Ghost Machine

A Distributed Virtual Machine Architecture for Mobile Java Applications

Sean McDirmid, University of Washington
mcdirmid@cs.washington.edu

The Ghost Machine platform lets you run a limited subset of Java on mobile computing devices. Because it supports a true Java subset, the same programs can also be compiled and run on any fully-implemented Java platform. The Ghost Machine platform currently supports PalmOS devices such as the PalmPilot. It has been designed with special consideration given to memory footprint and efficiency.

Mobile computing is increasing in popularity at a rapid pace. The PalmPilot organizer is almost ubiquitous in the business community. Microsoft's Windows CE handheld and Palm-size PCs are becoming more popular and are being marketed heavily. Smaller devices such as the REX and smart cards are increasing growing in importance and use.

Mobile devices are popular not because they attempt to replace the traditional desktop computer, but because they extend the reach of the desktop computer beyond the desktop. They let people view, add and update information on their desktop computers. Using synchronization, mobile devices can maintain data consistency with desktop PCs in a drop-and-go manner – the user need only drop a device onto a PC (using a cradle or cable), synchronize and then go about his or her business.

Of all the mobile computers currently on the market, PalmOS-based devices stand out. They have gained widespread acceptance among both consumer and business users. The PalmOS was designed specifically for PDAs. It is small, efficient and simple, and most importantly, it easily synchronizes with desktop computers, making it an ideal mobile computing platform, despite its constraints. The PalmPilot, for instance, has only one megabyte of memory for both data and applications and a modest 17 MHz 32-bit processor.

Development Obstacles

Mobile computing platforms are becoming more open to third-party application development. Despite many opportunities, however, third party development is proceeding slowly. This is changing with platforms such as Windows CE and PalmOS, yet there are still no known killer third-party applications.

Development for these devices is difficult and different for conventional developers. In most cases the programming languages (C and C++) are the same but the APIs can be very different. The PalmOS Application Programming Interfaces (APIs) are nonstandard; they have are no desktop equivalent which imposes a substantial learning curve on desktop developers. Windows CE tries to remedy this problem by using modified Win32 APIs. They are, however, significantly different from their Win32 counterparts and the Win32 APIs are complex and difficult to learn.

Effective application distribution also presents a major obstacle. Unlike the desktop, which has a few platforms with large installed user bases, mobile devices are more specialized. In order for an application to be economical, it may have to target multiple platforms. Even among Windows CE handhelds, devices with different versions of the OS or different processor architectures require different binaries. As a result, the developers have to target and test each device individually.

The Java Application Environment

Java has the potential to solve these problems. The Java Application Environment (JAE) has a virtual machine that executes machine-independent bytecodes and a standard set of core classes. Java was originally conceived for mobile and embedded devices. However, it's mostly known for its desktop use. With the advent of the World Wide Web, Java's portability and safety have made it the platform of the Internet.

Java programs compile into one or many class files that contain machine-independent bytecodes. The class file format is defined not only to ensure portability but also to allow most type checking to be done statically. This drastically reduces the amount of runtime checking needed to run Java programs, allowing safe execution with reasonable performance. With safety, it is possible to build and enforce runtime security policies.

Security and safety are perhaps more important on mobile devices than on desktop workstations. Software failure on a mobile device can lock up the entire operating system. Safety makes crashes due to buggy application code nearly nonexistent. Security ensures that sensitive information stored on a mobile device is not read or modified by unauthorized applications. It is almost impossible for an end user to ensure these qualities without substantial and costly hardware assistance, using code that is compiled and distributed in a native-executable format.

Application programming interface portability is at least as important as binary portability. It provides convenient and universal abstractions for input/output, utilities and various system resources such as the file system and the clock. Perhaps one of the most important class libraries include in the JAE is the Abstract Window Toolkit (AWT). The AWT lets programs implement portable and consistent user interfaces across various platforms. An implementation of the JAE is not complete without an AWT.

PersonalJava

The PersonalJava specification, designed for network-enabled applications on consumer, home and mobile devices, is a large subset of the desktop Java specification. It can run most applets written for the desktop. The PersonalJava Application Environment, a JAE that implements the PersonalJava specification, requires about two megabytes of read-only memory and between one half and one megabyte of RAM. It also requires a 50 MHz 32-bit CPU. PersonalJava is the Java subset that needs to be supported on mobile devices such as the PalmPilot.

