# Tailoring Java for a Pervasive Service Infrastructure

Philippe Bernadat, Ira Greenberg, Alan Messer, Dejan Milojicic
Mobile Systems and Services Laboratory
HP Laboratories Palo Alto
HPL-2002-24
January 31st , 2002*

E-mail: [bernadat, iragreen, messer, dejan] @ hpl.hp.com

Internet, service, pervasive, infrastructure, Java, interposition

A growing number of mobile computing devices are becoming available that can access large amounts of data and services over the Internet. While Java appears to be an appropriate platform to deal with diversity, our experience reveals that in a mobile environment it has insufficient support for system facilities such as remote storage, disconnected operation, and concurrent execution of multiple services. We believe that these facilities can be provided transparently and efficiently through the use of API interposition. We investigate the use of interposition to provide access to a remote storage service, to implement a cache for data and services, and to help isolate services that are sharing a Java virtual machine. We describe how our approach is implemented and present some experimental results for CPU and memory usage. Based on our results, we believe that API interposition will make it possible to support these infrastructure features transparently and efficiently with an acceptable overhead.

* Internal Accession Date Only
Approved for External Publication
© Copyright Hewlett-Packard Company 2002

# Tailoring Java for a Pervasive Service Infrastructure

Philippe Bernadat, Ira Greenberg, Alan Messer, and Dejan Milojicic
*[bernadat, iragreen, messer, dejan]@hpl.hp.com*
*HP Labs*

## Abstract

A growing number of mobile computing devices are becoming available that can access large amounts of data and services over the Internet. While Java appears to be an appropriate platform to deal with diversity, our experience reveals that in a mobile environment it has insufficient support for system facilities such as remote storage, disconnected operation, and concurrent execution of multiple services. We believe that these facilities can be provided transparently and efficiently through the use of API interposition. We investigate the use of interposition to provide access to a remote storage service, to implement a cache for data and services, and to help isolate services that are sharing a Java virtual machine. We describe how our approach is implemented and present some experimental results for CPU and memory usage. Based on our results, we believe that API interposition will make it possible to support these infrastructure features transparently and efficiently with an acceptable overhead.

**Keywords**

Internet, service, pervasive computing, infrastructure, resource, concurrency, mobility, Java, interposition.

## 1 Introduction

Pervasive computing is increasingly becoming a reality, with the promise of allowing people to access any service, anywhere, at any time. A growing number of mobile computing devices are becoming available that have the ability to access data and services from the Internet. This significantly increases the variety and number of services or applications that can be accessed from any mobile device that supports standard Internet protocols. Throughout the paper, we refer to services and applications interchangeably.

We would like to support the pervasive computing paradigm where devices are anonymous “empty” boxes, and users download the services and data they need from the Internet. A user would select a device and request a service, and it would be discovered, downloaded, and installed, perhaps with the help of a service broker. The downloaded program could be a complete, stand-alone service or the client portion of a service. As the service executes, user data would be written and stored persistently on a remote storage service based on Internet standards so the data can be accessed later from another device. In this way, a user could select any device, customize it, and use it to access any previously written data or to resume a service.

Wireless devices have several limitations when compared with wired desktops. First, disconnection from the Internet may be more frequent, or may even be desirable for standard operation. For example, an area may lack sufficient coverage or a user may intentionally disconnect to save power or communication charges. Second, devices may lack the storage to hold large numbers of services or large amounts of data. Third, it is easier to lose or damage portable devices, which can lead to the loss of data stored on the device.

In addition, mobile, computing devices are very diverse, and they are becoming more so. They cover a wide range of functionality from cell phones and Personal Digital Assistants (PDAs) to laptops, and include a variety of processors, memory units, disks, communication channels, operating systems, etc.

These factors indicate that infrastructure support is needed for remote storage and disconnected operation. The ability to access a remote storage service would allow a user to access personal and service data from any available device and to store and work with large amounts of data. This could also prevent from losing data if a device is damaged or lost. Some services work well, and make sense to use, even when the user is disconnected from the Internet. Disconnected operation would allow the user to continue to work while communication is unavailable, undesirable, or unnecessary.

An obvious choice for programming diverse devices is Java. Java is an accepted, mature technology that is being used to write an increasing number of Internet-based services. However, while Java has worked well for servlets and applets on desktops, it is missing some key functionality to support mobile services. In particular, we believe that Java requires additional transparent support for remote storage and disconnected operation.

Another concern with Java is execution overhead. Java already suffers a performance penalty because it is interpreted or compiled on the fly. We believe that it is

important to not introduce significant additional overhead. For example, users would like to simultaneously execute multiple services on a device. It would be unacceptable to start a new Java Virtual Machine (JVM) for each of the services that is executed because of the start-up and memory overhead. However, having multiple services share a JVM requires infrastructure support for sandboxing, that is, for multiple namespaces, security, and concurrent execution. Sandboxing can be provided by enabling the Java security manager to use a different class loader for each service, but this doesn't solve the problem of conflicting properties.

Our goal is to provide general, transparent support for this environment. We believe that it is important to provide a solution that does not rely on platform-specific features, change the Java specification, change the JVM implementation, or require services to be written in a special way.

Our approach is to provide infrastructure support for remote storage, disconnected operation, and concurrent execution, through the use of API interposition. This will allow appropriate pieces of the infrastructure to be transparently slipped into the execution of unmodified programs. For example, APIs that access storage can be replaced with accesses to a remote file system; services and data can be stored into and accessed from a cache, which will help with performance and disconnected operation; and accesses to user, system, and service properties can be mediated to support secure, concurrent execution in a single JVM.

The paper is organized as follows. Section 2 describes our vision for pervasive service access, its requirements, and its general architecture. Section 3 examines the suitability of the standard Java platform for our requirements, and the deficiencies we believe exist. Section 4 describes our approach to overcoming these deficiencies using API interposition. Section 5 discusses the implementation of our approach. Section 6 provides our experimental results for CPU and memory usage. We discuss the lessons we have learned from this investigation in Section 7. Section 8 compares our approach to related work, and Section 9 concludes the paper and proposes some future work.

