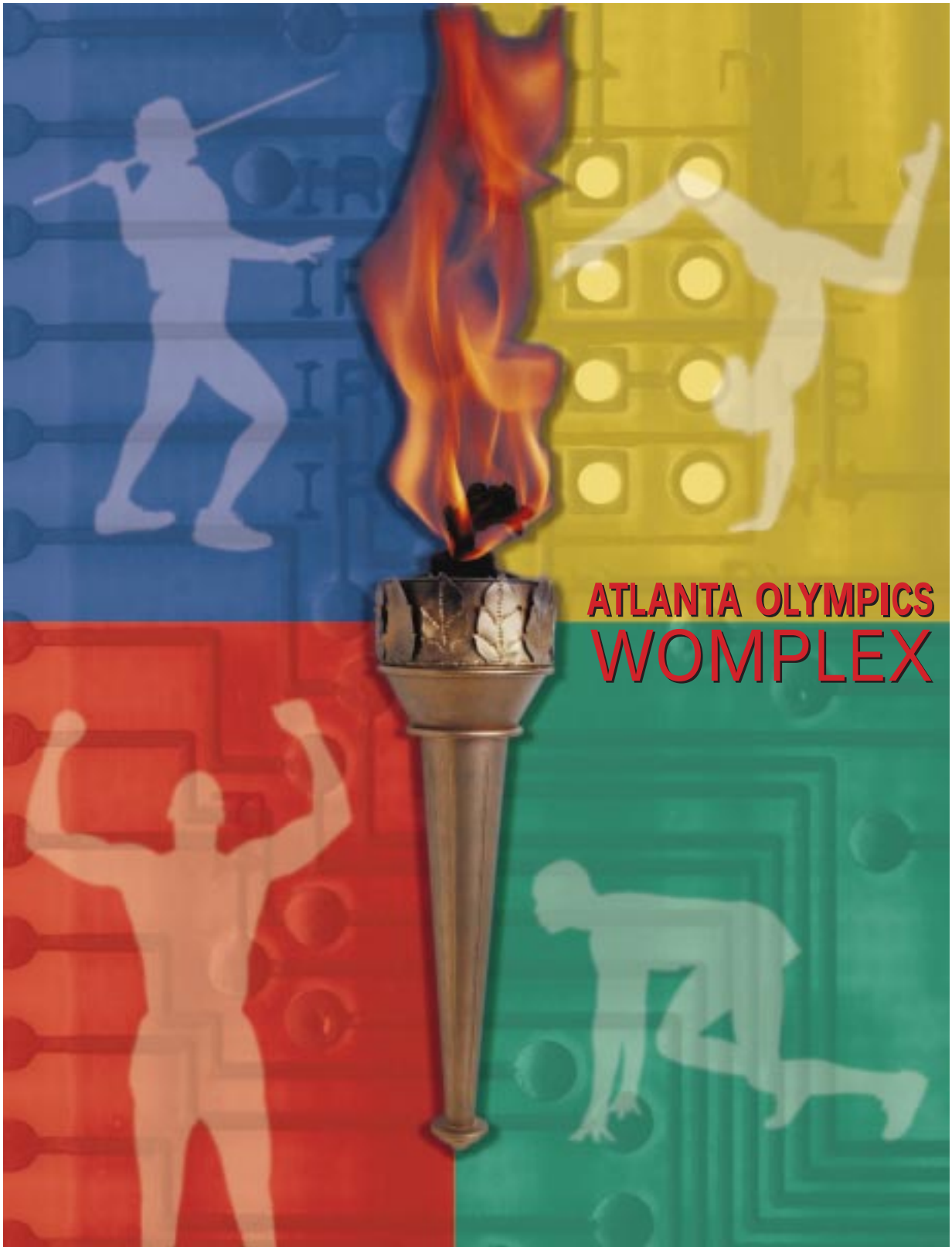
IBM's Magazine for AIX Developers March 1997

AIXpert

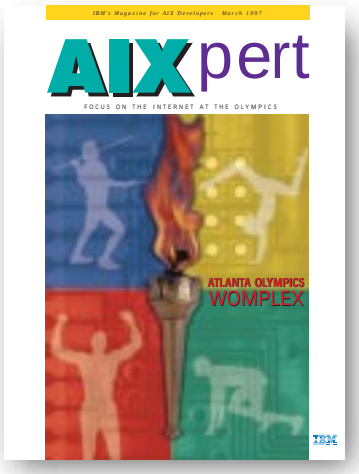
FOCUS ON THE INTERNET AT THE OLYMPICS



IBM

AIXpert DIGITAL VERSION IBM

TABLE OF CONTENTS



MARCH
1997

AIXpert BACK TO MAIN MENU

AIXpert Goes Electronic

Welcome to the new Electronic Edition of *AIXpert* Magazine! You can do online searches in each issue, or use the cumulative database to search across all the issues on the CD-ROM. When you find an article that you must take with you, you can print it out on a PostScript® printer and keep all the page formatting and illustrations—just as if it were a printed magazine (for those of you who just can't be weened). Or check out the Acrobat® documentation on the CD-ROM—it shows you how to change the whole issue's PDF file into a PostScript file and you can then print out the entire magazine (usually around 72 pages). So in this issue you have all the advantages of the online medium with the capabilities of the (old) hard copy magazine.

Plus, this CD-ROM is truly disposable! No more clinging on to old issues of *AIXpert* because you might need the information "someday." No more having to justify the stack of old magazines to your significant other. Each new edition of the *AIXpert* CD-ROM will include all of the previous editions (back to May of 1994—collectors may want to hang on to older issues) plus the most recent issue. This will be true as long as we have room on the CD-ROM, which at less than 5 megabytes per issue will last for quite some time. So, you can truly throw out that old CD-ROM when the new issue comes, but in the interest of the planet you may want to recycle it by giving the old CD-ROM to a less fortunate colleague.

If you are going to keep an issue, though, this one is worth keeping. We cover the Web from many angles. Our keynote article tells you how those mystical wizards kept the 1996 Summer Olympic Web site running and up to date during the busiest 10 days of the summer. You can also learn about delivering rich multimedia content over the Web using VideoCharger for AIX. Plus, we have two flavors of Java: Java threads and accessing legacy applications from Java. In a different vein, learn about using the programming interfaces in the Common Desktop Environment (CDE), and explore message passing while writing parallel programs for the RS/6000 SP.

We also have two new additions to this issue. The Book Reviews section debuts with a new AIX book for system administrators and users written by a trio of AIX developers. Check it out! The Events section gives you information about upcoming conferences and events of interest to AIX developers. So that's it for a full and interesting issue. Oh, and for those of you who have been tracking the subject of my previous two editorials, suffice it to say that, "it's nice to be needed."

George Noren

***George Noren**, IBM Corporation, Internal Zip 1034, 11400 Burnet Road, Austin, TX 78758. Internet: geo@austin.ibm.com. Since joining IBM in September 1979, Mr. Noren has written hardware and software manuals, including AIX and RS/6000 manuals, and was a member of the InfoExplorer™ design team. He has worked as a system administrator for several AIX systems and is a Certified AIX System Administrator. He is currently Editor in Chief of the World Wide Web site for IBM's Solution Developer Program (www.developer.ibm.com) in addition to his work with* AIXpert *Magazine. Mr. Noren studied engineering at Illinois Institute of Technology, holds a BA in English from the University of Minnesota and an MBA from St. Edwards University in Austin.*

George Noren

Legacy Application Access Through Java

By Jeff Jilg and John Dodson

Java provides a mechanism for tying existing applications to platform-neutral Java interfaces. Java applets and applications can also be extended to take advantage of platform-specific (native) features. This article explores the native method interface deployed in Java 1.0 for connecting Java to platform-specific programs. The proposed native method interface in Java 1.1 is also discussed in this context.

Since Java™ is now virtually pervasive on all operating systems, it is quickly becoming a *de facto* programming and runtime environment. The Java model provides near platform independence through the Java programming language. Programs written in Java can be compiled on one platform and executed on any computer that has a Java runtime environment. (For a review of Java on AIX®, see *"Java on AIX—A Strong Brew"* in the November 1996 issue of AIXpert). Figure 1 shows the different types of Java applications that will be discussed in this overview. The focus of this article is on scenario D, connecting Java to legacy applications.

Java Applications

There are a variety of ways to program in Java or access Java functions. It is useful to characterize the different methods that span the spectrum from pure (traditional) Java applets to non-Java applications, which interact with Java programs.

Scenario A: Many Java programs are classified as applets since they run inside a Java-enabled Web browser. A well-known example of this scenario is developing an applet on one platform, such as AIX, and distributing the applet through a Web server to an arbitrary client on the Internet (for example, Windows® 95 running Netscape™).

Scenario B: Many applets already on the Internet demonstrate this example. While this provides a useful outlet for small, independent applets, few full-featured applications currently use this model. Corel's® Java-enabled office suite is just now emerging as one of the first major application sets to exploit Java in this way. Lotus® has publicly announced an upcoming port of SuiteSpot™, which will be enabled through Java. The number of these full-featured applets and application sets is expected to increase dramatically through 1997 and 1998 as Java matures.

Scenario C: Another avenue from which Java applications are occurring is through Web server side Java applets. These Java applets result from a Web client request that is satisfied on the Web server (through Java). An example of this model is a Web client requesting data on the Web server that must be processed through some server side logic. For example, the Web client may request a list of current office supply inventory from the Web server used by an office supply warehouse. In this case, the Web server must have a mechanism for obtaining and formatting the information before forwarding the answer back to the Web client. Different ways of providing this server side logic include Common Gateway Interface (CGI), JavaScript™, and Java applets. The benefit of Java applets in this example is their platform independence.