Due to these PersonalJava resource requirements, there is no successful PersonalJava port that runs on any PalmOS device – the PalmPilot has neither the memory nor CPU speed. The PersonalJava API may be too large, but the real problem with porting a Java application environment to the PalmPilot is the monolithic structure of the virtual machine. If these mobile devices are meant to be extensions of the user's desktop machine, why not make the desktop machine an extension of the mobile device?

The Ghost Machine moves up-front virtual machine services from the client device running the Java application to a desktop server workstation. These services are used only once on a class file at the beginning of its execution. The Ghost Machine is designed to minimize both its installation and its runtime footprint. Compared to the PersonalJava Application Environment, the Ghost Machine is targeting a requirement of 256 KB of read-only memory and 128 KB of RAM.

The Ghost Machine Architecture

The Ghost Machine architecture is designed for efficiency, minimized memory footprint, portability and simplicity. To do this, the Ghost Machine architecture contains distributed and layered components.

Perhaps the most innovative feature of the Ghost Machine is its distributed design. Since a Java class is verified and processed for running only once, these are ideal services to move off the client. Since these services are only executed up-front, the client device need only be connected to the server when it is installing a new application, not when the application is executing.

The conventional Java virtual machine architecture is monolithic. All services are performed within one process on one machine. The class file enters the virtual machine (perhaps from the Internet) and executes as shown in Figure 1.

The Ghost Machine platform, however, can move very resource-intensive up-front services onto a server as shown in Figure 2. Since these services are no longer present on the client, the client virtual machine is smaller and need only focus on the class execution.

The Server

The Ghost Machine server is responsible for the up-front processing of Java classes and delivering the processed classes to the client. It reduces the installed memory footprint of the each class while putting the class into a runtime form that requires the client to do little extra work to complete the class loading.

The server reduces the memory footprint of a Java class by transforming class type information normally stored as strings into an integral data type. Type information is used to identify method signatures like `void foo(int x)`, field signatures like `public String bar` and classes like `java.lang.Object` during verification and linking. Storing this information as strings keeps Java classes loosely connected: a change in one Java class does not necessarily require recompiling another class that depends on it. However, after linking, it's safe to discard the strings as long as types can be consistently identified across all installed Java classes.

To do type transformation the Ghost Machine server maps the string representation to unique and consistent integral data types. This map is maintained persistently across class installations; the server must remember what types are already assigned identifiers and the identifier values. If a type is encountered in another class file the same identifier must be used. If a type is encountered for the first time, an integral identifier that is not currently assigned is assigned to the new type.

The constant pool contains entries for all constants needed for execution, including strings, constant integral data types and instruction arguments. Each entry in the constant pool is tagged so that during verification the verifier can ensure that all instructions access the constant pool properly. After verification the tags can be eliminated. Also, the constant pool must be reorganized to prepare a class for execution.

These installation optimizations reduce class files in size by about half in the common case, though reduction varies from class to class.

A Java class needs to be prepared only once for execution. This is done at the server during the class installation. In order for execution to be efficient, each frequently-used structure in a class needs to be immediately accessible. Variable-length structures need to be eliminated or extra data must be added to the Java class with any offsets that are not automatically known. The constant pool is variable length, but most entries are easily transformed into an array of constant-length structures (the only entries that cannot be made constant length are strings). The other inherently variable-length structure in a Java class is the array of method bytecodes. Extra offset tables are added to the processed class to accommodate these structures.

Another important issue in preparing a Java class for execution is the byte ordering and alignment of integral data types and integral data type composites. It is important to ensure that the alignment is compatible with the target platform. Also, byte ordering should be swapped if necessary. For the PalmOS, each integral type must be aligned on a boundary divisible by its size. The PalmOS is little endian so byte swapping is not necessary.

Finally, after the Java class is processed the server is responsible for delivering it to the client. This can be done with a constant network connection, the server can inform a desktop synchronization manager to install the class on its behalf, or the server might even relay the class directly to the client using the serial port.

Since type information is transformed from its string format into identifiers, virtual machine features that require runtime type information in string format are difficult to support. This mainly applies to reflection, the process of examining the attributes of a class in human-readable format. Reflection is a part of the PersonalJava specification and can be supported by sacrificing memory, though the strings needed to support reflection can be shared among all the installed classes on the client and can be compressed.

