



Java and SNA: A Case Study (TR 29.2168)

By Matt MacKinnon, Adam King, and David Kaminsky
Network Studies, IBM
Research Triangle Park, North Carolina

ABSTRACT

This report describes a Java tool kit for CPI-C programmers. (Java is a registered trademark of Sun Microsystems.) As published by Sun Microsystems, Java supports on sockets programs on IP network. We describe a tool kit that allows programmers to write LU6.2 programs in Java. We also include preliminary performance measurements.

ITIRC KEYWORDS

- Java
- CPI-C
- Network Programming
- Internet Programming

ABOUT THE AUTHORS

Matt MacKinnon received a BS in Computer Science from Penn State University in 1994. He joined IBM in Research Triangle Park in 1995, where he works on CommServer for OS/2, along with Java projects.

Adam King is a Senior Associate Programmer working on an advanced technology Java project. Adam received BSs in Computer Science and Computer Science/Mathematics from Furman University in 1989 and an MS in Computer Science from Clemson University in 1991. He has been with IBM in RTP since 1991.

David Kaminsky is an Advisory Programmer and the team lead for an advanced technology Java project. David received a BA in Mathematics from the University of Virginia in 1988, an MS and MPhil in Computer Science from Yale University in 1990, and a Ph.D. from Yale in 1994. Dr. Kaminsky's dissertation investigated parallel programming over networks of workstations.

Introduction

Intranets are hot. So is good barbecue, and while we know what goes into good barbecue [1], there's a lot of disagreement about what comprises any intranet, let alone a good one. Since an intranet (like barbecue) is likely no better than its components, it's worth investigating what's in one.

The word "intranet" is derived from the "Internet," a world-wide collection of interconnected computers. However, an intranet is not merely a collection of interconnected computers--those have existed in enterprises since the 1960s.

Some might try to define an intranet by the networking protocol (or simply "protocol") used to ship information among computers--that is, intranets are networks of computers communicating via the "Internet Protocol" (IP).

Such a definition is narrow-minded: networking protocols are rather arcane subjects, left to the domain of a few computer bit-heads; the average computer user doesn't care about how packets are shipped around a network. He does care about applications, since they define how he interacts with his computing environment. The better the applications, the more productive he'll be. He also cares about cost--who doesn't?

User productivity is measured by the amount of work each person can complete--in this case by using the applications running on the network. It encompasses such factors as reliability and application "intuitiveness." "Cost" refers not only to the cost of the hardware used to build the network, but also to the cost of operating and servicing the network and the applications running on it.

With these factors in mind, we'll define an intranet as a low-cost, high-productivity network environment.

It's clear that for user productivity, IP offers great benefits. Web browsers such as Netscape's Navigator (Navigator is a trademark of Netscape Corporation) provide an intuitive, graphical way to locate any resource spread throughout IP networks [2]; SNA applications often present a character-based interface. Furthermore, relative to SNA, IP is quite easy to install on personal computers, thus lowering costs.

However, 70% of business data resides on mainframes and is accessed primarily through SNA, an enterprise-quality, open, networking protocol. The SNA protocol-handling code on most mainframes has been optimized for over a quarter-century and is now lightning fast; the mainframe IP code is still relatively immature and sluggish. So, compared to SNA, IP imposes a large burden on the mainframe and thus increases the cost of productivity.

Even when the mainframe's IP code matures, SNA will continue to offer benefits. SNA was designed for stability under heavy network loads; IP collapses meekly under similar circumstances [3]. Compared to IP, SNA's superior reliability characteristics will continue to offer productivity benefits. Furthermore, SNA includes transactional APIs which simplify the development of line-of-business applications.

So there will remain a place for both IP and SNA: IP is quite useful for the general Internet, where ease-of-use and ease-of-install are especially important; SNA is preferable in many intranets, where reliability is crucial.

In this paper, we'll focus on ways to improve SNA intranets by applying Internet technology.