Scenario D: Corporations and small companies have a huge investment in existing legacy applications. Rewriting these applications into Java just for the sake of Java's benefits is far too

Jeff Jilg

John Dodson

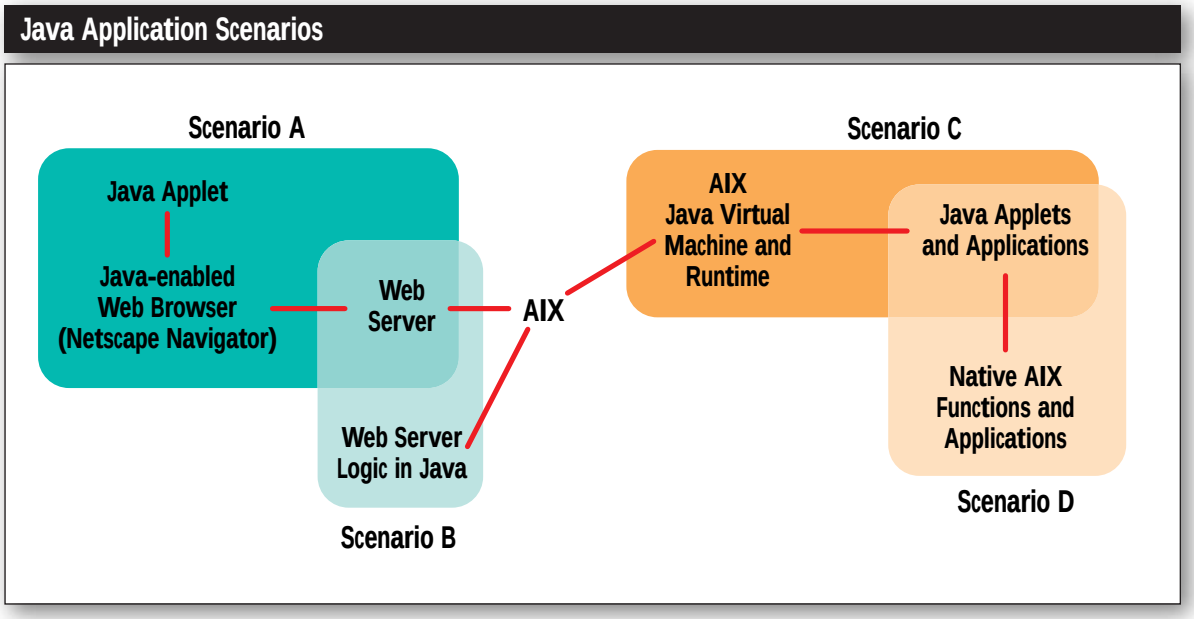


Figure 1. Scenarios of Java applications

expensive. Some applications are being connected through Web servers on intranets and the Internet using the scenarios described above. Another direct mechanism for connecting legacy programs to Java and Java-enabled browsers is through the Java native methods interface. This mechanism is already being adopted across a variety of industries. Native methods will continue to be deployed in the near term rather than the Java application types discussed above. That is because native methods provide a means of accessing legacy programs through Java. Java native method interfaces also can be used to extend the functionality of existing programs through Java applets and applications.

Java native method interfaces can be used in any of the four scenarios described above. This interface allows any Java program to execute a platform-specific binary, such as an AIX program. Thus, a native method can be accessed from the Java applet in scenario A in similar fashion to that shown in scenario D.

The remainder of this article will explore Java native method interfaces in more detail. The next section will describe some advantages, disadvantages, and features provided by this mechanism. The following section shows how native method interfaces can be deployed using the existing Java 1.0 on AIX 4.1 and 4.2. It then examines the similarities and differences between the existing methodology and the new methodology in Java 1.1. The article concludes with a discussion on interoperability and the projected future of Java interfacing.

Features of the Native Method Interface

Contrary to the easy Java applet Web distribution model, native method interfaces have some disadvantages, including the following:

- A native method interface creates instant platform dependence since the corresponding C or C++ program must be recompiled or even modified for each execution platform.
- Java security restrictions are essentially bypassed in the native method since C programs inherit the security properties associated with the user who executes the Java program.
- Coding linkage between Java and C is cumbersome and must be rebuilt. It is then redistributed to all machines that use it (just like current C programs) each time a change occurs to the native method code.

These disadvantages focus on the fact that pure Java programs have distinct advantages that are well-known. However, several key advantages to writing native methods include the following:

- The most important advantage is that native methods allow Java programs to access functions (and whole programs) in the vast array

of existing applications that already support most businesses today.

- Another advantage is the extended functionality provided by native method access to the system. While this is also listed as a disadvantage, the tight security restrictions in Java are sometimes overbearing. For example, the current Java implementation with Netscape Navigator™ Version 3.0 does not allow applets to read or write files on the client. Most production programs must be able to retrieve or store information at some point, so native methods can be used to supplement functionality that is not available in widely used Java applet runtime models.
- Speed and runtime efficiency are critical to the end user perception of all applications. Java on AIX has been supplemented with a Just-In-Time (JIT) compiler that improves runtime performance of Java programs. Mission-critical applications can take advantage of native AIX and other platform Application Programming Interfaces (APIs) to enhance runtime execution on a function-by-function basis from Java using native method interfaces.

The advantages provide insight into how to extend your Java program. Native methods work in both the Java applet and application models. Much of the Java programming over the next two years will use native methods to access existing C-coded function. Java on AIX program execution speed should not be a big issue with the current and future releases of the JIT compiler.

Steps for Native Method Interface on AIX

The steps to implementing a Java native method are straightforward, but must be followed closely for the native method to work properly. The tools required to produce working native methods come standard with the Java Development Kit (JDK) that shipped with AIX 4.2. Access to a C compiler (not part of the JDK) and a linker (provided with AIX) is also required. The following outline can be followed to create native methods.

Steps for Implementing a Java Native Method

Java native methods require six steps to implement.

Step 1. Write the Java code. Subtasks for writing the code include the following:

- Create a Java class that defines the native method. This class should also load the shared library containing the native method implementation.
- Create a Java driver class that will invoke the native method.

Step 2. Compile the Java code using javac. The standard Java compiler, javac, is provided with the JDK. This will create a bytecode class file with the same name as the Java class. Java source file `linkCheck.java`, for example, would result in `linkCheck.class` upon compilation.

Step 3. Create a header file using javah. Also a standard tool, javah is included in the JDK. The usage is `javah <CLASS NAME>`. Note that the actual class name is used, not the class file name. This will create a C header file named `<CLASS>.h`, where `<CLASS>` is the name of the Java class containing the native method definition.

Step 4. Create a C stubs file using javah. Later, the resulting C file will be compiled and linked into a shared library.

Step 5. Write the actual native method. Subtasks for this step include the following:

- The C function signature for the native method including name, parameters, and return type must exactly match the one in the header file created in step 3 above.
- Write the actual native method code as you see fit.

Step 6. Create the AIX shared library. Subtasks include the following:

- The native method must reside in a shared library. Creating this library is platform specific.
- Compile the C code.
- Create the exports file.
- Build the library.

AIX Hints for the Exports File

The steps for implementing native methods are universal. However, there are some AIX-specific items to be aware of. These are detailed in this "hints" section.

Hint 1: The exports file must contain two entries for each native method. The actual function name of the native method from the header file (generated in step 3) and the stub name from the C file (generated in step 4) must

Native methods allow Java programs to access functions (and whole programs) in the vast array of existing applications that already support most businesses today.

be listed on lines by themselves. Note that the exports file requires only the symbol name for the function and does not include function parameters.

Hint 2: The first line in the exports file *must* contain `#!` (on a line by itself).

AIX Hints for Building the Shared Library

The shared library is created using the C compiler and linker. Several AIX-specific items are important to mention:

Hint 1: The library name must be of the form `lib<NAME>.so`, where `<NAME>` is the library name referenced in the Java source file (from step 1) that invokes `System.loadLibrary`. An example library name is `liblinkExample.so`.

Hint 2: Since a shared reentrant load module is required, pass the following flags to the AIX linker: `-bM:SRE -bnoentry`

Hint 3: The Java exports file must also be passed to the linker:

```
-bI:$(JAVA_HOME)/include/java.exp
```

Hint 4: Several libraries must also be linked:

```
-L/usr/lib/threads
-lm -lpthreads -lc_r /usr/lib/libc.a
```

Hint 5: When running the Java program, make sure the `LD_LIBRARY_PATH` (not `LIBPATH`) includes the directory containing the newly created shared library. Example:

```
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:new_path
```

will add `new_path` to the existing environment variable.

To run the Java application, first ensure the `LD_LIBRARY_PATH` is set to include the shared library created in step 5 above. Then, simply type:

```
java <CLASS>
```

where `<CLASS>` is the name of the driver class created in the second bullet of step 1 above. If the native method is called from a Java applet (instead of an application), then the applet would be referenced from a Web page and loaded through a call to the Web page from appletviewer or a Java-enabled browser such as Netscape Navigator.

The steps to implementing a Java native method are straightforward, but must be followed closely for the native method to work properly.

Java Native Method Example

The following example illustrates an AIX-specific function that is not available in Java. In a similar fashion, native methods can be used as interfaces to existing C applications.

Suppose that you want to determine if a file is a symbolic link. Java has no built-in methods to perform this AIX-oriented function. One solution would be to write a native method in C and invoke that native method from within your Java code. The following example shows how to solve this unique problem using the six steps discussed above.

Step 1: Write the Java code. Figure 1 shows the `linkCheck.java` file. Figure 2 shows the `driver.java` file. Note that this Java code defines a native method interface called `NMisLink` that requires one String parameter, a filename, and returns an int. Also note that a shared library called `linkExample`, which maps to `liblinkExample.so` on AIX, is loaded. Finally, the Java method `isLink` invokes the actual C function. Other Java code can access the native method only through the `isLink` Java method.

Step 2: Compile the Java code. `javac linkCheck.java` creates `linkCheck.class`. `javac driver.java` creates `driver.class`.

Step 3: Create a header file using javah. `javah linkCheck` creates `linkCheck.h`.

Step 4: Create a C stubs file using javah. `javah -stubs linkCheck` creates `linkCheck.c`.

Step 5: Write the actual native method. Figure 3 shows the `NMisLink.c` file. Note that the C function signature exactly matches the prototype in the header file created in step 3.

Step 6: Create the AIX shared library. Three tasks in this step include the following:

- Compile the C code.

```
cc -c NMisLink.c linkCheck.c -I
$JAVA_HOME/include -I
$JAVA_HOME/include/aix_pt
```

- Create the exports file. Figure 4 shows the `linkExample.exp` file.

```
import  java.lang.*;
 class linkCheck
 {
    private native int NMisLink (String fileName);
    static
    {
       System.loadLibrary (“linkExample”);
    }
    public boolean isLink (String fileName)
    {
       if (NMisLink (fileName) == 1) //Call to the native method
          return true;
       else
          return false;
    }
 }
```

Figure 1. linkCheck.java

```
//a “driver” to test calling code implemented on a native method

class driver
{
   public static void main (String [] args)
   {
      linkCheck lc = new linkCheck(); //arg[0] is expected to be a pathname to check

      if (lc.isLink(args[0]) == true)
         System.out.println (“File “ + args[0] + “ IS a link.”);
      else
         System.out.println (“File “ + args[0] + “ is NOT a link.”);
   }
}
```

Figure 2. driver.java

```
#include <stdio.h>
#include <StubPreamble.h>
#include “linkCheck.h”
long linkCheck_NMisLink (struct HlinkCheck *this,
                         struct Hjava_lang_String *javaFileName)
{
   char c_string[PATH_MAX+1];
   struct stat stat_buf;
   int rc;
   javaString2CString(javaFileName, c_string, PATH_MAX);
   rc = lstat (c_string, &stat_buf);
   if (rc == 0 && (stat_buf.st_mode & S_IFMT) == S_IFLNK)
      return 1;     /* IS a link  */
   else
      return 0;     /* NOT a link */
}
```

Figure 3. NMisLink.c

```
#!
linkCheck_NMisLink
Java_linkCheck_NMisLink_stub
```

Figure 4. linkExample.exp

```
ld -o liblinkExample.so NMisLink.o
   -bM:SRE -bnoentry
   -bI:$JAVA_HOME/include/java.exp
   -L /usr/lib/threads
   -lm -lpthreads -lc_r /usr/lib/libc.a
   -bE:linkExample.exp
```

Figure 5. LiblinkExample.so

- Build the shared library using the example in Figure 5. This will create `liblinkExample.so`.