Sometimes client native code needs to access class resources by their specific type signature. Unfortunately, client native code is usually compiled before type strings are assigned their integral identifiers. As a consequence, neither type-specific strings nor integral identifiers can be embedded in native code. The Ghost Machine solves this problem by adding a level of indirection to the server's resource access mechanism. When the native code is written, the programmer declares the type signatures in an intermediate type list. The programmer then uses the offsets into the intermediate type list to refer to the type signatures. After the signatures are assigned integral types, the server uses the intermediate type list to generate an intermediate type database. At runtime, this database is used to resolve the offsets used by the programmer to the integral identifiers of the types they represent.

The Client

Since the server handles most of the class loading responsibilities, the client can be quite small. It is still responsible for important services such as bytecode interpretation and garbage collection. Since the client services are limited by the client's computing resources, extra care must be taken in coding these services efficiently. Portability and maintainability were also strongly considered when the client was designed.

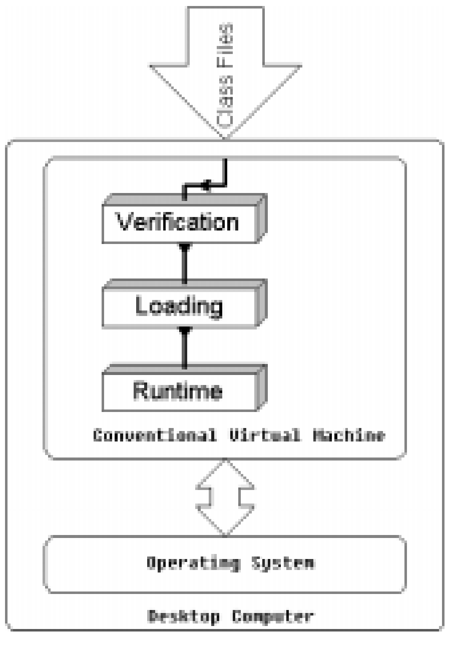


Figure 1 - Conventional Java virtual machine architecture.

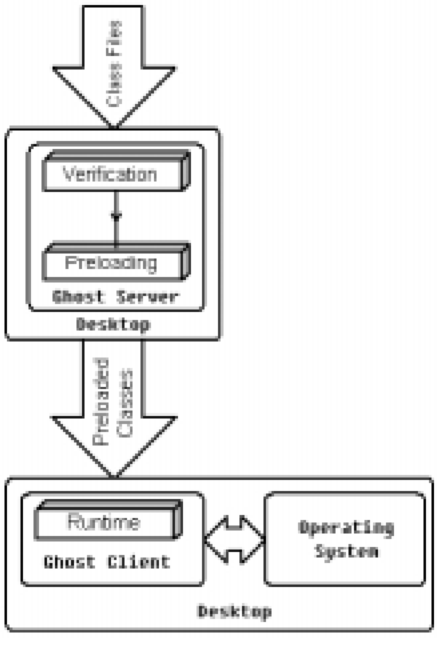


Figure 2 - The Ghost Machine architecture.

The client software is divided into three layers (see Figure 3). These layers are tightly coupled during runtime to ensure efficiency. They are conceptually important only at the source-code level to enable good portability characteristics.

The Platform Abstraction Layer

The Platform Abstraction Layer (PAL) factors out platform-dependent services needed by the core virtual machine. The PAL provides only the minimum necessary to run the virtual machine. Platform-specific functionality added at the third level layers extends the virtual machine to provide new services.

The PAL is responsible for providing the core virtual machine with heap-management methods. The heap is a pool of memory borrowed from the operating system – platform-dependant code is needed to adjust the size of the heap, create new heaps and give unused heaps back to the operating system. The PAL is also responsible for providing the virtual machine a way to access the classes installed by the server. Finally, the PAL contains the routines needed to boot the virtual machine and enter the virtual machine event loop.

Portable Core Services

The second layer of the client software provides the core virtual machine services that are C source code portable across multiple platforms. The services implemented by this layer are the minimum needed to run Java bytecodes and provide an interface to native methods.

Automatic memory management is an important aspect of the virtual machine and a core virtual machine service. Data, when allocated by a Java bytecode, is not reclaimed until it can be proven that it is not in use. To determine this, the garbage collector sweeps through the runtime structures for objects that are still alive. Those objects that are not in the alive set must be dead (though the converse of this statement is not always true).

The Ghost Machine currently implements a mark-and-sweep algorithm with compaction. When memory allocation fails because the object heap is exhausted, the garbage collector is invoked. The mark-and-sweep algorithm first determines what data is still alive and then all live data is compacted to the bottom of the object heap. Not only does this prevent memory fragmentation, it also facilitates quick object allocation by merely incrementing a top pointer.