## 2 Requirements

We believe that in the future applications will be deployable as services to a diverse set of devices. We envision many kinds of services being delivered in this way, from small games to enterprise applications. Such services may be delivered over the Internet either in the form of self-contained code or as client code for a network-based service. Overall we foresee an architecture where services can be located with service brokers and

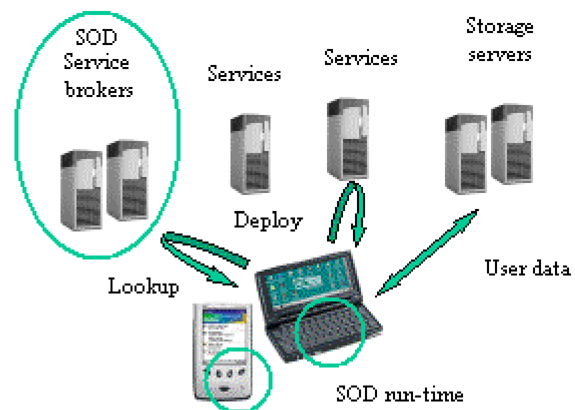


**Figure 1: SoD overall architecture**

deployed at runtime into any device with a standard-based runtime (see Figure 1).

We believe users of future pervasive services will want to be able to use services on devices with the following characteristics:

- **Anonymous Use** — Any user can use any service from any device. As a result, use of any service may be resumed on any other device, without worrying how to retrieve or synchronize preferences and data.
- **Device Independence** — Any device is as good as another no matter what device hardware is used. Processor type, storage capacity, etc. should not determine which devices can be used.
- **Multiple Service Execution** — Users will be able to run multiple simultaneous services on any device.
- **Secure, Persistent Data** — A persistent version of data is always transparently maintained to ensure the system will tolerate data loss. Data should be stored separately per user and per service to support security.
- **Tolerant to Disconnection** — Due to coverage limitations or user preferences services should be to tolerate occasional periods of disconnection.

However, we also believe that to be widely adopted, these characteristics must be supported on devices within the following constraints.

- **Legacy Support** — Legacy services must be able to execute on the largest range of devices, especially personal and mobile devices.
- **No Significant Additional Overhead** — Device runtime support should not add a significant amount of performance or resource overhead to already limited devices.

Supporting such a comprehensive set of features under these constraints requires a comprehensive service infrastructure. Investigation of this infrastructure is part of the Service-on-Demand (SoD) project that we are currently undertaking at HP Labs. In this paper, we focus

on the device-side support required to provide these features by using as much existing infrastructure as possible.

## 3 Platform Analysis

Several systems meet some of our requirements, such as Sun's Java and Microsoft's .NET Common Language Run-time (CLR). When this paper was written, it was too early to fully analyze and experiment with .NET, so for the purposes of this paper we consider Java as a suitable platform. Java is also a good choice due to the large number of existing applets and applications for various devices.

Looking at off-the-shelf Java environments, we believe they provide good support for device independence using abstract bytecodes. However, we believe they do not meet our other requirements for two reasons.

- **Lacking pervasive environment support.** Supporting anonymous use, persistent data and disconnected operation requires support that we believe is missing from the existing Java platform. Java platform implementations typically store data locally through several APIs preventing anonymous use and persistence across devices. Some Java installers cache bytecodes, but do not consider user data or other content. We believe that it would be difficult to support these features unless changes are made to the source code of legacy applications or to the JVM. We intentionally exclude the idea of proposing new APIs for mobile services because it is difficult to get new APIs widely adopted. In addition, we do not think it is appropriate to modify virtual machines because getting new JVMs adopted for the large diversity of devices will be difficult in practice.
- **Minimizing resource and performance overhead with multiple services**. Java is commonly used for applets and servlets, but less frequently for applications. To convince users to switch from platform native applications (such as those for Windows, Windows CE, and Linux) to Java applications, the users must perceive that the execution performance will be comparable. It is well known that executing Java classes causes some performance penalty because the code is either interpreted or compiled on the fly. Adding a noticeable amount of additional overhead must be avoided.

  We believe that starting a new JVM for each service is unacceptable because of the additional storage and execution overhead. To investigate this belief, we sequentially executed two instances of a 6KB Java program named *SimpleTextEditor,* and measured how much memory would be incrementally consumed. The results are reported in Table 1 for two types of PDAs and a standard workstation. We found that a significant amount of memory is consumed and that it would severely limit the number of concurrent tasks on a memory-constrained device.

| Number of Instances | Jornada 720 StrongArm | | Jornada 548 SH3 | X86 PIII 600 Mhz |
|---|---|---|---|---|
| | Pjava | ChaiVM | Pjava | JDK1.3 |
| 1st | 3452 | 3498 | 2398 | 9592 |
| 2nd | 1524 | 2004 | 2398 | 9328 |

**Table 1: Memory usage per program (KB)**

We then examined the startup time. We wrote a simple Java program that reads a given file's last modification date, displays it, and overwrites the file. We ran several instances of this program sequentially. The results are shown in Table 2. Given the very small processing performed by this program, the elapsed time approximates the startup time for the virtual machine by itself. On a PDA device, the JVM's startup time varies between 2 and 4s, which would add significantly to the application startup time if a JVM where used for each application.

| | Jornada 720 StrongArm | | Jornada 548 SH3 | X86 PIII 600 Mhz |
|---|---|---|---|---|
| | Pjava | ChaiVM | Pjava | JDK1.3 |
| VM startup time | 2000 | 4000 | 3000 | 200 |

**Table 2: Virtual machine startup time (milliseconds)**

The additional memory overhead and, to a lesser degree, the additional startup time overhead would significantly degrade the overall performance. We believe that the best solution is to execute all of the services within a single virtual machine. The fact that some embedded devices may not be able to concurrently execute multiple virtual machines supports this approach. Using a single virtual machine also greatly simplifies sharing system components such as caches.