Now, `LD_LIBRARY_PATH` is exported to include the current directory. The program can be run by `Java driver /etc/init`, which shows the following:

```
File /etc/init IS a link.
```

This example is invoked as a Java application. The code for a Java applet would be similar, but the Java code would not have a main method. Instead, `driver.java` code would be encapsulated in the driver class without main wrapper code.

The example shows good programming practice in the Java method call since the native method (in C) was accessed through a Java method that encapsulated the native method. The private method type was used to insulate the implementation from the end user (programmer), similar to standard object-oriented programming.

Java 1.1 JNI and Future Interoperability

Java 1.1 is in a beta state as this article is being published. A new interface has been proposed and implemented in Java 1.1. The new interface, Java Native Interface (JNI), is designed to satisfy the same requirements as the existing interface. JavaSoft™ considered several interfaces in order to design a best-of-breed consolidated interface. Namely, the existing designs for the following were integrated into JNI:

- Java native method interface from Sun®
- Java Runtime Interface (JRI) from Netscape
- Raw Native Interface (RNI) from Microsoft®

A variety of reasons account for the changes, including efficiency, portability, COM interoperability, and functionality. The new design allows more flexibility. It also shows the openness of JavaSoft to consider current limitations and other models with the goal of delivering technology that can grow over time.

The new Java 1.1 implementation will still execute existing native method interfaces (created under the Java 1.0 specification). When Java 1.1 becomes formalized, Sun is recommending that new interfaces be written using the new specification. The draft specification can be found on the JavaSoft home page under developer news under Java 1.1. It is more complex than the current mechanism, but this complexity also allows for more flexibility.

This article outlined common Java programming scenarios. The most prevalent model is connecting existing C (or C++) applications to Java applets and applications. Java native method interfaces can be used to accomplish this scenario. Required steps for programming interfaces were demonstrated through a simple example.

A new Java JNI method has been introduced for Java 1.1. This evolution of Java to incorporate new mechanisms is backward compatible and shows both the openness of the technology and the rapid evolution that Java is undergoing. IBM is delivering a robust Java-compatible port of Java JDK on AIX. Java on AIX is being used in a variety of programming contexts, including the scenario outlined in this article.

Jeff Jilg, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Dr. Jilg currently works in AIX System Architecture on leading-edge technology. His recent work areas include systems management, object-oriented tools, and the Internet. His current responsibilities include release architecture, Java on AIX, Internet integration, and the AIX Bonus Pack. His PhD in Computer Science from Texas A&M complements his master's degree in the same field from the University of Texas at El Paso.*

John Dodson, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Dodson is currently working in AIX Development on the Common Desktop Environment (CDE) and various Java development projects. His current assignments and responsibilities include systems management, Java development, providing JDK on AIX, and CDE development and support. He has a BS in Computer Science from Brigham Young University and is currently pursuing an MBA at Saint Edwards University.*

CDE Plug-and-Play

By George Kraft IV

The programming infrastructure, such as ToolTalk, is a major strength of the Common Desktop Environment. This article illustrates client and server plug-and-play through the use of the Desktop's Application Programming Interfaces (APIs).

ToolTalk, in the Common Desktop Environment (CDE), is a message brokering system that enables applications to communicate with each other without having direct knowledge of one another. Client and server applications can be developed independently, mixed and matched, and upgraded separately through plug-and-play. In addition, the Desktop Service can be called to perform methods on file and buffer objects on behalf of ToolTalk.

Figure 1 shows the ToolTalk Service listening for `TtChmod` client requests. ToolTalk Service brokers pattern-matched `Chmod` messages to the registered mock change-mode application server (`ttchmodd`) that is waiting to handle the incoming messages.

ToolTalk brokers the requests from the client to the server application. The Desktop Service can forward CDE object method invocations to the ToolTalk Service. With the Desktop, both C programs and `dtksh` scripts can initiate actions that are transmitted to the ToolTalk Service. Consequently, client invocations from the `dtaction` command line, Application Manager icons, and File Manager icons can be directed through the Desktop Service to ToolTalk application services. Therefore, double clicking on a file icon in the file manager can be plugged into a ToolTalk registered application by first routing through the Desktop action and data-type service.

Ptype

The key to the ToolTalk message brokering system is its ability to define process-type identifiers with specific operations and arguments. In Figure 2, the process-type (`ptype`) `TtChmod` will execute the ToolTalk change-mode daemon application `ttchmodd`. This occurs when the session operation `Chmod` with filename and mode arguments are matched from a request.

Compiling the `ptype` definition with the `tt_type_comp` utility will register services for ToolTalk client applications to call. Consider the `ptype` as a C header file describing an Application Programming Interface (API) and the compiled suite of ToolTalk Services definitions as a library of methods to call. For the list of installed process-type identifiers, try running `tt_type_comp -P` at the command line to dump the database to the screen.

Chmod Service

The file change-mode application (`ttchmod`) described in Figure 3 is simply the Motif® command widget. In Figure 4, the `ttchmodd` server application graphically prompts the user for a command, then calls the command callback `callCB` to execute it; however, for this example, the application just prints the command.

The file change-mode application quickly becomes a plug-and-play service when it registers itself with the ToolTalk Service, then listens for messages to be handled by its `ToolTalkCB` receiver routine.

George Kraft IV

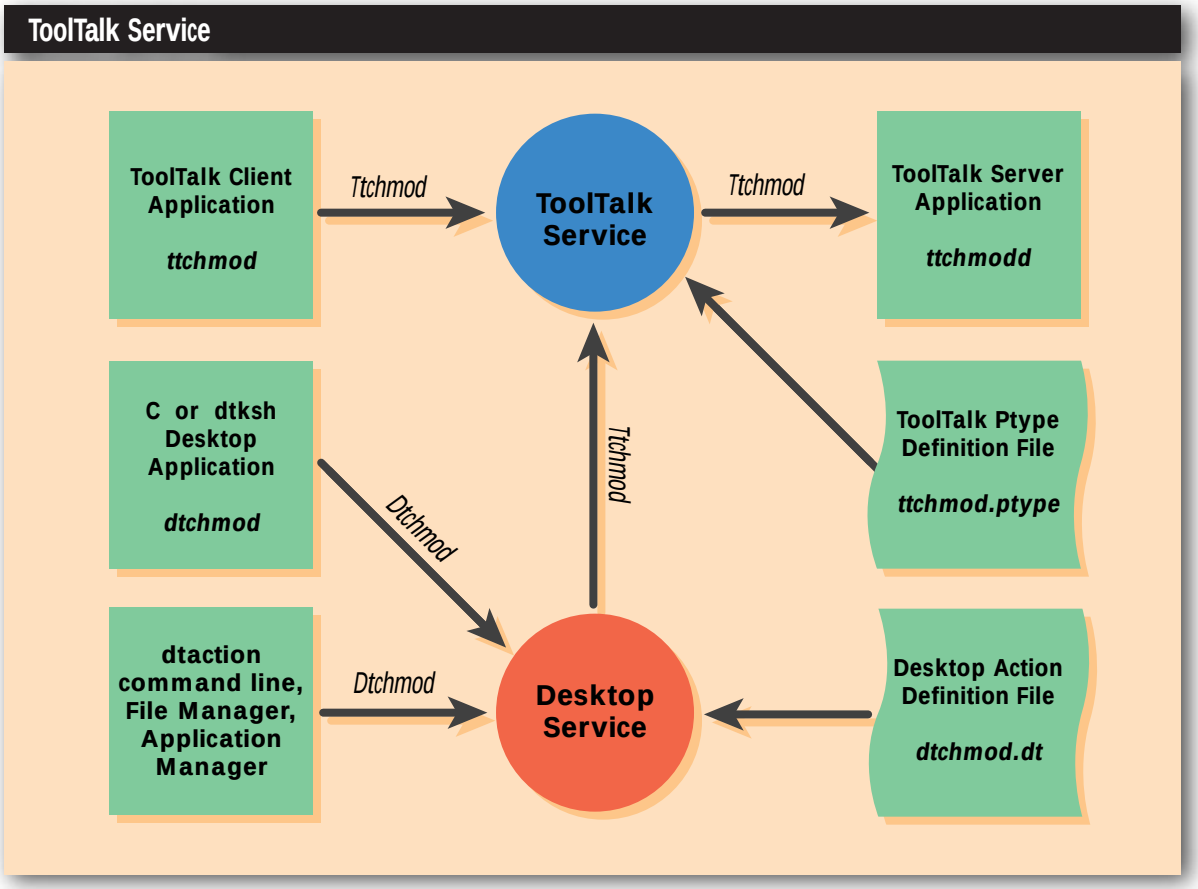


Figure 1. ToolTalk service

```
ptype TtChmod {
  start "/usr/tmp/ttchmodd";
  handle:
    session Chmod(
              in string filename,
              in string mode)
              => start opnum = 1;
};
```

Figure 2. TtChmod ptype definition

Register ToolTalk Service

An application must first locate the ToolTalk session associated with the X display to register itself as a service, as shown in Figure 5. After the application sets its default session to the display's session, the application can initiate itself as a ToolTalk process and obtain a ToolTalk file descriptor. When the `ttchmodd` application gets a handle on the ToolTalk session, then it can register the `TtChmod` process type and join the session to listen for requests.

Handle ToolTalk Requests

ToolTalk sends a message to a registered service; the service listens on its ToolTalk file descriptor for input. When input is observed, the `ToolTalkCB` routine is triggered where the message is read and analyzed. The message's operation is checked, then the arguments are read from the message. ToolTalk messages are similar to a reenterantive version of an ordinary C program's command-line arguments.

```
#include <stdio.h>
#include <Xm/Command.h>

#include <stdio.h>
#include <Tt/tt_c.h>

Widget          topLevel;
XtAppContext    appContext;
Display         *display;

void CmdCB         (Widget, XtPointer, XtPointer);
int  Register      ();
void ToolTalkCB    (Widget, XtPointer, XtPointer);
int  ToolTalkAbort (char *, Tt_status);

void main (argc, argv)
        int     argc;
        char    *argv[];
{
        Widget  cmd;
        int     ttfd;

        topLevel = XtVaAppInitialize(
                &appContext,
                "TtChmodd",
                NULL, 0,
                &argc, argv,
                NULL,
                NULL);

        cmd = XtVaCreateManagedWidget(
                "command",
                xmCommandWidgetClass,
                topLevel,
                NULL);

        XtAddCallback(cmd, XmNcommandEnteredCallback, CmdCB, NULL);

        ttfd = Register();

        XtAppAddInput(appContext, ttfd, (XtPointer)XtInputReadMask,
                ToolTalkCB, cmd);

        XtRealizeWidget(topLevel);

        XtAppMainLoop(appContext);

        tt_close();
}

void CmdCB (w, clientData, callData)
Widget          w;              /*  widget id           */
XtPointer       clientData;     /*  data from application   */
XtPointer       callData;       /*  data from widget class  */
{
        XmCommandCallbackStruct *cbs = (XmCommandCallbackStruct *) callData;
        char *cmd;

        XmStringGetLtoR(cbs->value, XmSTRING_DEFAULT_CHARSET, &cmd);

        printf("CmdCB(): %s\n", cmd);
}
```

Figure 3. Ttchmodd: command widget

The ttchmodd service is no longer needed after it reads the message; so the recipient tells the ToolTalk service to discard the message.