The core virtual machine implements an interpreter to execute Java bytecodes processed by the server. The interpreter loop is a giant switch statement that mostly executes instructions according to the Java Virtual Machine Specification and lets Java bytecodes interface with the virtual machine. In addition, an important virtual machine service is interfacing bytecodes with native methods. To do this, the virtual machine exports a native method interface for people who want to write native code, providing native code consistent and convenient access to virtual machine services and the data in the Java class environment.

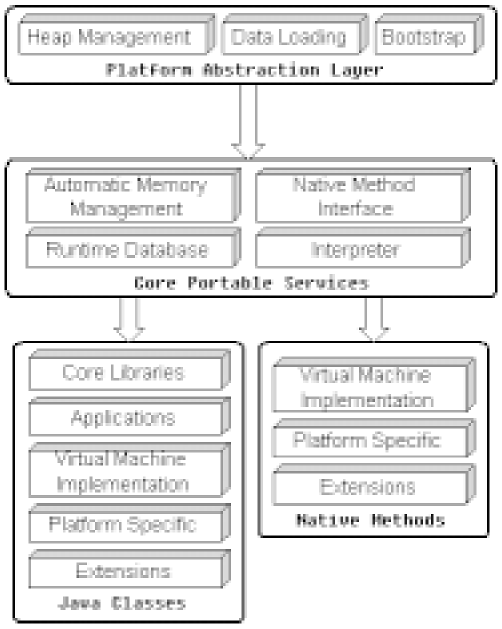


Figure 3 - The client architecture.

Classes and Native Methods

Native methods and Java classes comprise the third and final layer. Native methods let you extend the virtual machine without modifying the core virtual machine services, in a manner not possible with Java code. Native methods are suitable for implementing advanced virtual machine features, such as getting the name of a class and adding platform-specific features such as getting the time under PalmOS.

You can also use Java classes to implement virtual machine and platform-specific functionality. By keeping the native methods primitive and factoring out to Java classes more advanced capabilities using these primitives, the virtual machine becomes more flexible since Java code is more flexible than native code. An example of this technique is the PalmOS AWT described in the next section.

The PalmOS Ghost Machine Implementation

PalmOS devices do not have desktop-level computing resources, making it a challenge to create software like the Ghost Machine. The PalmPilot is limited to one megabyte of battery-backed RAM that maintains its contents even when the device is powered down. It uses a Motorola 68K CPU that runs at 17 MHz. Other devices may have more or less memory but all currently share the same CPU.

The PalmOS also has many input/output constraints that affect the Ghost Machine AWT implementation. The screen is fairly small and supports only 160 X 160 pixels. Color depth is limited to two bits and the PalmOS supports only one bit depth. Input is from a pen device and text can be entered using Graffiti handwriting recognition.

The PalmOS is designed to be fast and efficient given its resources. Because of this, there are many memory-model limitations imposed by the operating system. Applications and databases cannot exceed 64 KB. The runtime application heap is very limited under PalmOS 2.0 – it ranges from 14.5 KB (including stack space) to around 40 KB if TCP/IP is not being used. PalmOS 3.0 is a bit more lenient, allowing around 36 KB for the runtime application heap.

The Server

The server is currently implemented as a Java command line tool. It transforms groups of Ghost Machine-processed classes into PalmOS database files for loading using HotSync. The server also keeps track of its mappings in a persistent text file.

The grouping of classes to be installed in a single database file is located in a list file, with an extension of “LIST”. Each class must be listed by its full internal qualified name: `Object` becomes `java/lang/Object`, not `java.lang.Object`. Care must be taken to ensure that the generated database files are not greater than 64 KB.

The server can be invoked on each LIST file this way:

```
java ghost.classinfo.PilotProducer
  <local classpath> <system classpath>
  <base name of list file>
```

The first two arguments are paths to the local and system classes. They can either be the path to a directory or to an uncompressed zip file. The local classpath should include application classes as well as ghost classes. The system classpath should include the classes included with the JDK 1.0.2 distribution.

The base name of the list file is the name of the list file without its extension. If the server processes the list of class files successfully, it creates a PalmOS database file with the extension “PDB” and the same base name as the list file. Installing the PDB file on the PalmOS device overwrites any other data previously installed using a PDB file generated by the server with the same name.