One Internet technology that has garnered much attention lately is Java (a trademark of Sun Microsystems), a network-enabled computing environment developed by Sun Microsystems. Java is a useful technology for two reasons: It is platform-independent, so applications can be written once and run on a variety of computers; and it can be dynamically downloaded--that is, code can reside on a server and flow to the client as needed, thus reducing client-installation costs.

However, the Sun developers chose to release Java with IP bindings but without SNA bindings. We have removed this limitation. To improve Java's suitability for enterprise, high-reliability intranets, we've created a Java binding to CPI-C, an open API for SNA's LU6.2.

We recognize that the majority of SNA applications use LU2. We chose to do an LU6.2 binding first because it can run on peer workstations, making it a relatively simple learning platform; LU2 requires mainframe interaction, complicating development. With the skills developed during our LU6.2 work, we plan to push forth to other communications protocols.

To date, we've implemented the 16 CPI-C calls needed to write the five "classic transactions": Pipeline, Credit Check, Inquiry, File Transfer, and Database Update. Implementing the remaining 31 calls in CPI-C version 1.2 will be straightforward.

The remainder of the paper discusses our Java binding to CPI-C, including a discussion of the Java/CPI-C API and the development issues that we encountered. For our development, we used the OS/2 Warp operating systems and the CommServer 4.0 SNA stack and CPI-C API. Some familiarity with Java and CPI-C is assumed.

The Java/CPI-C API

While designing our API, our primary goal was to mimic the C version of the CPI-C API and thus leverage C programmers' learning investment. To a great extent, we've accomplished this goal. To current CPI-C programmers, the calls to the CPI-C library will look extremely familiar.

Preserving the look of the actual calls requires some additional work by the programmer, such as creating objects to hold some of the output parameters. This is necessary to resolve an unavoidable conflict: Java uses pass-by-value function calls; CPI-C uses pass-by-reference [4]. Since Java is an object-oriented language, we feel that it is natural to create objects to hold CPI-C parameters, although we do acknowledge that it creates a small amount of extra work for the programmer.

Classic Transaction #1: The Pipe

To introduce the API, we'll start by examining the first classic transaction: the Pipe. We present the example by interleaving code with commentary.

We based our transaction code on that given in *CPI-C Programming in C*, an excellent reference written by John Q. Walker and Peter J. Schwaller. For further information on CPI-C, we recommend the book highly.

First, we import the CPIC package, our Java/CPI-C bindings. A CPIC object is the interface to our LU6.2 code. It contains many constants defined in CPI-C, such as the

length of a conversation ID, along with methods that are passed through to the native CPI-C calls.

The programmer need only declare one CPIC object per class. Java will load the dynamic link library (DLL) containing the native methods when that CPIC object is instantiated.

```
/*-----
 * Pipeline transaction, client side.
 *-----*/
import com.ibm.cali.net.CPIC.*;
public class Pipe extends Object {
    public static void main(String args[]) {

        // Make a CPIC object
        CPIC cpic_obj = new CPIC();
```

Each type of parameter has its own class, and each of these classes has associated constants defined as class variables. For example, the CPICReturnCode class has the success return code, CM_OK, defined in it.

There were two major reasons for having a class for each type of parameter. Most importantly, because Java passes all parameters by value, there was no way to return data in simple types, such as integer. If we pass an object as a parameter to a method, the method can set a variable in that object, thus returning data to the caller. The second reason to use objects is to encapsulate constants within the objects that understand those constants--a standard information-hiding technique.

```
// Return Code
CPICReturnCode cpic_return_code =
    new CPICReturnCode(CPICReturnCode.CM_OK);

// Request to send received?
CPICControlInformationReceived rts_received =
    new CPICControlInformationReceived(
        CPICControlInformationReceived.CM_NO_CONTROL_INFO_RECEIVED);
```

The CPI-C send function expects a C-language buffer--that is, allocated space of no specific type. Unlike C, Java has no facility to allocate untyped memory; besides primitives, everything in Java is an object. So whatever the program wants to send must be converted from its object type into an C-style array of bytes. In fact, for each call, the CPIC library must convert from the Java types to a C type.