### 3.1 Mobile Environment

Managing user data well in a mobile environment places strong requirements on the infrastructure. Access to remote resources (such as Java classes, user data, and URLs) must be detected and locally cached so that disconnection will be less likely to cause services to fail and so that runtime performance will remain acceptable.

**Remote storage.** Roaming must be supported. A user's personal data and profiles should be available on any device that the user will employ, and the data must be stored persistently and securely. We believe that this should be accomplished by using a third-party storage service to store and retrieve the data. Because we want

to support legacy services, we do not want to impose any software changes to provide remote storage. In addition, most embedded and mobile devices do not offer such a remote storage facility, and even if they did, it would probably be incompatible from device to device.

Consequently, the remote storage facility must be part of the SoD infrastructure and should not rely on any platform-specific support. That is, it must be pure Java.

We do not intend to develop new protocols or distributed file systems. Instead, the infrastructure is designed so that any off-the-shelf software that implements a remote storage service can be plugged in. In our prototype, we use an FTP Java client developed by IBM [1] as the storage service. The base runtime only contains the modules to cache and synchronize the data. Our prototype synchronizes storage at the granularity of a file because this is the granularity used by FTP, but nothing prevents the prototype from supporting different policies.

Applications use the standard `java.io` package to store data. When methods from this package are invoked, the infrastructure transparently loads the files, and refreshes and updates them on the remote storage server. Similarly, other methods such as `java.awt.Toolkit.getImage` need to be redirected to the remote server.

**Disconnection.** Continuous connectivity to the Internet remains expensive and is not guaranteed. However, some services can proceed locally when the user's device is disconnected from the Internet if the necessary data and code is cached on the user's device. The cache should also be regularly synchronized and flushed, as required. Our infrastructure caches three types of data:

1. **Application Java classes.** A special case of URL content, whose refresh period can be arbitrarily large.
2. **User data.** Must be synchronized periodically with a rather small period. In our prototype, FTP URLs are used to support file reads and writes.
3. **Other URL content.** Non-static content must be refreshed adequately and URLs using the HTTP protocol's PUT command must bypass the cache.

The cache also contributes to better performance, especially when restarting a service and accessing user data.

## 3.2 Concurrency within a Single JVM

To execute multiple applications concurrently within a single virtual machine, they must be carefully isolated into what we refer to as sandboxes [24] so that they will execute as securely as if they are not sharing the virtual machine. Without this support, the applications would share the namespace, the file system, the thread pool, system-sensitive APIs such as `java.lang.System.exit`, and system properties. We do not consider scheduling, resource management, or denial-of-service issues resulting from the sharing of CPU and memory [3, 4] because the device is dedicated to a single user.

Namespace isolation is mandatory because package and class names may collide, but it also helps prevent applications with different trust levels from communicating. Sharing a file system between applications can also lead to security violations. Notice that the security problems associated with sharing a file system can occur even when the applications are executed in separate virtual machines.

**Disjoint namespaces.** In Java, class loaders can be created to load classes in any way desired and to establish private namespaces. We instantiate one class loader per application to guarantee that the class or package names for distinct applications will not collide. The class loader attempts to load the classes from the locations returned by the service broker. It can download both applications and applets as jars or collection of classes. Each service is started from a thread in a new thread group associated with the namespace.

There is a one-to-one correspondence between a class loader, a namespace, and a sandbox. Any entity that is private to an application (besides its classes) is part of the sandbox. Examples of entities are a thread or a property such as `user.home`. At any time, system APIs may need to retrieve information from the sandbox. The thread context cannot be used to reliably identify a sandbox because packages such as AWT dispatch events through a system-distinct thread. This problem is solved through the use of a security manager as described in the following sections.

**System sensitive APIs and properties.** A Java application can create a single security manager [11] to control access to any system-sensitive API. Within the security manager, it is possible to walk through the invocation stack to retrieve the class loader (i.e., the sandbox) of the requester, and use this information to grant or deny access to the API. This is appropriate for some APIs such as `java.lang.System.exit` where the security manager will deny the access and destroy the sandbox instead of shutting down the virtual machine. A problem with a security manager is that it can only allow or deny access to a file system; it cannot redirect access to a remote file system.

**Resource access control.** Because applications with various levels of trust can run simultaneously, the resources that each application can access must be iso-

lated. For storage, the ideal approach would be for the user to specify the storage path (or set of files) that the application is authorized to access. However, the infrastructure could automatically create a distinct storage space for each application by an approach such as adding a directory name to each storage path.

The Java Security Manager makes it possible to grant or deny access to a given API or resource, but it doesn't allow the API's behavior or the resource's name or location to be altered. The application usually opens user-private files through some dialog and intermediate files from the current user directory. We want to dynamically change this behavior so that file access is performed relative to some storage private to the application and its sandbox.

**Properties.** Applications access properties, such as `user.dir` or `java.io.tmpdir`, through the `java.lang.System` package. The values of these properties need to be distinct for each application. The security manager is contacted on each property request and is able to either grant or deny the access. This behavior is insufficient. We need to maintain a property set for each sandbox so that applications do not share the same storage space. The appropriate private property value must be returned instead of the common system value. One way to solve this problem is to overwrite the shared system properties with the private sandbox properties before granting access, but there is no guarantee that the private properties will not be changed by a concurrently executing application before the initial application can access them.

### 3.3 Affected APIs

Based on our analysis in sections 3.1 and 3.2, we identified the Java APIs for which the infrastructure must perform some special task. These APIs are summarized in Table 3. We did not employ any formal method to identify all the Java APIs that access resources, so there may be other classes to consider. For instance, `java.awt.Toolkit.getImage()` may be implemented through the instantiation of a `java.io.File` object, but this cannot be guaranteed. Even if it is, it would be difficult to modify the `Toolkit` standard class, as we explain later. Instead, we must consider the `Toolkit` class as accessing resources.