The intermediate type list is created to facilitate installing new native code. If only Java applications are being installed, the intermediate type database does not need to be rebuilt. Each intermediate type list is tightly coupled to the native code that depends on it – if the intermediate type list changes, the native code that depends on it needs to be rebuilt.

In the PalmOS implementation, the intermediate type list is kept in a file with the extension "NAT". To rebuild it, enter the following:

```
java ghost.classinfo.PilotProducer –s
 <local classpath> <system classpath>
 <base name of nat file>
```

The resulting file has the extension "_MAP.PDB" added to the base name of the NAT file. This file should be reinstalled on the PalmOS device. Currently, only one NAT file, SYSTEM.NAT, is supported in the virtual machine. Both native methods and core virtual machine services use this when they need to access type-specific resources from Java classes. When installable native code becomes more common, support for multiple intermediate type databases will be added.

The Client

The only client layers that require PalmOS platform-specific source code are the PAL and the installable layer that contains Java classes and native methods. When possible, platform-specific functionality is expressed with Java classes using native methods only when necessary. Heap management and the AWT implementation are especially complex and consume a majority of the PalmOS Ghost Machine.

As mentioned before, PalmOS is very restrictive about how much heap space an application is allowed. Typically, there is more memory available on a PalmOS device, it just cannot be allocated directly to the virtual machine. To get around this, the Ghost Machine completely bypasses the application heap for dynamic memory requests and instead uses heaps reserved for storage. These heaps are limited to 32 KB in size and only one heap is allocated for garbage-collected objects. Unlike the application heap, storage heaps are not directly enabled for writing; a system semaphore must be obtained when writing to a storage heap. In addition, this semaphore must be released before a PalmOS system call is made. This restriction must be taken into account when writing PalmOS-specific native methods.

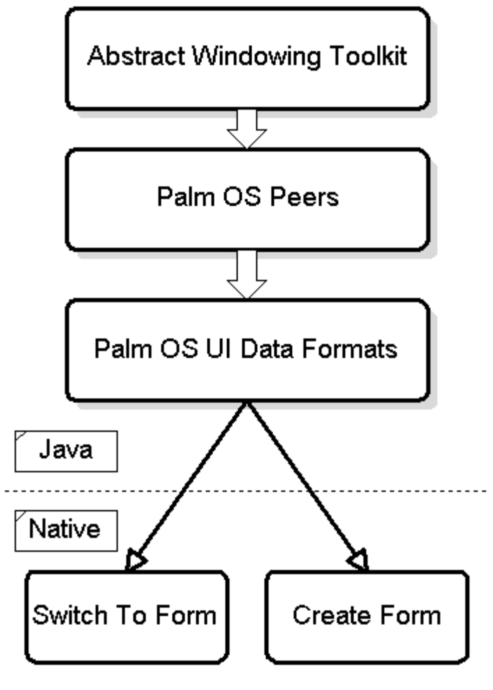


Figure 4 - The AWT implementation.

The AWT

Like the client virtual machine architecture, the PalmOS implementation of the AWT is layered (see Figure 4). To implement AWT on a new device, an abstract factory known as a toolkit must be implemented. This toolkit has implementations of both AWT peers and graphics. The graphics implementation contains methods for natively accessing routines to draw directly to its screen. The peers implementation, however, is very unconventional – interfaces directly with the PalmOS native user interface support.

The PalmOS user interface is built upon the concept of a form. Each form is composed of widgets such as controls, labels and fields. The data of each widget and its form are stored into a resource database. When creating a user interface using conventional PalmOS programming tools such as Metrowerks' Constructor and GNU tool's PiLRC, these databases are created when the application is compiled and linked into the final PalmOS executable. This makes the user interface static.

In a Java application, AWT objects are created and added to enclosing containers at runtime. Instead of forms, frame containers which can contain both sub containers and widgets are used. It's almost impossible to statically determine the user interface of a Java application. How can the dynamic AWT be implemented on the static PalmOS user interface?

Because user-interface objects are merely databases, they can be generated on the client on the fly and attached to application as resources. When the frame components change, the database must be recreated. This makes dynamic AWT features expensive but at least possible.

The data formats of the PalmOS user-interface widgets are represented in Java classes. The peers need only create the proper data formats for the database; when the frame is shown the database is made into a resource and displayed.

Installation

The Ghost Machine binary distribution is available at www.cs.washington.edu/homes/mcdirmid/ghost. This distribution contains both the server and precompiled client. You can use it to install new applications. It can be uncompressed using either WinZip or standard UNIX tools. Extract the distribution to its own directory.

