Trimming Java Down to Size

UCSC-CRL-97-22

C. E. McDowell* M. R. Allen† E. A. Baldwin‡ B. R. Montague*
M. Montoreano*

August 26, 1997

Abstract

Java is both a programming language and a collection of libraries called packages. Much of the interest in Java is due to the large collection of existing packages which allow complex applications to be built quickly. Perhaps even more important, many programmers find the Java language to be a more desirable language than C or C++, Java's primary competitors. The advantages of Java apply to any application domain that previously used C or C++, including embedded systems. However, typical Java systems today require 4-8 Mbytes of RAM or more, including those Java systems supplied by traditional embedded systems companies. JavaCam is an embedded Java application that can operate with as little as 1 Mbyte of memory. In this paper we describe how we were able to build a system that runs applications that exercise all of the Java language features in as little as 1 Mbyte of memory.

Keywords: Java, class unloading, embedded systems, Internet device.

1 Introduction

Java is both a programming language and a collection of libraries called packages. Much of the interest in Java is due to the large collection of existing packages, which allow complex applications to be built quickly. Perhaps even more important, many programmers find the Java language to be a more desirable language to program in than C or C++, Java's primary competitors. The advantages of the Java language apply to any application domain that previously used C or C++, including embedded systems. However, typical Java systems today require 4-8 Mbytes of RAM or more, including those Java systems supplied by traditional embedded systems companies. Recently, JavaSoft released its PersonalJava API which is designed to run on systems with 2 Mbytes of RAM and 2 Mbytes of ROM. PersonalJava applications are still expected to have significant graphical user interfaces and as such the API includes a rather large subset of the Java Abstract Windowing Toolkit (AWT). We are interested in building embedded Java systems that are mini-servers, not mini-clients. As such, our systems have no graphical displays and hence do not need any part of the AWT. The result is that we have running systems that support the complete Java language, including all of the core Java packages except AWT, in as little as 1 Mbyte of RAM - with no additional memory, including no disk or ROM beyond a small boot ROM. Such a system is capable of running non-trivial Java programs, although the memory space available to the application is limited to a few 100 Kbytes.

Java provides remote execution mechanisms which can support dynamic software updates and dynamic system reconfiguration. These mechanisms are similar to the mechanism that makes possible Java applets, which are the primary source of interest in Java by the Internet community.

Java applets are programs (technically portions of programs) that are loaded from remote servers onto local machines and then executed inside of another program that contains the Java Virtual Machine (JVM). Java provides

*Computer Science Dept, University of California Santa Cruz.
†Work completed at UCSC, now with JavaSoft.
‡Work completed at UCSC, now with National Semiconductor Inc.

mechanisms for a program to safely execute unknown applets without fear of compromising the security of the local machine. Using these same mechanisms, it is possible for a locally executing program to send program fragments to a remote machine, and have the remote machine execute the fragments by incorporating them into a remotely running Java program. The remotely running program can thus be updated with new versions of existing capabilities, or provided with new capabilities not originally configured into the system.

Many potential embedded applications do not require a GUI running on the embedded device. With Java and the Internet, this does not mean the application cannot have a graphical end-user, it simply means that the interface is not running on the embedded device. The GUI for the embedded device could be an applet, served up by the embedded device to another computer that supports graphical output and the AWT. An example might be a VCR that is programmed via an applet. At home you could use your set-top web browser to program the VCR. But you could also program your VCR from your office if you found out that a great show was going to be coming on before you could get home. You could similarly check on the security system in your home, or have messages from your home answering machine mailed to your office. A bank employee could check on the status and possibly adjust various options on an ATM remotely from the main office. A factory supervisor could access and manipulate the various control systems spread around the factory floor. You do not need Java to do any of this, but if Java is the preferred choice purely from the point of view of the programmer, then we believe we have demonstrated that Java is a viable option for systems that can afford as little as 1 Mbyte of RAM.

In order to determine how small a Java system we could build today, we built one. We wrote a very thin OS which we call JN, for Java Nanokernel. Its primary purpose is to support a Java Virtual Machine and be as small as possible. We then ported Sun's JDK1.0.1 to JN. As suggested above, this port includes the complete Java language and all of the core Java packages except `java.awt`. Our first significant application was a camera that is remotely programmable in Java and is Internet accessible. We call this system JavaCam.