Figure 4. ttchmodd

ToolTalk Client

The ttchmod ToolTalk client is much simpler than the ttchmodd ToolTalk server. The client opens a ToolTalk process and locates the session. The TtChmod process-type message request consists of the Chmod operation with the filename and mode input arguments. It is sent to the ToolTalk Service to be brokered to a registered server application accepting the ptype pattern. However, note that this example omits error checking and garbage collection with tt_mark and tt_release.

Desktop Action

The Desktop Action database in CDE describes methods and objects for applications to act upon. CDE's Desktop Service can describe an action like DtChmod, shown in Figure 8, that can be forwarded through the ToolTalk Service to ttchmodd. If the action does not receive the appropriate arguments, then the Desktop can prompt the user, as shown in Figure 9.

The relationship of the Desktop Action definitions to CDE methods is similar to the

```
Register()
{
        int     ttfd;
        char    *session;
        char    *procid;
        Tt_status ttrc;
        int     ttmark;

        ttmark = tt_mark();

        session = tt_X_session(DisplayString(XtDisplay(topLevel)));
        ttrc = tt_default_session_set(session);
        ToolTalkAbort("TtServer(): tt_default_session_set", ttrc);

        procid = tt_open();
        ToolTalkAbort("TtServer(): tt_open", tt_ptr_error(procid));

        ttfd = tt_fd();
        ToolTalkAbort("TtServer(): tt_fd", tt_int_error(ttfd));

        ttrc = tt_ptype_declare("TtChmod");
        ToolTalkAbort("TtServer(): tt_ptype_declare", ttrc);

        ttrc = tt_session_join(tt_default_session());
        ToolTalkAbort("TtServer(): tt_session_join", ttrc);

        tt_release(ttmark);

        return(ttfd);
}
```

Figure 5. Ttchmodd: ToolTalk registration

```
void ToolTalkCB (w, clientData, callData)
Widget          w;              /*  widget id           */
XtPointer       clientData;     /*  data from application   */
XtPointer       callData;       /*  data from widget class  */
{
        Tt_message incoming;
        Tt_status ttrc;
        int ttmark, args;
        char *filename;
        char *mode;
        char command[BUFSIZ];
        XmString xmcmd;

        ttmark = tt_mark();

        incoming = tt_message_receive();
        ToolTalkAbort("ToolTalkCB(): tt_message_receive",
                tt_ptr_error(incoming));

        if (incoming && (0 == strcmp(tt_message_op(incoming), "Chmod"))) {

                args = tt_message_args_count(incoming);
                ToolTalkAbort("ToolTalkCB(): tt_message_args_count",
                        tt_int_error(args));

                filename = tt_message_arg_val(incoming, 0);
                ToolTalkAbort("ToolTalkCB(): tt_message_arg_val",
                        tt_ptr_error(filename));

                mode = tt_message_arg_val(incoming, 1);
                ToolTalkAbort("ToolTalkCB(): tt_message_arg_val",
                        tt_ptr_error(mode));

                ttrc = tt_message_reply(incoming);
                ToolTalkAbort("ToolTalkCB(): tt_message_destroy", ttrc);

                sprintf(command, "/bin/chmod %s %s\n", mode, filename);
                xmcmd = XmStringCreateLocalized(command);
                XmCommandSetValue(w, xmcmd);
                XmStringFree(xmcmd);
        } else {
                printf("ToolTalkCB(): unknown message %s.\n",
                        tt_message_op(incoming));
        }

        ttrc = tt_message_destroy(incoming);
        ToolTalkAbort("ToolTalkCB(): tt_message_destroy", ttrc);
        tt_release(ttmark);
}

ToolTalkAbort(char *procname, Tt_status errid)
{
        if (tt_is_err(errid)) {
                fprintf(stderr, "%s returned ToolTalk error: %s\n",
                        procname, tt_status_message(errid));
                exit(1);
        }
}
```

Figure 6. Ttchmodd: ToolTalk message receiver

```
#include <stdio.h>
#include <Tt/tt_c.h>

main(int argc, char *argv[])
{
        Tt_message msg;
        Tt_status ttstat;

        if (argc != 3) {
                printf("Usage: %s <mode> <filename>\n", argv[0]);
                exit(1);
        }

        tt_open();
        msg = tt_message_create();
        tt_message_scope_set(msg, TT_SESSION);
        tt_message_session_set(msg, tt_default_session());
        tt_message_class_set(msg, TT_REQUEST);
        tt_message_handler_set(msg, "TtChmod");
        tt_message_address_set(msg, TT_PROCEDURE);
        tt_message_op_set(msg, "Chmod");
        tt_message_arg_add(msg, TT_IN, "string", argv[2]);
        tt_message_arg_add(msg, TT_IN, "string", argv[1]);
        ttstat = tt_message_send(msg);
        if (ttstat != TT_OK) {
                printf("%s\n", tt_status_message((int) ttstat));
                exit(1);
        }
}
```

Figure 7. Ttchmod: ToolTalk client

```
ACTION DtChmod
{
        LABEL           Chmod
        ICON            DtDFile
        TYPE            TT_MSG
        TT_CLASS        TT_REQUEST
        TT_SCOPE        TT_SESSION
        TT_ARG0_MODE    TT_IN
        TT_ARG0_VTYPE   string
        TT_ARG0_VALUE   %Arg_1"File name:"%
        TT_ARG1_MODE    TT_IN
        TT_ARG1_VTYPE   string
        TT_ARG1_VALUE   %Arg_2"Octel mode:"%
        TT_OPERATION    Chmod
}
```

Figure 8. DtChmod CDE action definition

relationship of `ptype` definitions to ToolTalk processes. For an example of Desktop actions and data types, run `dttypes` as the command line to dump the database to the screen.

Desktop Service Client

The APIs of the Desktop Service can invoke actions registered in the Desktop database either from a C program or from a `dtksh` script, as shown in Figure 10. Here, the `dtchmod.ds`, `dtksh` script prompts the user, as shown in Figure 11, with the message dialogue to confirm with the user before requesting changes to the file's mode.

Command Line

In addition to calling Desktop actions from C programs and `dtksh` shell scripts, users can initiate requests from the command line, as shown in Figure 12. If the appropriate arguments are given, then the action is forwarded to ToolTalk; otherwise the user is first queried, as shown in Figure 9.

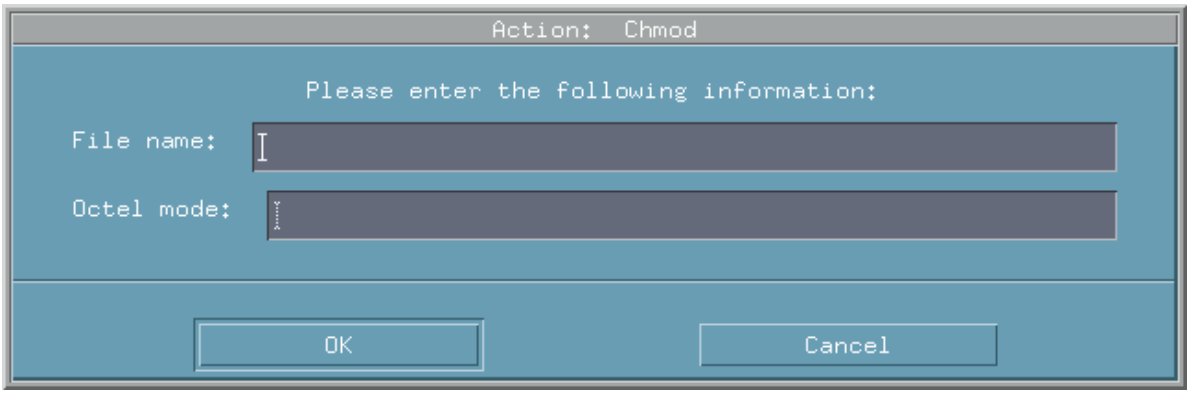


Figure 9. dtaction

```
#!/usr/dt/bin/dtksh

chmodCB()
{
        DtActionInvoke $TOPLEVEL DtChmod '' '' '' True NULL \
                "FILE" "$FILE" \
                "FILE" "$MODE"
}

main()
{
        XtInitialize TOPLEVEL dtChmod DtChmod "$@"

        DtDbLoad

        XmCreateMessageDialog CHMOD $TOPLEVEL motd \
                dialogTitle:"DtChmod" \
                helpLabelString:"Chmod" \
                messageString:"$(print Chmod $FILE to $MODE)"

        XmMessageBoxGetChild OK $CHMOD DIALOG_OK_BUTTON
        XtUnmanageChild $OK

        XtAddCallback $CHMOD helpCallback chmodCB
        XtAddCallback $CHMOD cancelCallback exit

        XtManageChild $CHMOD

        XtMainLoop
}

MODE=$1
FILE=$2

main
```

Figure 10. DtChmod command-line action

Figure 11. dtchmod

Plug, and Play...

We have seen how the `ttchmodd` service registered with ToolTalk can receive messages matching the `TtChmod ptype` pattern from ToolTalk clients, from Desktop clients written in either C or `dtksh`, from the command line, and from double clicking on file object icons. Understanding these examples demonstrates how client and server applications can be developed independently, mixed and matched, and upgraded separately through plug-and-play. A ToolTalk-enabled application service registered with its `ptype` definition can be developed without specific knowledge of its counterpart.

CDE defines a message dictionary of Desktop-specific ToolTalk process types, operations, and

```
dtaction DtChmod /etc/motd 644
```

Figure 12. DtChmod command-line action

arguments as seen from viewing the database. Others, such as Computer-Aided Design (CAD) and Electronic Design Automation (EDA) services have developed supplemental dictionaries. You can use existing `ptypes` or define your own, but the important point to know is how to register the process-type identifier, operation, and arguments.

For more information regarding the Common Desktop Environment, visit IBM's CDE Web page at URL http://www.austin.ibm.com/software/CDE/.

George Kraft IV, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Kraft is an advisory programmer for the RS/6000 Division working in the AIX Desktop group. He is currently working on the Common Desktop Environment Version 2.1 and a version of the Network Computer. Mr. Kraft received a BS in Computer Science and Mathematics from Purdue University.*

Java Threads

By Chary Tamirisa

This article explores thread support provided in the Java language, the Java classes, and the Java Virtual Machine. It describes the Java execution environment with particular reference to the AIX 4 Java Development Kit (JDK) 1.0.2. We assume the reader is familiar with Java and has some knowledge of thread-related concepts.