The Client

To install the Ghost Machine client on a PalmOS device, the device should have at least 256 KB free. Install the single PRC file found in the distribution's root directory. Then install all PDB files found in the CLASSES subdirectory. Check the Memory control-panel to ensure that all components are correctly installed. The client should use around 178 KB of memory.

The Demo Programs

Three simple demo programs are installed by default (the source code for these demos is included with the Ghost Machine distribution).

TicTacToe, shown in Figure 5 on the next page, is a very simple implementation of the game of the same name. This is the first demo run under the Ghost Machine in December of 1997. It is almost impossible to beat this game.

Draw, shown in Figure 6, demonstrates a few user interface widgets but mainly demonstrates a simple drawing tool. The graphics implementation is slow and unoptimized so the demo is a little shaky. Touch the stylus in the middle of the screen then drag it halfway across the screen, lift the stylus from the screen and repeat. To clear the screen, press the Reset button.

Route, shown in Figure 7, demonstrates user-interface widgets created with AWT. Because of the complex layout, this program takes longer to load than the other demos because it builds a static database for the form to display. The application stores records in a vector and lets you review those records. Each record has name, phone, street, city and state fields. The Insert button creates a record using the current information in the fields and inserts it into a vector. The Clear button clears all fields (but does not remove a record if applicable). The Next and Previous buttons cycle through the records in the vector.

A special feature you can access using the menu bar is auto complete – the Route/Query Record menu command attempts to match the current information in the fields with a record in the vector. For example, if you enter a record with name "Mary" and phone "123-4567", merely entering "ma" into the name field and selecting Query Record displays the record on the screen if no other records before it in the vector match.

Route/Query City also auto completes a city based on the state entered. For example, if you select Washington as the state and enter "sp" entered in the city field, Query Record" causes "Spokane" to appear in the city field. Auto complete is a useful feature for applications running on the PalmPilot since complete text entry can be time consuming.

The Server

To install the Ghost Machine Server, you need both JDK 1.0.2 (archived at Sun's Java site) and JDK 1.1. The JDK 1.1 must be in your PATH environment and JDK 1.0.2 must not. Also, the environment variable JDK102 must be set to the directory of the JDK 1.0.2 installation.

Once you install the server you can install new applications. After setting the environment variables test the installation by rebuilding a list file. While in the classes directory, execute the following command:

```
java ghost.classinfo.PilotProducer .
 %JDK102%/lib/classes.zip apps
```

This rebuilds the APPS.PDB file. The command should not generate any output messages. Reinstall the file and test the demo applications to ensure that the server is installed correctly.

New applications can be installed onto the Ghost Machine Client using the Ghost Machine Server. The head class of a Ghost Machine application must be a subclass of `java.applet.Applet` (this is akin to executing in a web browser). The applet interface is convenient since it has well defined lifetime (`create`, `init`, `start`, `stop`, `destroy`, `finalize`). Also, an applet is not expected to create its own window to run in. Rather it runs in the space allocated by the container application. The applet interface simplifies the logistics of installing an application on the Ghost Machine.

To install an application on the Ghost Machine, the Java classes that make up the application must be located relative to the classes directory. If the classes are located in an anonymous package, this is the classes directory. The head class of the application must be listed in the apps list file. The apps list file is special since the driver application considers all classes in the generated PDB file eligible for execution and lists them in the application selector. Other classes should be listed in the AUXAPPS list file since the user should not consider them for direct execution.

After the classes are added to the list files, execute the following command while in the classes directory:

```
java ghost.classinfo.PilotProducer .
 %JDK102%/lib/classes.zip apps auxapps
```

Reinstall the generated PDB files onto the device. The newly installed application should appear in the application selector by its name in the order that it appears in the apps list file.

Deinstallation and System Crashes

It is always possible that the Ghost Machine may crash the PalmOS device it is running on. Since the PalmOS does not protect the operating system from its applications, this can lock up your device. First, power-cycle it. If this doesn't work, do a hard reset to prevent the it from remaining locked up until its batteries are dead. Doing a hard reset by sticking a paper clip in a small hole located on the back of the device causes you to lose all your data. When experimenting with the Ghost Machine, HotSync frequently to protect data.

Supported Features

Although it's possible to implement all of these Java features eventually, long and floating point arithmetic and finalizers have not yet been implemented. The PalmOS does not expose a native thread interface and green threads, threads that are emulated and not natively, will be hard to do reliably because of realtime constraints of the PalmOS event queue.

