classes and non-null references in the target object. (RMI and hence EJB's do deserialization, but EAS 4 handles the deployment of EJB's specially. That will be the subject of a later article.)

[10] This assumes the defining loading is the implementation class loader. This is ordinarily the case for TAP objects and their progeny, but if the defining loader is some other class loader (say, the export class loader), then that will be the starting point for resolution.

problems. But they become committed to it when they begin to realize the benefits of composability.

In Java enterprise computing, there has been a disconnect between the design unit (a component) and the run-time unit (the object). Except in rare cases, components are not actually encapsulated by objects. Even worse is when the component model breaches object encapsulation. EJB does this in its design when requiring public methods and fields to support EJB container operations; the leading EJB vendors do this in their implementations when providing web container clients with unlimited visibility into EJB implementation classes. Further, most seasoned designers of EJB's find themselves designing around the implementation limitations, e.g., creating bulk interfaces to avoid excessive database updates. EJB component designs are often determined by OR-mapping and EJB container semantics, rather than vice-versa. EJB, like XML, is limited by being a data abstraction, an interface back to the relational database.

XML-based Web services, while not directly comparable to EJB's, seem by contrast quite attractive because they present a much richer semantic interface than possible from EJB. Unlike EJB's, they do not unnecessarily restrict the invocation layer and hence can integrate more disparate systems. They make it possible for cross-business collaboration between partners and through supply and workflow chains. But like EJB's, web services come at the cost of encoding, of rendering objects into something else (XML) and back again - a potential loss both in semantics and in performance, not to mention security.

The EAS Services architecture advances composability significantly with the dynamic association of binding invocation context and by enforcing a distinction between implementation and interface classes in services. By associating binding context on JNDI lookup to a service, EAS provides the runtime flexibility of a web services architecture. By using delegation to class loaders rather than copying library jars, EAS enables you to safely share services across applications and with the system, avoiding duplication. By segregating implementation and interface at run time, EAS presents in live systems exactly what the service designer ordered - the required classes are available and other classes are not - providing a level of correctness missing from the EJB specification. This reserves for the developer the freedom to upgrade long after the initial deployment, and sets actual boundaries for the run-time components.

Thus, the class loader scheme enables the EAS 4 to present a set of composability features unmatched elsewhere:

- Component designs are not entangled with extrinsic concerns of data encoding or interface granularity
- Component form is not determined - libraries, proxies, call-backs, and respondents are all supportable
- Component resolution can be smaller because deployment units are smaller
- Component boundaries are verified at compile-time and enforced at run-time
- Component deployment units are not monolithic
- Component type and instance numericity is undetermined; 0..n services and targets can even be shared

- Component assembly is easy and intuitive, both programmatically and declaratively
- Component upgrade is possible at runtime, providing unlimited integration time

The EAS class loading scheme is not the only contributor to composability. Most notably, service invocation decorators and interceptors extend composition and management to the method level and provide a new mechanism for layering in aspects of system behavior that would otherwise be scattered or unmodifiable. Pervasive management makes run-time system analysis and re-composition possible by providing both the data to make assessments and the controls to implement decisions. The class loading scheme decouples granularity and scope to provide more freedom; other features add views and controls to add power.

But beyond features, subsystems, and 'ilities, is feel. While understanding the EAS Services Architecture is half the fun, the bigger half is using it. It's intuitive and easy to program. Things work when they're supposed to. Gone are the integration nightmares of mentally tracking back through encoding and communications layers. It puts the fun back into writing enterprise software!

Freedom to innovate

The primary design goal of the EAS Services Architecture was to develop a services architecture which is composable and upgradable, to facilitate rapid development and redevelopment of the server and applications. Its first incarnation, EAS 4, supports J2EE, SOAP, and the Enhydra application programming model. It also exemplifies the EAS Services Architecture as a real contribution to the state of the art in Java enterprise computing, in part because of the dynamic federated class loading scheme. This scheme presents a simple use model to the application programmer, an intuitive design model to the service developer, a high-level composable component model for the system architect, and a powerful upgrade model to the systems administrator - all by virtue of the careful implementation of federated class loaders in accordance with Java's class loading requirements and the needs of enterprise applications. This care prevents the services model from being brittle or exposing implementation holes. Services are fun to work with and easy to evolve; in the race for more features (and fewer bugs), you can get started fast and stay ahead. EAS 4 supports well-known programming models, giving you the power to be productive with existing components, tools, and concepts. It also presents a new way to design and develop; with both compatibility and flexibility, you have the freedom to innovate.

Resources

JSR 111 is the Java Specification Request for Java Services Framework. Lutris is on the expert committee. http://www.jcp.org/jsr/detail/111.prt

The Java VM Specification section 5.4.3 defines class resolution. http://java.sun.com/docs/books/vmspec/2nd-edition/html/ConstantPool.doc.html#7349. Section 3.6.3 defines dynamic linking of method references. http://java.sun.com/docs/books/vmspec/2nd-edition/html/Overview.doc.html#1963

Binding invariants are discussed in Sheng Liang and Gilad Bracha, Dynamic Class Loading in the Java Virtual Machine in "Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications," October 1998. A postscript version is available from Gilad Bracha's home page at http://java.sun.com/people/gbracha. The VM spec discusses binding invariants in section 5.3.4, http://java.sun.com/docs/books/vmspec/2nd-edition/html/ConstantPool.doc.html#78621

"The safe network download and execution of platform-independent bytecode" is the answer I remember when one of the original Java product managers was asked about the design goals of the Java VM. James Gosling has what he calls the original white paper linked off http://java.sun.com/people/jag, and the VM specification has an introduction listing its goals.