We restricted our API coverage to Java 1.1. For later revisions, other system packages such as Swing would require the same sort of analysis. For instance, the file dialog class of the Swing package would require similar interposition to the AWT file dialog class.

| Class | Reason |
| --- | --- |
| java.io.File<br>java.io.FileInputStream<br>java.io.FileReader<br>java.io.FileWriter<br>java.io.FileOutputStream<br>java.io.RandomAccessFile<br>java.io.PrintWriter<br>java.util.zip.ZipFile | User files are cached locally. The cached local file is extended so that it can be filled/refreshed/flushed from/to a remote storage server. |
| java.awt.FileDialog | The dialog browser must browse the content of the remote storage server. |
| java.awt.Frame | Our infrastructure is 1.1 compliant but requires the 1.2 getFrames() API. It must be modified to keep a frame list. |
| java.awt.Toolkit | getImage() must look up remote storage using the cache. |
| java.lang.System. | getProperties() must check for overwritten properties on a per sandbox basis (user.dir ..) |
| java.lang.Class | forName() must be intercepted to detect the instantiation of interposed classes. |
| java.net.URL | Intercept openConnection() |
| java.net.URLConnection | Cache URL content |

**Table 3: Affected classes and APIs**

## 4 Approach

The problems we have identified require modifications to the functionality of standard system APIs. The most straightforward solution would be to replace the standard system packages with new implementations. However, this would be tedious because each affected class must be fully re-implemented. In addition, it may be impossible to do this anyway if some native code is required.

Because the JVM and the original application source code cannot be modified, we propose modifying API functionality by using *API Interposition* to intercept standard API invocations. This is accomplished by extending the APIs of the standard system classes by adding a distinct prefix to the class hierarchy namespace. For example, an intercepted version of `java.io.File` might be named `interposed.java.io.File`. Then its constructor cannot be magically invoked by the application code.

This approach is implemented by using the technique of “bytecode editing” to intercept calls to constructors and instantiate an object of the extended class. The extended class contains methods that supersede the initial ones and perform appropriate actions on the interposed object. For example, for the `java.io.FileInput-`

Stream class, the constructor is changed to instantiate a FileInputStream object in the cache directory, and its content is initialized from the remote storage server. The application then interacts with this local cache FileInputStream object. The close method could ensure that the file content is flushed to the storage server. Some methods of a class may not need to be redefined in the extension.

Final classes and abstract classes cannot be intercepted this way. By definition, it is impossible to extend final classes. Abstract classes can be extended, but because they cannot be instantiated, there are no constructor invocations. Consequently, for abstract classes, we intercept and override invocations of the specific methods whose behavior we want to modify.

In the following sections. we provide Java source code examples to illustrate how legacy application classes need to be transformed. In a real system, application source code is not required, and class files are modified as described in Section 5.

### 4.1 Interposing classes.

In order for instances of class A to be replaced by instances of class interposed.A, new statements and constructor invocations for class A must be replaced by new statements and constructor invocations for class interposed.A, respectively.

The following SimpleFile class

```
import java.io.File;

public class SimpleFile {
    static public File file;

    public File add() {
        return new File("add");
    }

    public static void main(String argv[]) {
        file = new File("aFile");
        String name = file.getName();
        file = new SimpleFile().add();
    }
}
```

when edited to interpose class java.io.File is transformed into

```
import interpose.java.io.File;

public class SimpleFile {
    static public java.io.File file;

    public java.io.File add() {
        return new File("add");
    }

    public static void main(String argv[]) {
        file = new File("aFile");
        String name = file.getName();
        file = new SimpleFile().add();
    }
}
```

Changes are underlined. The public fields declaration (File file) and method signatures (File add) are not modified because they are exported outside of the class. Because the newly created object is an extension, no other statement has to be modified.

The class extension would contain statements such as:

```
package interpose.java.io;

public class File extends java.io.File {

    public File(String name)
    throws NullPointerException{
        // .. our own initialization
    }

    public long lastModified() {
        // ...
    }

    // other modified APIs ...
}
```

Any number of APIs from this class can be overwritten in the extension.

### 4.2 Interposing Methods

Two kinds of classes cannot be interposed as described above.

- The class is *final,* and by definition cannot be extended (e.g., java.net.URL, java.lang.System, java.lang.Class)
- The class is *abstract*, and hence is never instantiated by the application but rather by platform-specific routines (e.g., java.awt.Toolkit, java.net.URLConnection)

An object from these types of classes cannot be instantiated from a class extension. Consequently, instead of replacing new and <init> method invocations, a call to a class method that needs to be altered must be replaced with a call to a static method of a newly created abstract class.

### 4.3 Abstract Classes

An obvious example of this approach is substituting the getProperty method from the java.lang.System abstract class. In SoD, this method needs to be modified so that it will return the private property value from the sandbox. The following GetMyProperty class

```
public abstract class GetMyProperty {
  public static void main(String argv[]) {
    String s;
    s = System.getProperty("foo");
  }
}
```

would be modified as follows to interpose the System.getProperty method

```
public abstract class GetMyProperty {
  public static void main(String argv[]) {
    String s;
    s = interposed.System.getProperty("foo");
  }
}
```

and the new abstract file could be

```
package interpose.java.lang;

public abstract class System {
  public static String getProperty(String s) {
    // ... any specific code to fetch
    // private property value from sandbox
    return (String) ...;
  }
}
```

## 4.4 Final Classes

If a class is final but not abstract, the reference to the object on which the initial method was called cannot be discarded. This is handled by passing the reference as an argument to the interposed static method. Here is an example for the final java.net.URL class.

```
import java.net.URL;

public class SimpleURL {
  public static void main(String argv[]) {
    try {
      URL url = new URL("foo");
      InputStream is;
      is = url.openStream();
    } catch (Exception e) {
    }
  }
}
```

The modified code would be

```
import java.net.URL;

public class SimpleURL {
  public static void main(String argv[]) {
    try {
      URL url = new URL("foo");
      InputStream is;
      is = interposed.java.net.URL.openStream(
           url);
    } catch (Exception e) {
    }
  }
}
```