2 JN

JN[Mon97b] follows a classic soft-real-time architecture informally known as a Cutler kernel. Although he was not the first to use the architecture, David Cutler led teams that used this architecture to implement the kernels for RSX-11, VMS, and NT.

The kernel consists of a single work-loop driven by a queue of control blocks. The kernel begins execution in response to an interrupt. Once started, the kernel continues to execute until no control blocks remain in the queue. Each control block contains the address of a routine which the kernel must execute. Historically, these routines have been called *fork* routines in the Cutler kernels. Other common names for such routines, especially as found in I/O managers, include *Second Level Interrupt Handler* (SLIH) and *Deferred Procedure Call* (DPC). Since fork routines cannot block and must have a bounded execution time, they have many characteristics of routines written for a hard–real-time environment. Neither the fork routines nor the kernel need to use explicit mutual exclusion to access global data structures, reducing the need for explicit synchronization.

Because JN is intended to support a Java Virtual Machine, there is no concept of protection or security in JN. All security is handled by the JVM.

The JN file system supports a Unix-like directory structure. The actual implementation is a flat file system supporting long filenames which can contain slashes. This gives the appearance of a hierarchical file system, as required by the JVM. In addition, our implementation supports Unix sparse-file semantics in that only sections of the file that contain actual data are allocated space. Currently our file system can reside either in volatile RAM or persistent RAM on a PCMCIA RAM card.

JN currently runs on two embedded processors from National Semiconductor - the NS486SXF and the CR32. The NS486SXF is a "single-chip" 32-bit 486 PC. Included on the chip are a PCMCIA controller, a UART serial port, an enhanced bidirectional parallel port, an LCD display controller, infrared serial control, a real-time clock/calendar, a watchdog timer, programmable interval timers, two peripheral interrupt controllers, a serial high-speed synchronous interface (Microwire), a degree of power management, a DMA controller, a DRAM controller, and a bus interface unit. The bus interface unit can control the standard PC ISA bus or the PC/104 embedded system variant of ISA. We have currently implemented device drivers for the clock/calendar, the serial UART, the parallel port, and the PCMCIA. In addition we have implemented a driver for an Atlantic Ethernet card. The CR32 configuration is similar.

The JN Nanokernel consists of approximately 2000 lines of C. This does not include the KA9Q TCP/IP stack which has been ported to work with JN.

3 Porting JVM

We originally ported the JVM from Sun's JDK1.0.1 to JN running on the NS486SXF. This port includes all Java language features and all of the core Java packages except `java.awt`. The JVM, as implemented in JDK1.0.1 and modified to run on JN, requires about 183Kbytes of memory, not including any dynamic data structures.

Most of the OS support required by the JVM in JDK1.0.1 is listed in a file, `sys_api.h`, that is part of the standard JDK1.0.1 distribution. There were a few direct calls to Unix routines that were not listed in `sys_api.h`. In a port to a traditional operating system, it would be necessary to write some interface glue to map the calls that the JVM code was making into the actual calls supported by the OS. Because JN did not have a fixed API, we simply modified the JN API to exactly match that needed by the JVM code. We determined what the JVM needed by removing the interface files from the Unix version of JDK1.0.1 and trying to link. The missing routines were the ones we needed to implement. It was only somewhat later that we discovered that the public file `sys_api.h` included most of what we needed. We would still have had to do the exercise, because `sys_api.h` is not complete.

The support required by the JVM code falls into six categories - thread routines, monitor routines, file routines, exception handling routines, socket routines, and various miscellaneous routines which are largely standard Unix routines such as `malloc()` and `free()`. The thread routines are such things as `sysThreadCreate()` and `sysThreadResume()`. The monitor routines include `sysMonitorEnter()` and `sysMonitorExit()`. File routines are the usual read, write, open, close, etc.. The others are equally predictable. The details of the JN API are available in a technical report[Mon97a].

JN does not support the dynamic loading and running of programs. Instead, JN is linked with the application as a single image, that is, we link JN with the JVM. The JVM is run by calling the main routine of the Java interpreter.