Java does provide methods that facilitate such conversions. For example, Java can convert a String into a Java array of bytes. While an array of bytes is an object in Java, Java allows you to extract the data--the bytes--from an array of bytes with a native method.

So the programmer must convert his application data into an array of bytes; our code does not do automatic marshalling. While the conversion is not onerous, as we demonstrate below, we are considering augmenting our code to provide this function.

```
// String to Send
String sendThis = "Test of the PipeLine Transaction";

// Length of String to send
CPICLength send_length = new CPICLength(sendThis.length());

// Convert String to send to a Java array of bytes
byte[] stringBytes = new byte[ send_length.intValue()];
sendThis.getBytes(0,send_length.intValue(),stringBytes,0);
```

Like buffer processing, the CPI-C native calls expect symbolic destination names to be C-strings, not Java Strings. Our tool kit automatically converts them from Java Strings to a C-strings as necessary. In general, automatic conversion is possible when the tool kit expects a specific Java type.

The conversation ID is a Java array of bytes that is converted automatically by our tool kit to a C array of bytes--that is, a simple block of bytes.

```
// this hardcoded sym_dest_name must
// be 8 chars long & blank padded
String sym_dest_name = "PIPE  ";

// Space to hold a conversation ID
// (which is just a bunch of bytes)
byte[] conversation_ID = new byte[CPIC.CM_CID_SIZE];
```

Now the programmer is ready to make the CPIC calls almost exactly as he would in C. The only noteworthy differences are that he prefaces every CPIC call with the name of the CPIC object and that none of the parameters are prefixed by the pass by reference (&) symbol.

```
//
// Initialize CPI-C
//
cpic_obj.cminit(      /* Initialize_Conversation */
    conversation_ID, /* O: returned conversation ID */
    sym_dest_name,   /* I: symbolic destination name */
    cpic_return_code); /* O: return code from this call */
//
// ALLOCATE
//
cpic_obj.cmallc(      /* Allocate Conversation */
    conversation_ID, /* I: conversation ID */
    cpic_return_code); /* O: return code from this call */
//
// SEND
//
cpic_obj.cmsend(      /* Send_Data */
    conversation_ID, /* I: conversation ID */
    stringBytes,     /* I: send this buffer */
    send_length,     /* I: length to send */
    rts_received,    /* O: was RTS received? */
    cpic_return_code); /* O: return code from this call */
//
// DEALLOCATE
//
cpic_obj.cmdeal(      /* Deallocate */
    conversation_ID, /* I: conversation ID */
    cpic_return_code); /* O: return code from this call */
} // end main method
} // end the class
```

That's the end of the client, so we'll turn our attention to the server. The server initializes itself, accepts a conversation, receives some data, and prints some diagnostic information. The code is similar to the client code, so we'll omit discussion of some of the redundant details.

As in the client, we instantiate classes to hold the CPI-C parameters, many of which have only an integer as instance data; recall that by using objects, we can mimic call by reference. We also allocate a byte array to hold the received data.

```
/*-----
 * Pipeline transaction, server side.
 *-----*/
import com.ibm.cali.net.CPIC.*;
import java.io.IOException;
```

```

public class PipeServer extends Object {
    public static void main(String args[]) {

        CPIC cpic_obj = new CPIC();

        // Space to hold the received data
        byte[] data_buffer;
        data_buffer = new byte[101];

        CPICLength requested_length = new CPICLength(101);
        CPICDataReceivedType data_received =
            new CPICDataReceivedType(0);
        CPICLength received_length = new CPICLength(0);
        CPICStatusReceived status_received =
            new CPICStatusReceived(0);
        CPICControllInformationReceived rts_received =
            new CPICControllInformationReceived(0);
        CPICReturnCode cpic_return_code =
            new CPICReturnCode(0);

        // Space to hold a conversation ID -- a bunch of bytes
        byte[] conversation_ID;
        conversation_ID = new byte[cpic_obj.CM_CID_SIZE];
    }
}