The new abstract class would be similar to

```
package interpose.java.net;

public abstract class URL {
  public static InputStream
                openStream(java.net.URL url)
    throws IOException {
    // ... specific code
      return (InputStream) ...
  }
}
```

Note that the substituted method may be a constructor (<init>) as it would be for the URL(java.net.URL) constructor. While the replacing method initializes an instance of the extended class when regular class interposition is used, here it must return (versus initialize) an instance of the initial class. So the following class

```
import java.net.URL;

public class SimpleURL {
  public static void main(String argv[]) {
    try {
      URL url;
      url = new URL("foo");
    } catch (Exception e) {
    }
  }
}
```

would be replaced by

```
import java.net.URL;

public class SimpleURL {
  public static void main(String argv[]) {
    try {
      URL url;
      url = interposed.java.net.URL.URL("foo");
    } catch (Exception e) {
    }
  }
}
```

and the abstract class could be

```
package interpose.java.net;

public abstract class URL {
  public static java.net.URL URL(String s) {
    throws MalformedURLException {
      // ... any specific code ...
      return (java.net.URL) ...
    }
}
```

## 4.5 Reflection Support

Classes can be instantiated indirectly through the java.lang.reflect package to allow the application to control and manipulate Java code. Since this API provides abstract control of Java code, replacing method invocations is insufficient. We have chosen to interpose with access to the java.lang.Class itself and intercept the forName() call to detect lookups of interposed classes and substitute the class name. We understand that this approach may not cover all cases (e.g., JavaBeans) because an object can be instantiated without using its full interface name. An alternative would be to interpose on access to the newInstance APIs from the java.lang.Class and java.lang.reflect.Constructor classes.

Similarly, the invocation of a method that is interposed (versus a class) may be performed through reflection. This case was not handled in our prototype, but similarly the java.lang.reflect.Method class would require interposition to detect *invoke* method invocations.

# 5 Implementation

SoD is written entirely in pure Java. It has been used to perform experiments on PDAs and workstations under the Windows NT, Windows CE, and Linux operating systems. SoD has been executed with a variety of JVMs, such as Sun's JDK, Sun's Personal Java (PJava), and HP's Chai. Each is a different implementation of the Java specification, either version 1.1 through 1.3 (JDK) or just version 1.1 (PJava and Chai). The exact same runtime library runs on the various platforms.

The interposition mechanism is implemented as an independent module, and is also written in pure Java so it can be reused outside of SoD. Our implementation defines a classloader, with an instance per application,

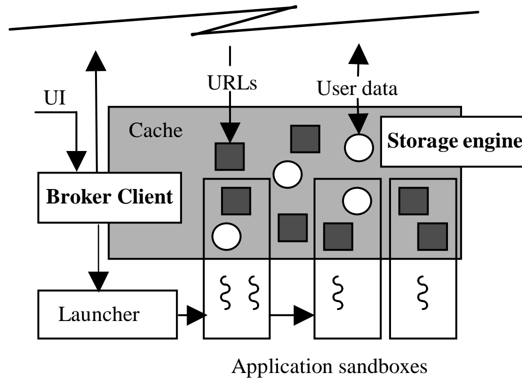


**Figure 2: Client runtime framework**

to allow us to gain control when code is loaded into the virtual machine. It is given a list of class and method names to interpose, and the corresponding names to use to extend classes and redefine methods. The interposition occurs dynamically when a class is loaded. The modified classes may be cached on the user's device to avoid re-applying the same changes when it is next used. The modified class byte array is then passed to the virtual machine. The overall implementation architecture can be see in Figure 2.

**The class file format**. A class file is typically composed of a constant pool and a collection of descriptors.

- **Constant pool.** Four of the eleven constant types are of interest here: constant strings (utf8), class names, method references, and method signatures. All of these constants are referenced in the bytecode by indexes into the constant pool. There are no duplicate entries. If a user-defined string and a method have identical names, they will refer to the same constant.
- **Descriptors.** There are descriptors for fields, interfaces, methods, and other class attributes, such as exception tables and inner classes. Field, interface, and method descriptors include an index into the constant pool for the component's name plus information related to the component itself. Examples of this information include a method's bytecode, and a field's access flags, type, and initial value.

Classes can be loaded much more quickly if the classes that need interposition can be detected without having to parse all of the bytecodes. To achieve this goal, we identify the APIs that require interposition by examining the class names and method references in the Java constant pool.

## 5.1 Class Interposition

Once we identify a class that needs to be interposed, we have to detect whether it will be instantiated. We assume that if the class contains a constructor (named `<init>`) then objects of that class will be created. We can interpose these constructors by parsing the method references and redirecting them, and in this way avoid having to parse all of the bytecode.

If such a constructor reference exists, then we must also handle invocations to the class throughout the bytecode. In addition, because a constructor invocation is always preceded by a `new` statement with a class reference as an operand, we must also handle the class reference.

Surprisingly, these tasks can be performed with only very small modifications to the bytecode. They are accomplished by renaming the initial class name to the interposed class name in the constant pool. This change will automatically cause the `new` statements and `<init>` invocations to reference the proper interposed class.

However, other kinds of references to the initial class are wrong, such as those for the class's fields and methods. These references are handled by making a copy of the initial class name and entering it as a new constant in the constant pool. We then modify these references so that they point to this new constant, which contains the initial class name.

## 5.2 Method Interposition

Method interposition for final and abstract classes is more complicated because a non-static method invocation (i.e., `invokevirtual`) must be converted to a static method invocation. This can only be accomplished by actually parsing and modifying the bytecode.

First, similar to the way that class interposition is handled, the constant pool is parsed to detect candidate methods and obtain their indexes. A new constant pool entry is created for the replacing method. Then the bytecode is scanned to retrieve the invocations. If the method is static then we must change the reference to a non-static invocation and the operand of the method's reference index must be changed. Static invocations place the object reference as the first item on the stack. We can leave this on the stack, treating it as the first argument of the non-static invocation, and thereby limiting the changes required.