One important feature of Java™ is its support for multithreaded programming. Part of this support is built into the Java language itself; the rest is provided through a few basic Java classes. The Java Virtual Machine also creates and manages threads to ease program execution. Programming languages such as C and C++ do not have thread support built into the language; in these languages, threads are typically supported through separate runtime libraries. For example, AIX 4 provides a POSIX.1c-based threads package called the *pthreads library* that is typically used by applications written in C and C++.

What is a Thread?

A *thread* is the basic unit of execution. You typically create a separate thread to execute an independent sequence of actions. There are two distinct parts of creating a thread that go hand in hand: the thread itself as an execution unit, and the action it executes (often referenced as the thread function or procedure, or thread body).

The distinction is often blurred between the execution unit and the thread body. For instance, when you create a thread using the pthreads library, you specify action as part of creating the thread. However, it is important to distinguish between these two.

A thread has certain attributes that govern its execution. For example, a thread has a priority that governs its scheduling. Other attributes, such as stack size, may be specified at thread creation time. Threads can be created and terminated as needed. In addition, threads can be temporarily suspended and later resume. When multiple threads concurrently operate on shared data, it becomes important to guard against simultaneous modifications of shared data by synchronizing the execution of multiple contending threads. Mutual exclusion locks among multiple threads typically guarantees this.

Understanding these basic concepts helps in writing well-behaved multithreaded programs.

Java Threads Overview

The following section provides a brief description of the Java Virtual Machine memory model that governs thread execution. It outlines the threads support in the Java language, then summarizes the built-in classes that assist in multithreading programs.

Memory Model

A *variable* is a memory storage location in a Java program. During their execution, threads share class variables, instance variables, as well as components of arrays. A thread may assign a value to a variable or use the value of a variable. When two or more threads act on the same variable concurrently, the variable may have timing-dependent values (this situation is also known as a race condition); therefore, the program can have indeterminate results. A well-behaved multithread program avoids a race condition.

The Java Virtual Machine provides a memory model that allows the fast and efficient implementation of concurrent programs. Briefly, this model provides a master copy of variables shared by all threads in *main memory*, with each thread keeping a local *working copy* of these variables in *working memory*. As threads operate on the

Chary Tamirisa

shared variables, data is transferred between the main memory and working memory.

The Java memory model specifies a set of rules when a thread is allowed or required to transfer data from the working memory to main memory, or vice versa. The main memory also associates each object with a lock. A thread typically obtains a lock before accessing a shared variable. The acquisition of a lock flushes a thread's working memory and guarantees that a thread's working memory is refreshed with the latest values from the main memory. Similarly, when a thread unlocks a lock, it guarantees that the variables in working memory will be written back to the main memory.

Language Features

Java supports thread synchronization by providing the keyword *synchronized*. This keyword can be used to create synchronized methods or to protect a block of code from a concurrent execution. A *block* is a sequence of statements with local variable declarations within braces. The syntax is as follows:

```
synchronized (Expression)Block
```

Java allows threads that access shared variables to keep private working copies of variables for the sake of efficiency.

The Expression must be a reference type (class, interface, or array). The executing thread acquires the lock associated with the Expression; then the Block is executed. If the Block executes normally, the lock is released and the synchronized statement completes normally. If the Block executes abnormally for any reason, the lock is unlocked and the synchronized statement completes abruptly.

Java allows threads that access shared variables to keep private working copies of variables for the sake of efficiency. These working copies are typically synchronized with the shared master copies only when objects are locked or unlocked. As a rule, threads should lock to enforce mutual exclusion before modifying shared variables.

Besides lock support, Java provides the *volatile* modifier for variables. This guarantees that a thread's copy of a variable is reconciled with its master copy every time it accesses the variable. The volatile modifier is useful in the following two cases:

- **Instance variables that are accessed in busy wait loops.** Here, by declaring them as volatile, we force the reloading of values that are typically changed by other threads during the iterations of the loop.
- **Instance variables used in native methods.** In this case, the best you can do is to mark the instance variable as volatile.

Built-in Classes

In addition to these two keywords—synchronized and volatile—in the language, a few basic Java classes support Java's thread creation and management. All classes that support multithreading are part of the `java.lang` package. By default, all Java programs (applications or applets) import this package. The thread-supporting classes are as follows:

- `java.lang.Thread`.This basic class relates to the actual creation and control of a Java thread. It provides basic thread management methods, such as start and stop a thread, suspend and resume, and get and set priorities of a thread.
- `java.lang.ThreadGroup`.This basic class is similar to the one above. It relates to the actual creation and control of a Java thread and provides basic thread management methods. It differs, however, by relating to the whole group of threads, not just to a single thread.
- `java.lang.Runnable`. The interface defines the `run()` method. Any class that implements this interface can provide a thread procedure.
- `java.lang.Object`. The thread-related methods in `java.lang.Object` are synchronization methods such as `wait()`, `notify()`, and `notifyall()`.

Using these Java language features and its basic classes, you can write efficient multithreaded programs. Moreover, the Java Virtual Machine creates threads to support the execution of a program, and these threads are additional to any threads created by a program. Since the actual threads created by the runtime may depend on a particular environment (such as AIX 4 or WIN32), you cannot depend on their specifics when writing Java programs. However, knowing what they do can help in understanding the execution and control flow of a program.

Java Threads Programming Model

Based on functionality, the methods provided in the Java `Thread` class can be categorized as

follows: thread creation, thread groups, thread properties, thread control, thread synchronization, thread scheduling, thread blocking, and asynchronous interruption of threads.

In the following sections, we discuss these and also the Java security model with respect to threads, followed by multithreading aspects of Java applets. Finally we examine thread safety in the Java built-in classes.

Thread Creation

A thread executes a specified action as a separate execution unit. Java provides the basic class `java.lang.Thread`, which you can extend (subclass) to create a thread of execution. You override the `Thread` class' `run()` method to specify the action to be executed by the thread.

Java allows only single class inheritance; it does not allow a class to subclass from multiple classes. So, if your class extends a thread, it cannot subclass from any other classes that you may need; this can be restrictive. For this reason, subclassing from the `Thread` class is not generally the best way to create classes that must be run in separate threads.

Implementing the `java.lang.Runnable` interface is another way to create classes that can be run as threads. A class can subclass from any user-defined class and still implement the `Runnable` interface to run as a separate thread. Implementing the `Runnable` interface in a class does not prevent the class from implementing other interfaces, because Java allows a class to implement multiple interfaces. This overcomes the restriction if you just subclass from the `Thread` class.

The following sections illustrate both ways of creating threads.

Extending the java.Thread class

In this method, your class simply extends the `java.lang.Thread` class and overrides its `run()` method to provide the thread action. Note that you need to invoke the `start()` method of the `Thread` class to run the thread.

In Figure 1, the `Sorter` class subclasses the `Thread` class and invokes the `start()` method in its constructor. The `run()` method specifies the desired sorting action.

An application program may use the sorter as shown in Figure 2.

In Figure 1, the sorter thread is started in the constructor. If you want to better control the starting of the sorter thread, you can do so from an application program that creates the sorter object. Figure 3 shows an example.

An application program can create the sorter object and start the thread as shown in Figure 4.

```
class sorter extends Thread
{

      sorter()
      {
            start(); // run the thread.
      }
      public void run()
      {
      }
}
```

Figure 1. A thread class invoking the start() method

```
class Sort
{
      // Create the sorter object. The  sorter class is a
      // thread that starts running right away.
      new sorter();

}
```

Figure 2. Creating a thread

```
class sorter extends Thread
{
      sorter()
      {
      }
      public void run()
      {
      }
}
```

Figure 3. A thread class without invoking the start() method

Implementing the Runnable Interface

Three steps are involved in implementing the `Runnable` interface. First, declare the class as implementing the `Runnable` interface. As part of this process, provide the implementation for the `run()` method—the real action to be executed in a separate thread. Then create an instance of the

```
class Sort
{

     // Create the sorter thread object.
     sorter s = new sorter();
     // start the thread.
     s.start();
}
```

Figure 4. Creating and starting the sorter thread

```
Class myThread extends myClass implements Runnable{

          // the action for the thread
          public void run()
          {
          }

}
```

Figure 5. Implementing the Runnable interface

Thread groups in Java are organized as a tree, with the system thread group at the root.

`Thread` class specifying the class to run as a thread, as an argument to the `Thread()` constructor. Finally, you can run the thread by invoking the `start()` method on the `Thread` object. Details of these steps follow.

Step 1: Specify the action to be executed in the thread. Figure 5 shows an example of a class implementing a `Runnable` interface. A `Runnable` object does the following:

- Declares that it implements `Runnable` interface
- Provides an implementation for the `run()` method; the `run()` method must be specified as a `public void`

Step 2: Create a thread object. A thread is typically created by invoking the constructor on the `Thread` class as follows:

```
Thread thread = new Thread(target);
```

where target is a `Runnable` object, an instance of `myThread` class.

Step 3: Run the thread. After a thread is created, it can be run by invoking the `start()` method on it as follows: `thread.start()`.

Each program decides how to start a thread. For more control on starting the thread, follow the steps outlined above; however, it is possible to invoke the `start()` method in the class constructor after creating the Thread.

The `Thread()` constructor has seven varieties (overloaded forms); the generic form is as follows:

```
public Thread(ThreadGroup group, Runnable
target, String name).
```

Typically, programs do not specify any thread group (or name) to be associated with the thread. If you do not specify a thread group (see below), the program assumes the current thread group. For a typical Java program (application or applet), the `ThreadGroup` is the main group. Remember the default `run()` method of the `Thread` class does not do anything; it simply returns (it is expected to be overridden). Therefore, it is important to override the `run()` method and provide some useful implementation for it. The `name` can be specified to identify threads created by a program, which can be useful when debugging programs.

Thread Groups

Sometimes it is desirable to create several threads and be able to control them collectively. If you need to `stop()`, `suspend()`, or `resume()` a collection of threads as a whole, the notion of thread groups becomes useful. With thread groups in Java, you can also impose security by controlling the ability of a thread to create another thread outside its own group.

Thread groups in Java are organized as a tree, with the `system` thread group at the root. A typical Java application (one with `main()`) will contain at least the `system` and `main` thread groups. A typical Java applet will contain at least these two groups and the `applet` group. A thread group can also contain other thread groups. That is, the `system` thread group contains the `main` group in the case of applications, and it contains the `main` group and the `applet` group in the case of applets.

Thread Properties

When you create a thread in Java, the newly created thread inherits the following properties from the parent thread: daemon state and priority.

After creating a thread but before starting it, you can change both properties. The new thread will run with the new values for these properties. However, once the thread is started, you cannot change the daemon state. Priority works differently; it can be set before starting a thread or while the thread is running.

Java runtime schedules the background (daemon) threads periodically. In the AIX 4 implementation of Java VM, the *Finalizer* thread and the *Async Garbage Collection* thread are examples of background threads; each thread has `Thread.MIN_PRIORITY`. Another important point to remember about daemon threads is that if only these background threads remain in an application, the Java runtime terminates the application.