```

The CPI-C receive call returns a Java array of bytes, while the Pipe transaction expects a String. The programmer can translate the array of bytes into a String by using the String class-constructor that takes an array of bytes as an argument.

```

//
// ACCEPT
//
cpic_obj.cmaccp(    /* Accept_Conversation */
    conversation_ID, /* O: returned conversation ID */
    cpic_return_code); /* O: return code */
//
// RECEIVE
//
cpic_obj.cmrcv(    /* Receive */
    conversation_ID, /* I: conversation ID */
    data_buffer,    /* I: where to put received data */
    requested_length, /* I: maximum length to receive */
    data_received,   /* O: data complete or not? */
    received_length, /* O: length of received data */
    status_received, /* O: has status changed? */
    rts_received,    /* O: was RTS received? */
    cpic_return_code); /* O: return code from this call */
//
// Do some return code processing
//
System.out.println(" Data from Receive:");
System.out.println("  cpic_return_code    = " +
    cpic_return_code.intValue());
System.out.println("  cpic_data_received   = " +
    data_received.intValue());
System.out.println("  cpic_received_length = " +
    received_length.intValue());
System.out.println("  cpic_rts_received    = " +
    rts_received.intValue());
System.out.println("  cpic_status_received = " +
    status_received.intValue());
// Create a Java String from the array of bytes that you received
// and print it out.
String receivedString = new String(data_buffer,0);
System.out.println(
    "  Received string          = "
    + receivedString );

//
// BLOCK so that the Server Window doesn't disappear
//
try{
    System.out.println("Press any key to continue");
    System.in.read();
}

```

```
    }  
    catch  
    (IOException e){ e.printStackTrace(); }  
    }  
}
```

Classic Transaction #5: The Database Update

Now that we've looked at the Pipe, the simplest of the classic transactions, we will look at the most complex transaction, Update, and study the differences.

In the Update transaction, a client requests information from a server, modifies the information, and returns it to the server. It demonstrates a "conversational transaction"--that is, it includes multiple sends and receives within a single conversation. Because the code follows directly from our prior discussion, we omit a detailed explanation of the transaction; the code is included in the appendix.

More interesting than the transaction itself is the use of an extended CPIC class. The update transaction requires two actions that are among the "building blocks" described in Walker and Schwaller's book. They place these actions in C functions (DOCPIC.C in the book) that are easily reused.

In JAVA we can take advantage of inheritance and can extend the CPIC class to include these new actions. We created a CPICClassic5 class that extends (Java's term for inheritance) the CPIC class and added the new actions to the CPICClassic5 class.

When a user instantiates a CPICClassic5 object (rather than a CPIC object) the user now gets all of the CPIC methods, augmented by the code developed for this transaction. The user of the CPICClassic5 object cannot distinguish between the native CPIC methods and the added function.

This example illustrates an important point: Because the CPIC tool kit is object-oriented, users can customize and extend it to fit their needs, seamlessly extending the base function. This opportunity does not exist in C.

Native Calls

Note that a CPIC user can't tell whether the package is included in the standard Java distribution or calls an add-on DLL. In fact, what is and isn't included in standard Java depends largely on what Sun chooses to include. When standard function is lacking, programmers can compensate by implementing libraries that call native methods. Of course, it then becomes their responsibility to provide cross-platform support--something lacking in the prototype.

Diversion: Native Calls and Applets

There is one drawback to implementing native methods: They can't be called from applets. If the reader were to download our library, write an applet that used it, and then try to run the applet on (for example) a Netscape browser, he would trigger a security exception.

The reason is simple: Applets are not permitted to make native calls. ("delete *.*" is a native call--do you want arbitrary applets issuing that?) However, in an intranet, certain applets will be trusted, at least sufficiently trusted to call another networking protocol. We expect that browser manufacturers will implement more flexible applet-security

policies. Until then, SNA applications will have to be run as applications.

Performance

Because Java is an interpreted language, many people are concerned about its performance. We conducted an experiment to determine Java's impact on network performance.