The advantage of this technique is that the bytecode size is unchanged and no bytecode translation is required. One exception is replacing a constructor invocation of a final class with a static method, which causes the `new` and `dup` bytecodes to be removed. We replaced them with `nops` bytecodes to maintain the length.

## 6 Experiments

First, we investigated the code size increase as a result of interposition, using the six applications described in Section 5. Table 4 shows that the code footprint is increased by 0.3% on average. In the worst case we only get an 8% increase due to the tiny, single class application used. Note, these numbers do not consider the size of the new extended classes and newly created static classes that are part of the runtime, which is only a fixed cost. Most of the increase is because of the class file renaming. In our prototype, each interposed class' name was prefixed with additional twenty characters.

| | Increase (bytes) | Total (bytes) | % |
|---|---|---|---|
| Average | 7004 | 1930436 | **0.3** |
| Worst case | 379 | 4680 | 8 |

**Table 4: Class file size increase**

Second, we analyzed how the class loading time was affected (see Table 5). The overhead increases close to linearly with the byte code size and the increase reaches 76% for a 1 MByte application. However, this overhead is incurred once at first load time, and does not affect the application's runtime. One cause of this runtime overhead is our use of the `DataOutputStream` write API for nearly every byte of the class file This can undoubtedly be improved. This significant overhead favors our use of a cache for the modified classes.

| Application | Size KB | Class count | | Load time (seconds) | | |
|---|---|---|---|---|---|---|
| | | Total | Changed | Initial | overhead | % |
| crossword | **26** | 7 | 2 | 0.57 | **0.11** | **19** |
| PIM | **51** | 29 | 2 | 0.91 | **0.36** | **39** |
| ftp | **142** | 22 | 4 | 2.00 | **0.66** | **33** |
| Browser | **180** | 51 | 8 | 2.88 | **0.81** | **28** |
| mail client | **366** | 138 | 23 | 3.05 | **1.95** | **64** |
| Imaging | **1145** | 129 | 21 | 4.23 | **3.23** | **76** |

**Table 5: Class load time overhead for interposition**

Next, we wanted to understand how much of an application was modified by the interposition we perform. For our investigation, we collecting interposition statistics from six applications (PIM, Ftp client, web browser, image manipulation, crossword, and a mail client). Our results show that 18% of the classes were changed due to interposition (Table 6).

| | Modified | Total | % |
|---|---|---|---|
| Number of classes | 71 | 383 | **18** |
| Size (bytes) | 589552 | 1930436 | **30** |

**Table 6: Modified classes**

| Operation | JDK1.3 | | Pjava | |
|---|---|---|---|---|
| | µs | % | µs | % |
| `new Random()` | 0.01 | 2 | 1.91 | 23 |
| `Random.nextLong()` | 0.02 | 9 | 0.41 | 11 |
| `Random.nextInt()` | 0 | 0 | 0 | 0 |

**Table 7: Class interposition overhead**

The class file size for the additional interposed classes and methods is around 30 KB (based on the APIs in Section 3.3). The interposition system is 90 KB including a non-optimized, general-purpose class file editing package of around 80 KB.

### 6.1 Micro-Benchmarks

Given the large number of interpositions, we next investigated the performance of actual single interpositionings. We performed three micro-benchmarks to measure the cost of interposed classes in terms of runtime overhead. First, we measured the interposition mechanism cost of null invocations using class interposition, and second we measured the same for method interposition. Third, we measured the overall cost of interposed file access for simple operations. Measurements were performed with both JDK version 1.3 (JIT enabled) and Pjava on the same PIII 600Mhz workstation.

For class and method interposition we interposed *null* classes and methods, using constructors and methods that invoke their super classes. Thus, we only measure the cost of going through an extra level of indirection.

**Class Interposition overhead.** We interposed the `java.util.Random` class with the following one:

```
package interpose.java.util;

public class Random extends java.util.Random {
    public Random() {
        super();
    }
    public long nextLong() {
        return super.nextLong();
    }
}
```

The results in Table 7 indicate that the overhead is less than 0.2µs with JDK and less than 2µs for Personal Java.

**Method interposition overhead**. We interposed two methods for the abstract `java.lang.System` class with the following code:

```
package interpose.java.lang;

import java.util.*;

public abstract class System {
 public static Properties getProperties() {
   return java.lang.System.getProperties();
 }
 public static int identityHashCode(Object o) {
   return java.lang.System.identityHashCode(o);
 }
}
```

The overhead is in the same range, less than 0.01µs for JDK and 1µs for Pjava (see Table 8).

| java.lang.System Operation | JDK1.3 | | Pjava | |
|---|---|---|---|---|
| | µs | % | µs | % |
| identityHashCode(x) | 0.01 | 5 | 0.93 | 67 |
| getProperties() | 0.01 | 12 | 0.21 | 18 |

**Table 8: Method interposition overhead**

For both class and method interposition measurements (see Tables 7 and 8), the initial constructor and method executions are relatively quick and explain why overheads vary from 9 to 23%. The `identityHashCode` relative overhead is itself very large (67%) because this method is native and the extension is pure Java.

**File access benchmark.** We performed more user-sensitive measurements related to file storage in the context of SoD. The interposed classes are `java.io.File` and `java.io.File.InputStream`. Typically the extensions check if the file is remote and if so either creates a local cache copy or refreshes it if outdated. The file is read through the `BufferedReader.readLine()` method. Table 9 shows that our overhead is modest, between 0.7 and 2.5% non-optimized. Given the increased functionality, we believe this is acceptable.

### 6.2 Concurrency

We ran the same experiments as those described in Section 3 but within a single virtual machine running SoD. Table 10 shows that the memory overhead reduction when starting an additional task is significant, varying from savings of 93 to 97%. The increase for the first instance is due to the memory usage by the SoD runtime, but this overhead is only incurred once. For most platforms this overhead is acceptable for even an unoptimized implementation of our extensions.