Thread Control

Thread execution in Java can be controlled in several ways. First, note that Java distinguishes thread creation from running of the thread. You must explicitly invoke `Thread.start()` in order to run a thread; otherwise, the thread object is created, but the thread is not run.

You can suspend a thread during its execution by invoking the `suspend()` method on it. Invoking `suspend()` multiple times on a suspended thread does not affect the suspended state of the thread. A suspended thread can be restarted by invoking `resume()` on it. Invoking `resume()` on a non-suspended thread does not affect the execution of the thread. Similarly, even if `resume()` is invoked on a thread multiple times, only one call to `suspend()` is required to suspend it.

Lifetime of a Thread

A thread starts running when the `start()` method is invoked on it, but there are three ways to terminate it:

- It returns normally from the `run()` method (which constitutes the thread body)
- An uncaught exception is raised in it
- The `stop()` method is called on it

You can invoke `stop()` on a thread only one time. After it is stopped, you should not invoke `start()` on it. If you do, an `IllegalThreadState` exception will occur. When a thread is stopped, you can force the Java Virtual Machine to release the thread object (and its resources) by explicitly setting it to null; the thread object will be garbage collected later.

Thread Synchronization

Thread synchronization deals with mechanisms that allow multiple threads to operate on shared data concurrently without race conditions. In addition, Java allows a thread to wait for event notifications by another thread. The following sections discuss these synchronization mechanisms in detail.

Locks (Mutexes)

Every object in Java has a monitor or lock associated with it; that is, every class and every instance of a class has its own lock. This lock can synchronize concurrent access to the object by multiple threads. These locks are commonly known as *mutexes* or mutual exclusion locks. The synchronized modifier designates places where a thread must acquire the lock before proceeding. All static, synchronized class methods use the same class object lock. Similarly, all instance methods in a class use the same instance object lock. The following examples show static and non-static methods of locking.

Static method. In Figure 6, the static method `sync()` is locked using the class object's lock.

Every object in Java has a monitor or lock associated with it.

```
class StaticSynchExample {

     synchronized static void sync() {
     }
}
```

Figure 6. Locking a static method (class object locking)

Non-static method. The lock of the instance of the class, not the class object's lock, is used in Figure 7.

```
class SynchExample {

     synchronized void sync() {
     }

}
```

Figure 7. Locking a non-static method (instance object locking)

The synchronized modifier can synchronize or lock arbitrary blocks of code. Specify the object to be locked as follows:

```
synchronized(sharedobject) {
     // code to synchronize

}
```

The lock on shared object is acquired (locked) when the code in the block (within {}) is executed. A synchronized method is equivalent to the following:

```
void sync()
{

     synchronized(this) {
     }

}
```

Suppose a non-static method needs to synchronize with a static method. How does it acquire the lock used by the static method, that is, the class object lock? The class object lock can be locked as follows: `synchronized(getClass())` where `getClass()` is a method of the `java.lang.Object` class. Note that `getClass()` returns the runtime class of an object.

From the code in a synchronized block, you can invoke either the same method or another method in the same class without blocking. The same is true for calls on other objects for which the current thread holds a lock. Figure 8 shows an example.

```
class recursive {
     synchronized void method1()
     {
          method2(); //
     }
     synchronized void method2()
     {
     }
}
```

Figure 8. Recursive locking

In Figure 8, `method1()` holds the lock on the instance of the recursive object; it does not block when calling `method2()` on the same object.

Overriding A Synchronized Modifier

You can override the synchronized modifier in subclasses. Suppose you are subclassing from a class with synchronized methods and you want to override some of these synchronized methods. You must mark these methods "synchronized" in your subclass, otherwise these methods will not be synchronized in the subclass. The overridden method must be explicitly declared synchronized or the methods are treated as unsynchronized in the subclass.

Event Notification

Imagine a producer and consumer relationship where the consumer threads must wait for data from the producer thread. When the producer thread has data, it must inform the waiting consumer thread. These waits are best implemented with the consumer thread blocked and not polling some flag to check if the data is available.

To support this type of thread cooperation, Java provides a feature that is similar to the POSIX™ condition variable mechanism. Java allows a thread to release a lock and block atomically. The atomic unlock and block feature allows a thread to check if a condition to wait exists. If so, the thread waits for an event to occur to change the condition. This process involves locking a mutex, checking the condition, and if the condition indicates the thread should block, releasing the lock and then blocking.

Java allows this mechanism through the methods provided in the `java.lang.Object` class. Another thread notifies the blocked thread when the condition changes. This notification is done with the same mutex lock held by the other thread. When it is notified, the blocked thread is awakened and it returns from the `wait()` with the mutex locked again. When `wait()` returns, check the condition again to see if it is valid to proceed or if the wake up is spurious; hence, the thread should wait again. Figure 9 shows the idiom to follow when using the wait/notify mechanism.

```
Waiter:

    synchronized(obj) {
         while (  somecondition != true)
              obj.wait();
    }

Signaller:

     synchronized(obj) {
          somecondition = true;
          obj.notify();
     }
```

Figure 9. Waiting for an event

The waiter and signaler lock the same object: `obj`. The `wait()`, `notify()`, and `notifyAll()` methods must be invoked with the synchronization lock held on their targets. Failure to lock results in `IllegalMonitorStateException` at runtime.

When the wait is done on `obj`, the calling thread gets suspended. Similarly, when signaling, the `obj` is notified, the waiter thread is awakened as a result.

In addition, since `wait()` releases the lock of `obj` when it is blocked, it reacquires the lock when it is awakened. If it cannot get the lock immediately, it waits until the lock is acquired. Typically, the signaler thread releases the lock so that the waiter thread can reacquire the lock.

An important point to remember is that if the thread invoking `wait()` has acquired the same lock *n* times, the call to `wait()` will result in *n* unlock operations. This releases the lock so that another thread can acquire it. When the `wait()` returns, it reacquires the lock *n* times. Therefore, the invariant across a call to `wait()` is maintained; the lock on `obj` is in the same state before and after the call to `wait()`.

Although `wait()` releases the lock on the target object when blocking, other locks held by the thread are not released. This is important to note; otherwise threads may be blocked, holding locks that could potentially prevent progress of other threads that need these locks.

The following example shows two threads: a producer thread and a consumer thread that share a buffer (Vector). In this example, the `main()` creates a producer and a consumer object that share a common buffer object. The shared buffer supports `put()` and `get()` methods. The producer adds elements to the buffer and the consumer removes them. If the shared buffer is too full, the producer waits for the buffer to be emptied. If the shared buffer is empty, the consumer will wait. The producer notifies the consumer when an element is added to the shared buffer; the consumer notifies the producer when an item is consumed. The producer and consumer are synchronized.

Figure 10 shows an example of a producer and consumer sharing a buffer.

Deadlocks

Similar to the POSIX threads model, Java does not detect deadlocks; you must detect or prevent them.

Thread Scheduling

Java specifies a priority-based preemptive thread scheduling discipline; that is, an attempt is made to run the thread with the highest priority at any time. There is no guarantee that the highest priority thread is run always. If a thread with a lower priority is running when a higher priority thread becomes ready to run, the lower priority thread is preempted.

If two threads have equal priority, there is no guarantee that both threads receive equal CPU time; that is, Java does not mandate time-slicing. Some Java implementations may provide time-slicing; however, applications should not assume this. If it is necessary to share time across two threads of equal priority, use the `Thread.yield()` method to periodically yield CPU to another thread.

When a thread is created, it inherits the priority of the thread that created it. However, it is possible to change that priority by calling `Thread.setPriority()` with a priority argument between `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`.

Java specifies a priority-based preemptive thread scheduling discipline.

Thread Blocking and Interrupting

A thread may block in Java when the following methods are executing:

- `java.Object.wait()`
- `java.Thread.sleep()`
- `java.Thread.join()`

A call to `java.Thread.interrupt()` may interrupt a thread from these waits. This interrupt causes the blocked thread to return with the exception `InterruptedException`. Note that a thread may not react immediately to an interrupt.

A call to `stop(Throwable thr)` can also interrupt a thread. The target thread is forced to complete abnormally and to throw the `Throwable` object `thr` as an exception. The exception is delivered asynchronously because it can occur any time during a thread's execution.

In addition, Java Virtual Machine can throw `InternalError` exceptions at any time because of errors that may occur in the Java Virtual Machine. These can arise because of a fault in the virtual machine software, in the underlying host software, or in the hardware. Such exceptions can occur at any point in a Java program.

The Java Development Kit (JDK) 1.1 provides the `Thread` class method `interrupt()` as a way to interrupt a thread asynchronously. This interrupt may not take immediate effect. If the thread

```
import java.util.*;
public class demo {

    public static void main(String[] argv) {
         sharedBuffer s = new sharedBuffer();
         new producer(s);
         new consumer(s);

    }

}
class producer implements Runnable {

    sharedBuffer s;
    producer(sharedBuffer s) {
         this.s = s;
         Thread thd = new Thread(this);
         thd.start();
    }
    public void run() {
        for(int i=0; I <100;i++) {
             s.put(new Integer(i));
             System.out.println("put: int=" + I);
        }
    }

}
class consumer implements Runnable {

    sharedBuffer s;
    consumer( sharedBuffer s)
    {
        this.s = s;
        Thread thd = new Thread(this);
        thd.start();
    }
    public void run() {
        Integer ret;
        for(;;) {
             ret= s.get();
             System.out.println("get: int=" + ret);
        }
    }
}
class sharedBuffer {

    Vector v;
    static final int MAX=10;
    sharedBuffer() {
        v = new Vector();
    }
    synchronized void put(Integer I ) {
        while(v.size() == MAX){
             System.out.println("MAX size is reached");
             try{
                 wait();
             }
             catch(InterruptedException e){}
             System.out.println("put(): try again—out of Wait()");
        }
```

(continued on next page)

Figure 10. Producer/consumer sharing a buffer

(continued from previous page)

```
        v.addElement(I);
        notifyAll();
        return ;
    }
    synchronized Integer get() {
         while(v.isEmpty() ) {
              try{
                  System.out.println("get(): buffer is EMPTY");
                  wait();
              }
              catch(InterruptedException e){}
              System.out.println("get(): out of Wait()");
         }
         Integer ret = (Integer)v.firstElement();
         v.removeElement(ret);
         if(v.size() < MAX) notify();
         return ret;
    }
}
```

Figure 10. Producer/consumer sharing a buffer

is waiting, it is awakened and an exception is generated. An interrupted exception is thrown.

Note: When a thread is blocked in `wait()`, `sleep()`, or `join()`, and an interrupt is sent to it (by `stop()` or `interrupt()`), the desired interrupt is not delivered to the thread in JDK 1.0.2 on AIX 4. It is not clear if this is just a bug. The interrupt is delivered if the thread is not blocked, but is executing some computation.