For our benchmark, we used the Update transaction. Update requires three network flows: a "request" from the client to the server; a "reply" from the server to the client; and an "update" from the client to the server. In our experiment, each of the flows contained the same number of bytes.

Our server was a 150 MHz Pentium Pro (tm) computer with 32MB of RAM running OS/2 Warp and CommServer for OS/2 with the Access Feature. Our client was also 150 MHz Pentium Pro (tm) computer running OS/2 Warp and CommServer for OS/2 with the Access Feature; the client had 64MB of RAM. The test was conducted on an isolated 16Mbit/sec token ring.

We wrote the Update client in both C and in Java; the server was written in C and was pre-loaded. We measured only the time to execute a 2,000 iteration transaction loop; the start-up time for the Java interpreter was not included. We ran each client five times, for a total of 10,000 iterations. The standard deviation among the five trials was less than 2%.

The times per transaction are given below:

Bytes/Send	C client (msec)	Java client (msec)	Difference (msec)
256	10.9	16.2	5.3
1024	12.4	17.5	5.1
4096	15.7	20.9	5.2
16384	36.4	41.5	5.1

Switching from C to Java results in a small, constant performance loss of about 5.2 msec per transaction on the client. Since the amount of Java code executed--the only difference between the two clients--does not depend on the number of bytes sent, the constant variation is unsurprising. Since the difference is so slight, it will be unnoticed during interactive work.

It is important to note that there is no performance impact on the server. The server is written in C and is identical for both clients. Thus, **implementing the client in Java does not reduce the number of server transactions.**

CPIC Calls That Have Been Implemented

The following CPIC calls have been implemented to date. This includes all those calls necessary to implement the Classic 5 Transactions.

- cmaccp Accept Conversation
- cmallc Allocate Conversation
- cmcfm Confirm
- cmcfmd Confirmed
- cmdeal Deallocate Conversation
- cmecs Extract Conversation State
- cminit Initialize Conversation
- cmrcv Receive
- cmsdt Set Deallocate Type
- cmsend Send
- cmserr Send Error
- cmsmn Set Mode Name
- cmspln Set Partner LU Name
- cmssl Set Sync Level
- cmsst Set Send Type
- cmstpn Set TP Name

(Walker & Schwaller's book contains a complete list of the CPI-C calls.)

Conclusions

As we demonstrated above, it's a fairly straightforward design exercise to create a Java-callable CPI-C library. The biggest problem--the conflicting method-calling conventions--is solved by passing objects.

More importantly, we have shown that Java needn't be considered an IP-only tool. Enterprises use SNA and will continue to demand its reliability and stability. We've shown that Intranet technologies such as Java do not exclude SNA networks. We expect enterprises to benefit from the combination of Java's strengths (such as platform-independence and dynamic download) and SNA's mission-critical reliability.

Footnotes

[1] 1/3 cup butter, 2T water, 2T vinegar, 1T worchestershire sauce, 1t sugar, 1t onion

salt, 1/2t garlic powder, 1/2t pepper, dash ground red pepper--Betty Crocker's Cookbook, 1989, Western Publishing Company, Inc.
[Return](#) to text.

[2] Using [AnyNet's](#) sockets over SNA function, Web browsers can run over SNA.
[Return](#) to text.

[3] Lap Huynh, "Performance comparison between TCP slow-start and a new adaptive rate-based congestion avoidance scheme," Proceedings of the IEEE International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 1994.
[Return](#) to text.

[4] CPI-C was designed from the beginning to be cross-language-enabled. Clearly, the Java architects did not interact with the CPI-C architects.
[Return](#) to text.

JavaTM is a trademark of Sun Microsystems, Inc.

Microsoft, Windows and Windows NT are registered trademarks of Microsoft Corporation.

Other companies, products, and service names may be trademarks or service marks of others.

[Copyright](#) [Trademark](#)

[Java Feature Page](#) [Java Home](#)

[▶ IBM HOME](#)

[▶ ORDER](#)

[▶ EMPLOYMENT](#)

[▶ CONTACT IBM](#)

[▶ LEGAL](#)