Next, we examined the startup time overhead per task of using the SoD infrastructure. SoD caches application namespaces in order to avoid re-interposing and reloading classes remotely. Measurements were performed with and without this cache. Pjava's timer API precision is 1 second and affects the accuracy of the comparison.

| Location | JDK1.3 | | Pjava | |
|---|---|---|---|---|
| | ms | % | ms | % |
| Local to host | 0.3 | 1.9 | 4.1 | 0.7 |
| Remote (cached) | 0.4 | 2.5 | 6.1 | 1.1 |

**Table 9: Interposition overhead when applied to reading an 80KB text file.**

| Number of Instances | Jornada 720 StrongArm | | Jornada 548 SH3 | X86 PIII 600 Mhz |
|---|---|---|---|---|
| | Pjava | ChaiVM | Pjava | JDK1.3 |
| Baseline 1st | 3452 | 3498 | 2398 | 9592 |
| 1st | 4164<br>+ 20% | 5964<br>+ 70% | 3172<br>+ 32% | 10200<br>+ 6% |
| Baseline 2nd | 1524 | 2004 | 2398 | 9328 |
| 2nd | 100<br>- 93% | 100<br>- 95% | 80<br>-97% | 480<br>- 95% |

**Table 10: Memory usage per Java task (KB)**

Table 11 shows that the startup times or their improvements are not identical for the four platforms. When the namespace is cached, the improvement is significant as well, varying from 89% to 95%. ChaiVM's figures in Tables 10 and 11 are a little disturbing for both footprint and uncached startup times. We believe that startup time may be accounted for by the lack of a JIT compiler. Also, we believe the large footprint may stem from an interaction between SoD and Chai's use of a bytecode compressor for embedded applications. When running without SoD, most of the startup time is due to the internals of the VM and we suspect most of it is native code. On the contrary, SoD is entirely written in pure Java and its runtime execution overhead appears relatively large.

## 7 Lessons Learned

**Adaptability using interposition.** Any standard Java API may be transparently interposed to either extend or replace it. Neither the legacy software nor the virtual machine and standard Java libraries need to be modified. We demonstrated this for four different JVMs.

**Sandboxing and interposition enable true concurrency.** Using interposition, it is possible to effectively run multiple applications within one virtual machine. This is achieved by transparently avoiding namespace collisions, sensitive API interference, system property collisions and resource access security conflicts. Doing so reduces the incremental virtual machine memory cost per application from between 44-100% to 1.6-4.7% depending on the platform used (see Section 3).

| Java Task | Jornada 720 StrongArm | | Jornada 548 SH3 | X86 PIII 600 Mhz |
|---|---|---|---|---|
| | Pjava | ChaiVM | Pjava | JDK1.3 |
| Baseline | 2000 | 4000 | 3000 | 200 |
| Uncached | 2000<br>(- 0%) | 7056<br>(+ 76%) | 1000<br>(-200%) | 120<br>(- 40%) |
| Cached | 1000<br>(- 100%) | 423<br>(- 89%) | 1000<br>(-200%) | 10<br>(- 95%) |

**Table 11: Elapsed time per Java task (milliseconds)**

**Transparent remote persistent storage.** Java's IO package can be transparently mapped into remote persistent storage (see Section 4). By using interface interposition, we were able to achieve this without modifying the application and with reasonable overhead.

**Transparent caching for speed.** A cache mapping can effectively reduce Java application load time and the overhead of bytecode manipulation for interposition. Results indicate that load-time performance is increased 12-16 fold even while using our framework.

**Transparent caching for disconnection.** A cache can effectively allow certain types of applications to continue operating while disconnected. Apart from the benefit of class and data loading while disconnected, simple reconciliation policies can allow effective re-connected migration of data back to the storage server.

**Load-time interposition can be slow.** Even though our implementation is far from optimal, the number of points at which interposition is required for our Java enhance will clearly affect load-time performance. Our use of caching throughout the system mitigates a lot of this overhead, however, minimizing the pervasiveness of interpositions could still be improved.

**Variety of Java Platforms** — Even though the Java platform and specification are fairly comprehensive, subtle variations in their semantics can complicate support for all JVMs. Also implementation variations affect the size and nature of the benefits of our enhancements. For example, the incremental footprint per application between JVM implementations varies noticeably.

**API Interposition Completeness —** In this paper we used interposition to apply new semantics to the APIs that were affected by the requirements introduced at the beginning of the paper. However, finding the complete set of affected APIs for an arbitrary Java platform is complex. Without detailed insight into the APIs, it does not seem possible to automatically determine which system APIs need to be interposed.

## 8 Related Work

The idea of interposition is not unique to our work. JDK version 1.3 introduced **proxies** for reflection, that can also be used for interposition [26, 13]. However, fully transparent interposition is not supported in any JVM, since all interposed methods need to be defined as interfaces and the JVM must conform to version 1.3. **BCA** [17] and the **JavaClass API** [7] are somewhat more sophisticated bytecode editing frameworks that could potentially replace our class file editing module. BCA is more general than our approach, since the bytecode changes can be arbitrary. Also, we do not alter class or method signatures. Instead of interposition, **BIT** provides bytecode editing techniques for instrumentation purpose [19]. Our approach differs by not requiring any specific virtual machine or modifications. Our work also differs by only using API interposition to support pervasive services with the Java platform, rather than a complete bytecode editing framework.

The installation of Java applications is less than straightforward. This has lead to the creation of several application installation wrappers and frameworks, including **Sun's Java Web Start** [15] and **ObjectBox's JBee** [14, 16]. These systems focus on installation automation and caching of Java archives and applets, rather than considering Java in pervasive, mobile devices. As such they are a more pragmatic approach to the problem of seamless execution of Java services.

Several projects have investigated mobile data access, most notably **Bayou** [8] and **CODA** [25]. Our work complements this work and is more oriented towards enabling Java-based devices and legacy applications to benefit from such distributed systems without any change to the operating system and the application.