A similar restriction applies to native methods used by Java classes. Because JN does not support dynamic linking, the standard Java method `System.loadLibrary()` is not supported. All required native methods must be linked into JN at build time along with the JVM. Native methods are still supported, they are just not linked dynamically. The other major change we made to get the JVM code, as written by Sun, to work with JN was to eliminate all code with virtual memory dependencies. We also made some changes in the handling of garbage collection and added class unloading as discussed below.

4 JavaCam

Our first significant application was a network camera [All97]. We wanted an example sensor that would be accessible via the Internet and that would demonstrate Java's built in support for dynamically updating code running in the embedded system. Applications for such a device include security and traffic monitoring.

JavaCam is an example of such a remote network sensor, controlled by Java, that is fully programmable over the network without ever shutting down the sensor.

The camera we choose to use is the Connectix QuickCam, a commercially available product costing around $250. It uses a charged coupled device (CCD) to take a digital picture. The QuickCam receives control parameters and delivers image data using a standard bidirectional PC parallel port. The Connectix QuickCam camera always acquires an image 340 pixels wide (columns) and 250 pixels high (rows). A subset rectangle from this image may be returned from the camera on request. The camera can control the levels of hue, saturation, contrast, black-level, white-level, and exposure time. The camera delivers pixels in 24-bit color, that is, a byte of red, green, and blue is returned for each pixel. The camera operates in one of three decimation modes, indicating whether the camera is to deliver all rows and columns, every other row and column, or every fourth row and column. The JavaCam user can specify these settings via the Java camera control applet inside a Web browser or via a CameraControl servlet as described below.

The camera driver provides the basic interface to the camera. It allows higher-level programs to send commands to the camera and provides a simple interface to receive images. The driver communicates with the camera over the parallel port. The driver provides a basic API of three functions:

```
void qc_initialize();
int  qc_send_command( unsigned char command, unsigned char parameter );
int  qc_take_picture( char *array, char mode, int length );
```

These functions are implemented in C and called by Java native methods.

Function `qc_initalize()` resets the camera to a known state with all settings set to default values. It also terminates any camera operation already in progress.

The `qc_send_command()` function sends a command and argument to the camera. Each command to the QuickCam consists of two bytes - a command and an argument to the command. The camera echos the command and argument back to the driver, in order to verify command reception.

The function `qc_take_picture()` obtains an image frame from the camera. It has 3 arguments: a buffer in which to place the image pixels; a bitmask representing the camera mode (specifying color scheme and decimation level); and the length of the expected image in bytes. Presumably, the program has already sent any other parameters to the camera via `qc_send_command()`.

4.1 JavaCam Internet Systems

JavaCam provides two distinct ways to access the camera over the Internet at the Java level: the *applet* method and the *servlet* method.

The *applet* approach works as shown in Figure 1. In this approach there is an HTTP server running on the JavaCam system. When a WWW browser issues an appropriate `GET` command to the HTTP server, it hands back a page containing a Java applet. This Java applet, called the `QuickCamApplet`, communicates back across the network and contacts another server running on the JavaCam system, the `QuickCamServer`.

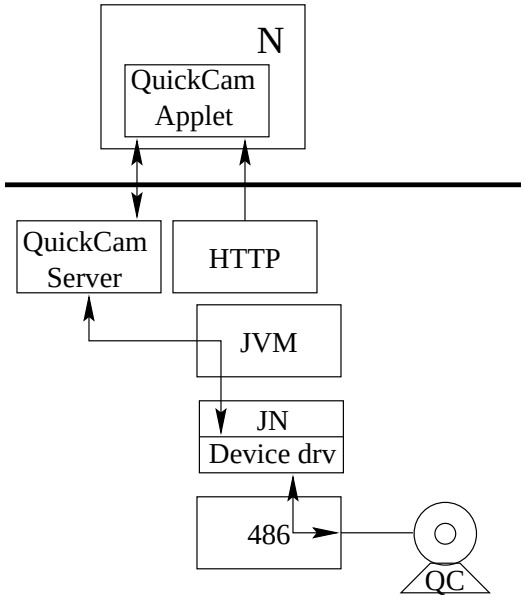

Figure 1: JavaCam being controlled by an applet.

