An Implementation of Java Personality and Communication Facility on a Real-time Kernel(I)

Abstract

The goal of this project is to research and implement the associated environments on top of our real-time kernel. Because Java's open technology is becoming the de facto standard for building networked applications, Java applications have the advantage of high portability that can be executed on cross-platforms. This promotes application sharing and saves application development costs. Our goal is to provide the Java personality and componentized communication facility on the real-time kernel. We use TCP/IP Stacks as our communication protocol since TCP/IP is widely used as internet communication standard. We will provide a Java platform which will serve as a testbed for our research of Java technology. We plan to develop real-time embedded Java based on our real-time kernel and explore the research issues and applications of Java Network Computers.

In this report we will discuss the design and implementation of the Java Virtual Machine and the TCP/IP communication module.

Keywords: Java; Java Virtual Machine; TCP/IP; Real Time; Embedded Systems; Component
be added, deleted or modified. There are two goals in this project. One is to develop a Java environment\[1]\[2] on top of our real-time kernel and adjust it according to the real-time and embedded requirements. The other is to develop a communication component to support TCP/IP since it is the most common internet protocol, and most operating systems such as Linux, BSD support it. We also try to optimize the TCP/IP component based on the requirements of the real-time and embedded systems.

3. Results and Discussions

3.1 Results and Discussions of JStar

JStar is the name of our Java Virtual Machine[3][4][5][6]. An overview of JStar is shown in Figure 1. There are four components in JStar which are class loader, execution engine (interpreter), native methods, and the garbage collector. Class loader reads in the binary unzipped class files from disk or other input/output device on system. The execution engine performs the methods invocation, including general methods, abstract methods, and the native methods invocation. During the methods dispatching and invocation, garbage collector will do the implicit garbage collection works to free the object that allocated but now is not referenced any more. Once we invoke the native methods, the native routine package that we implemented will be called through JStar native interface to perform the low-level works.

To link JStar library, the conventional C programs can invoke kickJava(args) with including the “JStar.h” header file. The example in Figure 2 shows the Java source that we want to execute. This Java application contains only the main method and print a string, which is “Hello Java !!”, on the standard output device. Figure 3 shows how JStar can be linked with the header file named JStar.h. Once the JStar is linked in C program, we can make a call to perform the Java application execution at run-time.

```java
public class HelloJava {
    public static void main(String args[]) {
        System.out.println("Hello Java !!");
    }
}
```

Figure. 2 The Java Example to Print “Hello Java” onto Screen

```c
#include "JStar.h"
void main(void) {
    kickJava("HelloJava");
}
```

Figure. 3 The Example of a Conventional C Program Calling Java Application Through JStar API
3.2 Results and Discussions of the TCP/IP Component

Implementing a TCP/IP component from scratch on our real-time kernel cost a lot of design and debug efforts. Therefore, we instead reuse the Linux TCP/IP code and integrate the code into our kernel.

To integrate the Linux TCP/IP code into our kernel, we make the following efforts. First, we trace and analyze the Linux TCP/IP code. We modify part of the code to adapt it to our kernel. And then, a kernel support module is implemented to integrate the kernel and the TCP/IP code. At last, we provide a socket-like interface to applications. Figure 4 shows an example of our implementation to create a socket.

```c
int socket(int family, int type, int protocol) {
    int err;
    NO_PREEMPT();
    err = sys_socket(family, type, protocol);
    PREEMPT_OK();
    return err;
}
```

Figure 4 Implementation of Socket()

The routine NO_PREEMPT() ensures that there is at most one thread in the execution of the system call `sys_socket()`.

4. Evaluations

There are three experiments used to evaluate JStar. We measure both start-up time and total running time for Kaffe[7] and JStar for every test case. The first is to print the `Hello Java` string onto standard output device. The second one is to invoke a method within the same class. Finally, the string insertion is performed.

The experimental environment for the JStar is illustrated in table 1. We use the same hardware configurations, but evaluate Kaffe and JStar separately.

<table>
<thead>
<tr>
<th>JVM</th>
<th>Kaffe</th>
<th>JStar</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Pentium 300 MHz</td>
<td>Pentium 300 MHz</td>
</tr>
<tr>
<td>MEMORY</td>
<td>64 MB</td>
<td>64 MB</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux 2.0.30</td>
<td>Windows NT 4.0</td>
</tr>
</tbody>
</table>

Table 1. The Experiment Environment

Within each experiment, the Java test case is executed for twenty times. The iX86 CPUs support the functionality to record how many CPU cycles were used by an application used. We insert the inline assembly into the source code to obtain the actual numbers of the CPU cycle, which are spent by every application that runs on Kaffe and JStar. We count the CPU cycles for both the start-up time and total running time. To evaluate the start-up time of Kaffe, we disable the garbage collector and skip the initialization of exception handler since these two functions are still under testing. Figure 5, 6, 7 show the results of the three experiments.

From these figures, we can see that JStar outperforms Kaffe. For example, in Figure 5, both the start-up time and the execution time of JStar spends only 25% of Kaffe’s.

Figure 5 The Comparison of Kaffe and JStar in printing Hello Java.
For the TCP/IP component, we have completely integrated the whole Linux TCP/IP code into our kernel. This component will be suitable for future researches.

5. References