Several recent standards have addressed service publication and access, including Universal Description, Discovery, and Integration (**UDDI**) [2] and Open Services Gateway Initiative (**OSGi**) [23]. Each standard views service delivery from a different perspective, which could be very beneficial to our work by increasing the breadth of our service support.

Viewing applications as services has also become prominent of late. Microsoft's **.NET** and **.NET My Services** enable users to access Web services on the Internet from various devices [12]. Using XML [6], UDDI [2], SOAP [5], and WSDL [27], and a Common Language Runtime (CLR), .NET presents a new application framework designed for user-focused Web services.

The work in this paper is conducted within the scope of an umbrella project called Pervasive Services Infrastructure (**PSI**) [21]. SoD complements another PSI project, Adaptive Offloaded Services (**AOS**) [20], which addresses partitioning of Java applications and offloading partitions to surrogate servers. SoD and AOS contribute to an overall vision: "Any service to any device".

Several projects are investigating the area of pervasive services, including **Oxygen** [9]**, Portolano** [10]**, Aura** [22] and **CoolTown** [18]. These projects take a distributed computing approach where composable services are distributed at locations throughout the environment. We instead view devices and infrastructure as temporary service caches of computation and storage that ultimately originate from back-end service providers.

## 9 Summary and future work

In this paper, we have presented our investigation into the use of API interposition in Java to enhance the platform. We showed the ability of interposition to add support for enhanced multiple application sandboxing, transparent remote storage, and service caching all without modifying either the virtual machine or the application implementation. From our results, we believe that this approach can effectively support these enhanced features with reasonable overhead (1.9% for file access). In fact, with service caching, load times can be radically decreased with 12-16 fold reductions.

As part of this ongoing work we would like to further investigate the APIs that need to supported to achieve our goal of pervasive service supporting using the existing Java platform. As part of this work, we plan on investigating high Java specification versions and as well as a set of guidelines for determining APIs that don't fit our requirements. Finally, we would like to investigate the performance of our platform further as well as consider a wide set of benchmark applications.

## Acknowledgements

We are indebted to Kimberly Keeton, David Lie, Dan Muntz, Todd Poynor, and Greg Snider for reviewing the paper. Their comments significantly improved its content and presentation.

## References

[1] IBM. alphaBeans - FTP Bean Suite Project - http://oss.software.ibm.com/developerworks/opensource/ftp.

[2] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S. 2000. UDDI Technical White Paper, available at www.uddi.org, September 2000. 2001.

[3] Back, G., Hsieh, W. and Lepreau, J. "Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java". Proc 4th USENIX OSDI, Oct 2000 pp 334-346.

[4] Bernadat, P., Lambright, D. and Travostino, F. "Towards a Resource-safe Java for Service Guarantees in Uncooperative Environments". Proc. IEEE PLRTIA Dec. 1998.

[5] Box, D., et al,. Simple Object Access Protocol (SOAP) 1.1, W3C. http://www.w3.org/TR/SOAP.

[6] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E. (Editors). 2000. Extensible Markup Language (XML) 1.0 W3C Recommendation, 6 October 2000.

[7] Dahm, M. "Byte Code Engineering". Java-Information-Tage 1999 (JIT'99), Sept. 1999.

[8] Demers, A. et al. "The Bayou Architecture: Support for Data Sharing among Mobile Users. Workshop on Mobile Computing Systems and Applications. IEEE, Dec. 1994.

[9] Dertouzos, M.L. "The future of computing," Scientific American, July 1999, pp 52-55.

[10] Esler, M., et al. "Next century challenges: data-centric networking for invisible computing: the Portolano project at the UW" Proc of 5th ACM/IEEE Conf. on Mobile Computing and Netw, Aug, 1999, Seattle, WA pp 256-62.

[11] Gong, L. "JavaTM 2 Platform Security Architecture". October 2, 1998. http://java.sun.com/j2se/1.3/docs/guide/security/spec/security-spec.doc.html.

[12] Hall Gailey, J. 2001. Introducing .NET My Services. MSDN Library note, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/myservintro.asp. September 2001.

[13] Harpin, T. Using class java.lang.reflect.Proxy to Interpose on Java Class Methods. SUN's developer technical articles, July 2001. http://developer.java.sun.com/developer/technicalArticles/JavaLP/Interposing.

[14] INSISO ObjectBox. http://www.objectbox.com/insiso_whitepaper.pdf.

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

[16] JBee. http://www.objectbox.com/javabee.

[17] Keller, R. and Holzle, U. "Binary Component Adaptation". ECOOP'98 Conference Proceedings, pp 307-329.

[18] Kindberg, T., et al., "People, Places, Things: Web Presence for the Real World,“ Proc. 3rd WMCSA, 2000.

[19] Lee, H.,B. and Zorn, B., G. "BIT: A Tool for Instrumenting Java Bytecodes". USENIX USITS, 1997, pp 73-82.

[20] Messer et al., "Towards a Distributed Platform for Resource-Constrained Devices", submitted for publication, available as HPL Technical Report.

[21] Milojicic, D. et al. "Ψ Pervasive Services Infrastructure." Technologies for E-Services, Proc 2nd Int'l Workshop, TES 2001, Rome, Italy, September 2001, pp 187-200.

[22] Noble, B.D., et al, "Agile Application-Aware Adaptation for Mobility." Proc 16 SOSP, October 1997, pp276-287.

[23] OSGI Service Gateway Specification, available at www.osgi.org.

[24] Secure Computing with Java: Now and the Future. Sun JavaOne conference, 1997.

[25] Satyanarayanan, M., "Scalable, Secure, and Highly Available Distributed File Access," IEEE Computer, May 1990, Vol. 23, No. 5, pp 9-21.

[26] Sun Microsystems. Dynamic Proxy classes. JDK1.3 documentation guide. http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html.

[27] W3C, Web Services Description Language (WSDL) 1.1, Note 15 March 2001. http://www.w3.org/TR/wsdl.