The `QuickCamApplet` sends parameters to the `QuickCamServer`, and then makes a request for an image. The `QuickCamServer` contains a `QuickCam` object which contains native methods that call the JN camera driver routines. The native methods fill in the image buffer and hand the buffer back to the `QuickCam`. The server then sends the data over the network back to the `QuickCamApplet`, to be displayed inside the user's browser.

The *servlet* approach works as shown in Figure 2. In *servlet* mode, a `ClassLoaderServer` must be running on the JavaCam system. This class listens on a particular Internet port for requests to load servlets to control the camera. A servlet is akin to an applet; it is a Java class which implements a specific Java *interface*. Like an applet, a servlet is not an entire program. It requires another program to instantiate and fill out the missing parts of its expected run-time environment. An applet is downloaded from a server to a local application in order to extend the functionality of that application. A servlet is sent from a local application to a server in order to extend the functionality of that server.

In order to control the QuickCam on a JavaCam system, a class must implement the `CameraControl` interface. A `CameraControl` servlet, if authorized, is instantiated and given a reference to a `Camera` object, which provides access to the QuickCam JN device driver. The instantiated servlet is also given a socket providing a connection back

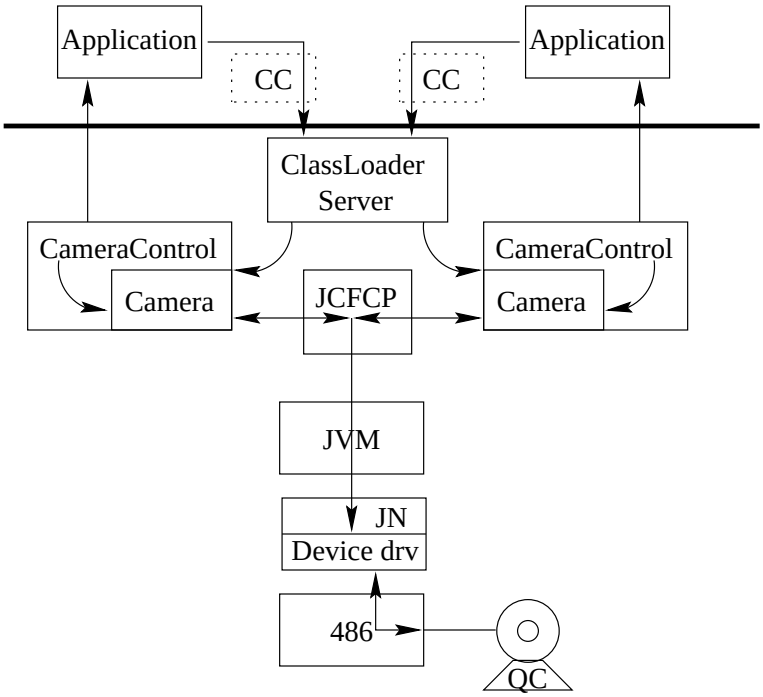

Figure 2: JavaCam being controlled by a servlet.

to the client which provided the servlet. Once instantiated and started, the `CameraControl` servlet can perform any desired processing of camera data before sending data back to the client. Like an applet, the servlet runs within a `SecurityManager` class. The `SecurityMangager` checks all servlet operations to make sure that no unauthorized operations are performed. For example, in JavaCam, servlets are not permitted to access the file system or open arbitrary network connections.

5 Fine tuning memory requirements

In a system with a tight memory budget, it is important not to waste memory. We identified two primary areas where it was possible to reduce memory requirements. First, our embedded application does not use all the core Java classes, therefore storing them is wasted memory. Second, prior to JDK1.1, the memory used for the system data structures associated with dynamically loaded classes could never be recovered, even when the class was no longer needed or even accessible. The Java Language Specification provides for class unloading, but it was not implemented in JDK1.0.1. We have implemented both class unloading and a tool that allows us to identify and load just those classes needed by our application.