APIs

The Java 1.0.2 version of the APIs was chosen because it has a small memory footprint in its core libraries (about 300 KB compared to the 1.5 MB of the core Java 1.1 libraries). Almost a third of the 1.1 library bloat is devoted to internationalization. The `java.text.*` package is over 500 KB and `java.lang.Character` is 139 KB (it was only about 15 KB in Java 1.0.2). Even after processing, this is too large to fit onto a PalmOS device. Java 1.1 is just simply too large in its current implementation.

Most AWT features are or will eventually be supported. The following components are partially functional now: `Frame`, `Canvas`, `Window`, `Panel`, `Button`, `TextField`, `Label`, `Choice`, `Checkbox`, `MenuBar`, `Menu` and `MenuItem`. Only a few routines to draw lines, circles and strings are supported in the graphics implementation.

Some 1.0.2 APIs do not work because the native methods they depend on have not been implemented. These include file system accesses, networking, dynamic class loading, native arithmetic functions, most `java.lang.Runtime` methods and the security manager.

Example Application

Here's a basic "Hello World" for the Ghost Machine environment on the Palm Pilot. The class is rather simple:

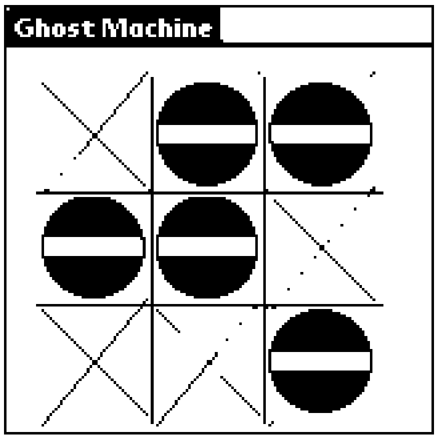


Figure 5 - TicTacToe.

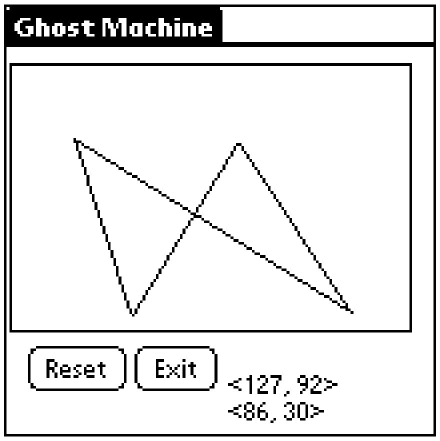


Figure 6 - Draw.

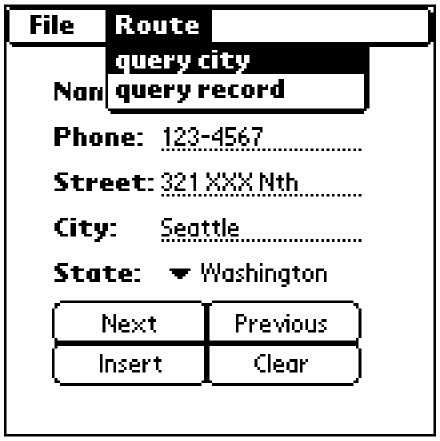


Figure 7 - The Route program.

```
import java.awt.*;
public class HelloWorld extends java.applet.Applet {

  public void init() {
    setLayout(new GridLayout(0, 1));
    add(new Label("Hello World"));
    add(new Label("hELLO wORLD"));
    add(new Button("EXIT"));
  }

  public boolean action(Event evt, Object what) {
    if ((evt.target instanceof Button) &&
      (((String) evt.arg).compareTo("EXIT") == 0)) {
      System.exit(0);
      return true;
    }
    return false;
} }
```

The applet places three user interface objects in one column using a grid layout (see Figure 8). The first two rows are labels. The last row is an EXIT button. When the EXIT button is pressed the application exits. This applet, when compiled, executes in a web browser or from within appletviewer.