Security

Java allows programs to prevent certain actions on a thread (in addition to other system resources) for security reasons. Your program can check security before performing actions to stop, suspend, resume, change priority, set name, and set daemon on a thread. It can disallow such actions, if so desired. The way to provide this security is to subclass the `java.lang.SecurityManager` and override the `checkAccess()` method to do the desired checking for thread operations.

The methods of the `SecurityManager` provide the strictest security by throwing exceptions for all of its methods. You must override its methods to allow useful work to be done. Java runtime invokes the `checkAccess()` method before performing thread-related actions, and it is up to the `checkAccess()` to disallow the action as needed to enforce security.

The example in Figure 11 shows how security can be specified. We provide the `threadSecurityMgr` class, which inherits from the `SecurityManager` class and overrides `checkAccess()` to throw exceptions whenever a thread is stopped, suspended, resumed, or its priority, name or daemon state is set. Figure 12 shows the results when run on AIX 4, JDK 1.0.2. A set of checks can be imposed for most thread operations.

There are certain rules for installing a security manager for a program. You can install a security manager only once, and after it is installed, you cannot change it. This makes it important to override all the methods of the `SecurityManager` class as needed and install the new manager early in the program. Java-enabled browsers such as Netscape Navigator install their security manager class before they run Java applets.

Applets and Threads

A Java applet can run in a Java-enabled Web browser, such as Netscape Navigator 3.0. The applet class in Java provides methods such as `start()`, `stop()`, `init()`, and `destroy()`. Although the methods of the `Applet` class are similar to those of the `Thread` class, they are not identical. For example, the `start()` and `stop()` methods of an applet can be called several times by the Web browser as the user moves around scanning the Web pages. Typically, a Web browser starts the applet when the applet is displayed, and stops when the user moves to another page or scrolls the applet out of view. When an applet is stopped, the underlying thread is suspended; it is resumed when the

Specifying a Security Manager

```
import java.awt.*;

class threadSecurityMgr extends SecurityManager {
     public void checkAccess(Thread g) {
          throw new SecurityException("access to thread" + g);
     }
}
public class secure extends Thread {
     static public void main(String[] args) {
          System.setSecurityManager(new threadSecurityMgr());
          secure s = new secure();
          s.start();
     }
     public void run() {
          System.out.println("Secure Thread: now set priority");
          setPriority(Thread.MAX_PRIORITY);
          System.out.println("Secure Thread: now stop it ");
          stop();
     }
}
```

Results of running the Security Manager example

```
Secure Thread: now set priority
java.lang.SecurityException: access to threadThread[Thread-2,5,main]
at threadSecurityMgr.checkAccess(secure.java:8)
at java.lang.Thread.checkAccess(Thread.java)
at java.lang.Thread.setPriority(Thread.java)
at secure.run(secure.java:25)
```

Figure 11. Specifying a Security Manager and the results

applet is started again. When the browser invokes the applet's `destroy()` method, the applet's thread is stopped and garbage collected.

In the applet example in Figure 12, an applet implements `Runnable` interface. In addition to any threads the Java runtime creates, we create a thread to execute the `run()` method, which invokes `repaint()`. This in turn invokes the `paint()` method, which then draws the current date and time on the panel. We create this thread with the applet class as the thread target class and start it when the applet's `start()` method is called by the browser. Note the number of threads that exist in the applet and their function.

The AIX 4 JDK 1.0.2 creates several thread groups: `system`, `main`, and `applet-myapplet.class`. The thread in the `system` thread group is the `Finalizer` thread. Threads in the `main` thread group consist of the main thread, the `AWT-Motif` thread, and the `ScreenUpdater` thread. The threads in the `applet-myapplet` thread group are Thread-1 and Thread-2. All applets have the Thread-1 thread. Since `myapplet` class creates another thread, Thread-2 is created. Thread-1 invokes the `init()` and the `start()` methods in the `applet` thread group. The `AWT-Motif` thread invokes the `paint()` method. Thread-2 invokes the `run()` method.

Java Core Classes and Thread Safety

A Java program typically subclasses (inherits from) the base classes, such as those provided in the following packages: `java.lang`, `java.awt`, `java.io`, `java.util`, `java.net`, and `java.applet`. For a multithreaded program to be safe with respect to threads, it is important to know if the classes (methods) are thread-safe, or if the application developer must do external locking before invoking any of the methods.

Java classes have evolved from JDK 1.0 through JDK 1.1. Some classes have improved

thread-safety. All Java base classes are thread-safe and do not require external locking; that is, you do not need to synchronize calls to the base Java classes because they are already supposed to be thread-safe (unless the documentation says otherwise).

Another point regarding thread-safety is that you cannot override static or final methods in a class. For example, you cannot override the synchronization methods `wait()` and `notify()` in `java.lang.Object` class because these are declared final. If you are working with object references on the client side using Java's Remote Methods Invocation (RMI) Application Programming Interfaces (APIs), synchronizing on the local object references does not translate to synchronizing on the remote (real) objects.

AIX 4 Java Threads

To help you better understand the execution environment, we will describe some Java runtime threads found in AIX 4 JDK 1.0.2. Our intent is to help you understand the actual control flow or how applications/applets work in Java. You must not directly interact with these threads or manipulate them in another way. AIX 4 support for Java threads is based on the pthreads provided in the AIX 4 pthreads library. Pthreads maps application threads to the native kernel threads. Currently, there is a one-to-one mapping

```
import java.applet.*;
import java.awt.*;
import java.util.*;
public class myapplet extends Applet implements Runnable {

     private Thread updateThread;
     public void init() { }
     public void paint(Graphics g) {
          g.drawString(new Date().toString(),10,20);
     }
     public void run() {
          for(;;){
               try {
                   Thread.sleep(1000); // sleep for 1000 milliseconds
               }
               catch(InterruptedException e){

                   System.out.println("Sleep interrupted");
               }
               repaint();
          }
     }
     public void start() {
          if(updateThread == null){
               updateThread = new Thread(this);
               updateThread.start();
          }else{
               updateThread.resume();
          }
     }
     public void stop() {
          Thread mainthd = Thread.currentThread();
          updateThread.suspend();
     }
     public void destroy() {
          updateThread.stop();
          updateThread=null;
     }
}
```

Figure 12. Threads in applets

```
public class helloWorld
{

     static public void main(String[] args)
     {
          new helloWorld ();
     }
     public helloWorld()
     {
          Thread mainthd = Thread.currentThread();
          System.out.println("Current Thread=" + mainthd);
          System.out.println(" Listall Thread Groups and Threads:");
          PrintThreadsInfo.getAllThreads();
     }

}
public class PrintThreadsInfo {

     public static void getAllThreads() {
               ThreadGroup rootgrp;
               ThreadGroup parentgrp;
               ThreadGroup thdgrp=Thread.currentThread().getThreadGroup();
               rootgrp = thdgrp;
               parentgrp = thdgrp.getParent();
               while(parentgrp != null) {
               rootgrp = parentgrp;
               parentgrp = parentgrp.getParent(); }
               listgroup(rootgrp);
     }
     static void listgroup(ThreadGroup group)
     {
          if(group == null) return;
          int num_threads = group.activeCount();
          int num_groups = group.activeGroupCount();
          Thread[] threads = new Thread[num_threads];
          ThreadGroup[] groups = new ThreadGroup[num_groups];
          group.enumerate(threads,false);
          group.enumerate(groups,false);
          System.out.println( " Group Name = " + group.getName());
          System.out.println( " Group Max Priority="+ group.getMaxPriority());
          System.out.println((group.isDaemon()? " (Daemon Group?)Yes":";      (Daemon Group?)No") );
          for(int I=0;i<num_threads;i++) print_info(threads[i]);
          System.out.println(" ");
          for(int I=0;i<num_groups;i++) listgroup(groups[i]);
     }
     static void print_info(Thread t)
     {
          if(t==null) return;
          System.out.println("\tThread=" + t.getName() );
          System.out.println("\t\tPriority=" + t.getPriority());
          System.out.println("\t\t" + (t.isDaemon() ?       "Daemon(Yes)":"Daemon(NO)"));
          System.out.println("\t\t" + (t.isAlive() ? "Live":"not Live"));
     }

}
```

Figure 13. Extracting information on all the thread groups and threads in an application

of the pthreads to the kernel threads. The Java Virtual Machine creates threads that are hosted using the pthreads package.

Every thread in Java belongs to a thread group, which may contain other thread groups or the actual threads themselves. There is a notion of a parent of a thread group, but there is one thread group called the `system` group for which there is no parent. The `system` group is the root of all the thread groups. A thread group is governed by certain attributes, such as the maximum priority a thread belonging to this group can have, the name of this thread group, and whether this group has a daemon thread group.

In AIX 4, the Java runtime creates the following thread groups for a non-GUI application (a program invoked through a Java command and contains main): `system` and `main`. The `system` thread group is a non-daemon thread group with the maximum priority of any thread within it set to `Thread.MAX_PRIORITY`. The `system` group consists of two threads, the `Async Garbage Collector` and the `Finalizer` threads, both daemon threads with the priority set to the value `Thread.MIN_PRIORITY`.

The `main` thread group is a single thread when the main of the application is invoked. This main thread, a non-daemon thread, runs at `Thread.NORM_PRIORITY`. All application threads belong to one of the following:

- The `main` thread group
- Thread groups created by the application
- Thread groups created by the Java runtime

If the application creates a GUI, such as a frame, the AIX 4 Java runtime creates two additional threads: the `AWT-Input` and the `AWT-Motif` thread. These two threads also belong to the `main` thread group and run as non-daemon threads with priority set to `Thread.NORM_PRIORITY`. The `AWT-Motif` thread handles all GUI-based events. Thus, if your application has a `paint()` or `handleEvent()` method, these are invoked in the thread `AWT-Motif` thread.

In addition to the `system` and `main` thread groups, a new thread group is created within an applet, for that applet class. This applet thread group is a non-daemon thread group. A non-daemon thread with priority of six is created in this applet thread group. This applet thread executes the `init()` method of the applet.

```
Current Thread=Thread[main,5,main]
Listall Thread Groups and Threads:
Group Name = system
Group Max Priority=10
(Daemon Group?)No

     Thread=Async Garbage Collector

          Priority=1
          Daemon(Yes)
          Live
     Thread=Finalizer thread
          Priority=1
          Daemon(Yes)
          Live

Group Name = main
Group Max Priority=10
(Daemon Group?)No

     Thread=main
     Priority=5
     Daemon(NO)
     Live
```

Figure 14. Threads in an application

To illustrate the Java runtime environment, consider first a Java application, then a Java applet. The complete code for both is shown for illustration purposes. The results of running the two examples were summarized above.

Java Application Example

In this example, we create a simple `helloWorld` class to display all the thread groups and threads running in the process. The `helloWorld` class does not create any threads; all the threads and thread groups you see are created by the Java runtime. The static method `getAllThreads()` of `PrintThreadsInfo` class also illustrates how to navigate the thread group and get to the root thread group.

Figure 14 shows the results of running the program shown in Figure 13 on AIX 4 JDK 1.0.2.