5.1 Custom subsetting of the standard classes

In a traditional embedded system, all routines needed for the application are linked together into a single executable image that is loaded onto the embedded system. This is exactly what we do with the C portion of our system, i.e., JN and the JVM. In JavaCam, the real application is a Java program that is dynamically loaded into the running JVM. Because of Java's late binding semantics, it is not a normal part of Java development to identify *all* of the classes that will be needed by a particular Java program. The necessary classes are simply loaded on the fly as the program executes. In non-embedded systems this is not an issue, because the nearly 1.5 Mbytes required for all the JDK1.0.1 core Java classes is not a significant amount of storage. For many embedded systems, this much storage is unavailable. In JDK1.1 this requirement has shot up to nearly 10 Mbytes. Even after we eliminated the package `java.awt`, which we do not support, we were still left with approximately 350 Kbytes for the class files, a significant portion of

a 1 Mbyte budget.

We have developed a tool that examines the class files recursively starting from the main application class, identifying all classes that are used by the application. Using this tool we now only need approximately 250 Kbytes for the class files for our JavaCam application, a savings of 10 percent of our 1 Mbyte budget. The only classes that are missed by our tool are classes that are loaded by name, as is done with the method `Class.forName()` in the standard package `java.lang`.

For applications that do not use a custom class loader to load classes from a network or other source, our tool will identify all classes loaded during the execution of the application. If classes can be loaded by a custom class loader, as is done in JavaCam, then there is no way to know in advance what classes will be needed. Our solution is to provide a tool that can be used to filter classes that are intended to be downloaded and notify the user if the classes do not conform to the subset loaded on the embedded system. This is the same approach used in the subsets defined by JavaSoft such as JavaCard and PersonalJava. Sending a JavaCam system a servlet that requires some standard class that is not loaded on the JavaCam system will not crash JavaCam, it will simply result in a failure of the servlet.

Another solution that we have not yet implemented is to allow certain standard classes that have not been preloaded to be loaded from the same location as the servlet. This could not be done for all classes because of security reasons.

5.2 Class unloading

When a Java program uses a class the first time, the JVM loads the class and builds some internal data structures. The Java Language Specification (JLS)[GJS96] allows for the unloading of classes, i.e., the reclaiming of these internal data structures. Without class unloading, a Java Virtual Machine (JVM) is like an OS that never releases the memory allocated for the code space of an application, even after the application has completed. In particular the JLS states that "This can be used, for example, to unload a group of related types... Such a group might consist of all the classes implementing a single applet." Class unloading is important for any long running Java program that continuously loads classes, uses them for some time, and then no longer uses them. This is exactly the type of behavior that occurs with servlets sent to our JavaCam. The servlets are Java classes that are intended to be loaded into JavaCam, and used only for the duration of a single connection. One of the features of Java that makes it attractive for many applications is the security model that makes this dynamic loading and execution of untrusted program fragments possible. It is essential that classes be unloaded periodically. Otherwise the computer's memory will become cluttered with unused classes. This is marginally tolerable in a system with virtual memory or one in which the JVM is frequently restarted. For long running or embedded systems without virtual memory, such as JavaCam, class unloading is necessary.

We completed the addition of class unloading to our port of JDK1.0.1 about the same time the JDK1.1 was released [Bal97]. JDK1.1 now includes class unloading. Our implementation of class unloading behaves similarly to that of JDK1.1, however, we are a bit more conservative about unloading classes. We believe the JDK1.1 implementation violates the requirement of the Java Language Specification which states that for static fields "there exists exactly one incarnation of the field, no matter how many instances (possibly zero) of the class may eventually be created." This is discussed in detail in another paper[MB97].

6 Conclusion

We have implemented an embedded Internet device in which executing Java programs can be dynamically updated, that is, altered on-the-fly. We can build useful devices in this environment with as little as 1 Mbyte of memory.

References

[All97] M. R. Allen. *JavaCam: Java Enabled Internet Camera*. M.S. Thesis University of California, Santa Cruz, 1997.

[Bal97] E. A. Baldwin. *Memory Management in Embedded Java*. M.S. Thesis University of California, Santa Cruz, 1997.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[MB97] C. E. McDowell and E. A. Baldwin. Unloading Java classes that contain static fields. Technical report, U. of Calif. Santa Cruz, UCSC-CRL-97-18, 1997.

[Mon97a] B. R. Montague. JN external API. Technical report, U. of Calif. Santa Cruz, UCSC-CRL-97-17, 1997.

[Mon97b] B. R. Montague. JN: OS for an embedded Java network computer. *IEEE Micro*, 17(3):54–60, May/June 1997.