To load this applet in a PalmOS device insert a line with only "HelloWorld" into the APPS.LIST file in the classes directory. Recreate the APPS.PDB file by running:

```
java ghost.classinfo.PilotProducer .
  %JDK102%/lib/classes.zip apps
```

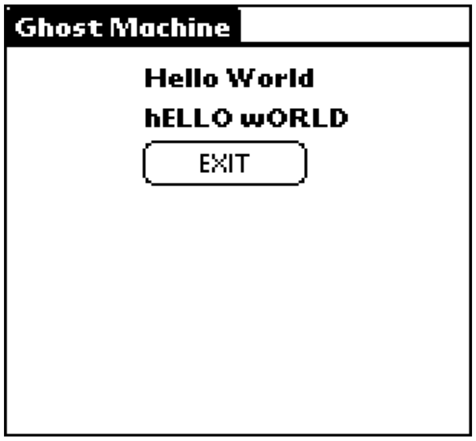
Install the new APPS.PDB file onto the PalmPilot, overwriting the previous one. HelloWorld is now ready for execution. Experiment adding text fields, buttons and checkboxes to the applet.

Conclusion

The Ghost Machine architecture was created to let Java applications run on mobile computers. The current implementation is a proof-of-concept that targets the PalmOS platform. It is very much a work in progress with the potential to grow into a thriving platform that turns mobile devices into first-class platforms for third party applications.

High-priority Features

Currently there is no mechanism in the Java API for a Java application on a client to interface with desktop synchronization. This is absolutely necessary if Java is to become a premier mobile computing platform. Java applications running on desktops should be able to synchronize data with Java applications running on mobile devices under a model similar to conduits and HotSync technology. With that functionality in place, the Ghost Machine platform will have implemented true end-to-end functionality and will be ready for useful applications.



Hello World
hELLO wORLD

Figure 8 - Hello World.

In order to upgrade to Java 1.1+, a version of the class libraries will have to be found or built. It must be scalable in memory footprint down to around 400 KB of unprocessed class files (which would be about 200 KB after processed by the server). Some features that are core to Java 1.1 are hard to implement given the dependencies on the introspection of a class yielding the original string types for attributes. This includes reflection, beans and RMI. These features will be nontrivial, though not impossible, to support. By including the server mappings of type strings to integral identifiers on the client, the client could support reflection at an expense of losing space. Such a database on the client would be able to take advantage of representing all classes installed on the client (no redundant information) and compression (since this information may not be used often).

The performance of a virtual machine relates to speed, response time and memory footprint among other things. Improving the performance of the Ghost Machine platform is an ongoing effort. There are many parts of the Ghost Machine that either need retuning or redesign. There are a few modules where most of the gains can be directly made. Garbage collection which currently uses a mark, sweep and compact algorithm is responsible for most runtime pauses. This algorithm is simple to implement but primitive compared to current garbage collection research. Switching to either a modern incremental or generational technique will go a far towards improving performance as experienced by the user. Also, reclaiming memory faster and efficiently can decrease the runtime memory footprint of the virtual machine.

Bytecode interpretation is slow, an order of magnitude slower than running native code. The first environment that allowed developer's to write PalmOS applications in Java, Jump, used native compilation to bring performance to an adequate level. The Ghost Machine platform currently uses an interpreter. With GUI-intensive applications, the lack of speed goes unnoticed because most cycles are devoted to native screen refreshes and waiting for user input. However, for more compute- intensive applications the interpreter is perceived as too slow.

It's possible to compile the bytecodes into machine code using the heavy-duty resources of the desktop workstation. Ideas for integrating Jump into the server or creating a bytecode-to-machine-code compiler from scratch are other options. There is also a question about the space efficiency of machine code as opposed to bytecode. The overall memory footprint of the virtual machine is a sensitive resource and every trade-off needs to be chosen wisely.

The Ghost Machine architecture is not exclusively designed for PalmOS devices, but for the whole class of mobile devices. Porting to Windows CE and the Newton OS should be relatively easy (there is already an effort underway to port the Ghost Machine to the Newton). The Ghost Machine architecture should also be able to target smaller mobile devices such as smart cards and pagers. The question "How small is too small?" remains unanswered.

Low Priority Features

Core API support should be finished eventually. The PalmOS does support a pseudo file system in memory so implementing the Java file system APIs is possible and should be done. The PalmOS also includes a TCP/IP stack and you can attach a wireless modem to a PalmOS device for constant connectivity. Full support should eventually be added for long and floating-point arithmetic. This advanced arithmetic must be completely emulated and floating-point must operate according to the IEEE 754 standard.

Open Source Code

The Ghost Machine source code is available for download under a GPL license. It is ideal for those who want to hack a virtual machine and also for those who want to see the Ghost Machine evolve. Feature and code contributions are greatly encouraged. Please contact the author at mcdirmid@cs.washington.edu for more information.✔