Java Applet Example

Details of the actual threads created in an applet may vary based on how the applet is launched. For instance, you can run the applet using the `appletviewer` command. The applet also can be launched through a Java-enabled Web browser such as Netscape Navigator 3.0.

We have summarized the results of running the applet using the `appletviewer` command. Results from running in a Web browser do not vary much from using this command.

In an applet, Java runtime creates a new thread group named after the particular applet class, in addition to the `system` and `main` thread groups. The `system` thread group consists of the `Async Garbage Collector` and `Finalizer` threads; the main thread group consists of the `main`, `AWT-Input`, `AWT-Motif`, and `Screen Updater` threads. Two threads in the `system` group are daemon threads; the rest are not. For example, in the following `helloWorldApplet` class, Java runtime creates the `applet-helloWorldApplet` thread group plus the system and main thread groups. In addition, an applet thread is created in this applet group. The `paint()` method of the applet is invoked in the `AWT-Motif` thread of the main thread group. Figure 15 shows an example of extracting information on all the thread groups and threads in an applet.

To run this applet, create the HTML file found in Figure 16.

Type the following command to run the HTML file: `appletviewer run.html`. Figure 17 shows the results of running the applet on AIX 4 JDK 1.0.2.

Summary

Java provides a simple, but powerful, programming model for multithreading applications. This overview covered Java's memory model, its threads-related language features, and the built-in Java classes. It also detailed the Java threads programming model.

We explored some implementation details of the AIX 4 JDK 1.0.2 to understand the actual runtime threads created. This is helpful in debugging programs as well as in understanding the control flow in Java programs.

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

     public void init() {

          Thread mainthd = Thread.currentThread();
          System.out.println("Current Thread=" + mainthd);
          PrintThreadsInfo.getAllThreads();
     }
     public void paint(Graphics g) {

          g.drawString("Hello World Applet", 10,100);
          System.out.println("paint()—Current Thread=" +
               Thread .currentThread());
     }

}
```

Figure 15. Extracting information on all thread groups and threads in an applet

```
run.html:

  <APPLET CODE=helloWorldApplet.class WIDTH=400 HEIGHT=400>
  </APPLET>
```

Figure 16. Simple HTML file to run the sample in Figure 15

```
     Current Thread=Thread[thread applet-helloWorldApplet.class,6,group app let-helloWorldApplet.class]

          Group Name = system
          Group Max Priority=10
          (Daemon Group?)No
               Thread=Async Garbage Collector
               Priority=1
               Daemon(Yes)
               Live
          Thread=Finalizer thread
               Priority=1
               Daemon(Yes)
               Live

     Group Name = main
     Group Max Priority=10
     (Daemon Group?)No

          Thread=main
               Priority=5
               Daemon(NO)
               Live
          Thread=AWT-Input
               Priority=5
               Daemon(NO)
               Live
          Thread=AWT-Motif
               Priority=5
               Daemon(NO)
               Live
          Thread=Screen Updater
               Priority=4
               Daemon(NO)
               Live

     Group Name = group applet-helloWorldApplet.class
     Group Max Priority=6
     (Daemon Group?)No

          Thread=thread applet-helloWorldApplet.class
               Priority=6
               Daemon(NO)

          Live

paint()—Current Thread=Thread[AWT-Motif,5,main]
```

Figure 17. Threads in an applet—results of running the sample in Figure 13

References

Gosling, James; Joy, Bill; Steele, Guy. *The Java Language Specification*. Addison-Wesley, 1996.

Flanagan, David. *Java in a Nutshell*. O'Reilly & Associates, Inc., 1996.

Lea, Doug. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1996.

Chary Tamirisa, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Internet: chary@austin.ibm.com. Mr. Tamirisa was the team leader for the threads package on AIX and OS/2® DCE. He has also worked in the fields of communication protocols, system software, and National Language Support. Mr. Tamirisa has an MS in Computer Science from McGill University and a BTech in Electrical Engineering from the Indian Institute of Technology in Madras, India.*

Comparison of POSIX and Java Threads Models

By Chary Tamirisa

AIX 4 supports multithreaded application development with a C language-based user level threads library called the Pthreads library, based on the POSIX.1c draft 7 standard. Typically, concurrent applications are written in C and C++ and use the Pthreads library. With the porting of the Java Development Kit (JDK) 1.0.2 to AIX 4, we have another threading model in the Java environment. This article compares the POSIX and the Java threads models. It is intended to explain the Java threads model to those familiar with the POSIX threads model. This article assumes the reader is familiar with the Pthreads library and the Java environment.

AIX 4 provides a threads library based on POSIX.1c draft 7 and supported by kernel threads. In fact, there is a one-to-one mapping of user-level threads to kernel threads. C and C++ programs typically use this threads package. POSIX threads (pthreads for short) is biased toward the C language and is not object-oriented.

Java threads and pthreads have several common features. For example, the two models have a similar synchronization mechanism. POSIX threads offers features such as thread-specific data and signal (UNIX® signals) support that are not found in Java threads. We will examine the two models briefly with focus on a few important pthreads APIs.

Thread Management

The following sections discuss various aspects of thread management comparing POSIX and Java.

Process versus threads. Before the advent of threads, the concept of a process referred to resources used by a program (such as the address space, file descriptors) as well as to the entity in an operating system that can be executed or scheduled. After threads were introduced, the term process now defines the resources and the thread defines a sequence of execution. Both POSIX and Java support the concepts of process and threads within process, with the separation maintained.

Thread creation. POSIX threads define a `pthread_create()` API for creating a thread. Figure 1 shows the syntax.

The `pthread_create()` Application Programming Interface (API) takes four arguments:

- Thread handle for the newly created thread is an output argument
- Optional attribute for the new thread
- Actual thread body to be executed in the newly created thread
- Optional parameter passed to the new thread

The thread handle and thread function are the two important arguments in the thread creation API. When `pthread_create()` returns, the new thread is created and ready to run; it is not necessary to start it explicitly. Unlike Java, all threads are created ready to run.

In Java, you implement the `java.lang.Runnable` interface, which involves implementing the thread body in the `run()` method. You then pass an instance of this class to the Thread constructor (the `java.lang.Thread` class). Finally, you start the thread running by invoking the `start()` method on the Thread object.

Chary Tamirisa

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

Figure 1. POSIX thread creation API

Thread Termination. In pthreads, a thread is terminated by simply returning from the thread body or by invoking `pthread_exit()` explicitly. In Java, a thread is terminated by simply returning from the `run()` method.

In pthreads, if the main thread exits by simply returning from the main thread, the whole process is terminated. To terminate only the main thread and not the whole process, you need to invoke `pthread_exit()` explicitly if the main thread needs to be terminated. In a Java application (with `main()`) if the main thread returns, the process does not exit if non-daemon threads are present. The Java Virtual Machine terminates a process if any thread invokes the `exit()` method of the Runtime or System classes—if the security manager allows the exit operation.

Thread Attributes. In pthreads, every thread has a set of attributes that can specify a thread's stack size, the stack address, and the detach state. Java does not have corresponding APIs; however, the Thread class allows you to specify if a thread is a daemon thread. You also can associate a name with a thread for easy identification. This is often useful when you are debugging a program.

Thread IDs. Every pthread has an opaque handle associated with it; this handle is returned when a pthread is created. Similarly, every Java thread has a Thread object associated with it, which is used to manipulate the thread.

Waiting for Thread Termination. Pthreads provides the detach state to control what happens when a thread is terminated. You can set the detach state of a thread to indicate whether you are interested in waiting for the exit status after the thread terminates. If you are not interested in the exit status of a thread, you can set the thread's detach state to indicate that any system resources associated with it should be freed up after it terminates.

The `pthread_join()` API allows you to wait for a thread to obtain its exit status. Java does not have a detach state associated with a thread. However, similar to pthreads, you can join with a thread using the `Thread` class' `join()method`. Java does not return any exit status associated with a thread when `join()` returns. In pthreads, the `pthread_join()` API does not have a timeout associated with it; it just blocks until the specified thread terminates. Java, however, allows a timeout to be associated with the `join()` method.

Detaching a Thread. In pthreads, you can specify the detach state of a thread by invoking `pthread_detach()` on a running thread. If the detach state of a thread is set, the pthreads library releases all the library resources associated with a thread. Java does not have a similar API.

Dynamic Package Initialization. Pthreads allows you to invoke an initializer only once using the `pthread_once()` API. This is useful in libraries where the locks must be initialized before they are used, and initialized only once. Since Java objects are always associated with a lock, an application does not need to initialize the locks associated with an object. Java has no such mechanism.

Comparison of Thread IDs. You can compare two pthreads using the `pthread_equal()` API. In Java, you can compare two Thread objects using the `equals()` method of `java.lang.Object` class.

Thread Suspend/Resume. Pthreads does not provide any explicit APIs for suspending or resuming a thread. Java allows a thread to be suspended or resumed.

Thread Synchronization

Pthreads provides the pthread mutex APIs to ensure mutual exclusion of threads to protect critical sections. The Java language supports the keyword synchronized (Expression) to provide locking for critical sections. Pthreads and Java provide similar functionality.

In addition to mutexes, pthreads allows a thread to wait for an event through the condition variable APIs. The condition variable is generally associated with a mutex and a boolean variable. Pthreads provides two APIs for waiting: the non-timed wait API `pthread_cond_wait()` and the timed wait API `pthread_cond_timedwait()`.

The following discussion concerns `pthread_cond_wait()`, but it also applies to the timed wait API. A thread invokes the condition variable wait API with the mutex locked.

In a loop (typically `while()` loop), determine if the flag associated with the event of interest indicates whether the current thread should wait. When the thread invokes `pthread_cond_wait()`, it atomically releases the mutex lock and blocks.

Another thread changes the flag and signals this thread when the event occurs. This thread is awakened, obtains the mutex lock again, and returns from `pthread_cond_wait()`. The while loop forces the thread to evaluate the flag again, and this protects against spurious wake ups. Once the loop is successfully completed, the thread assumes that the condition for which it has been waiting is satisfied and it can proceed. The mutex lock protects the shared boolean flag.

Finally, the mutex is unlocked and the thread continues. Figure 2 shows how condition variables are used in pthreads.

The mechanism of condition variable waits is similar to the `wait()/notify()` family of methods provided in the Java class `java.lang.Object`. In fact, the usage of these methods involves similar steps as outlined above. The pthread's condition variable APIs expect that a non-recursive mutex as the mutex lock is unlocked only once before blocking in `pthread_cond_wait()/pthread_cond_timedwait()`. If a recursive mutex is used and it is locked more than once by the thread, it cannot be used in the condition variable APIs.

Java is different. The same thread can invoke the `synchronized()` method on the same object several times and still block on the object by invoking the `wait()` method on the object. The `wait()` method will unlock the object as many times as needed and make the object lockable by other threads.

Thread-Specific Data

Pthreads supports thread-specific data through a set of APIs. You can associate a unique key with a memory you allocate on a per-thread basis. First, create a thread-specific key. The pthreads library ensures that all the threads in the process have a null pointer associated with this key. Each thread can then allocate memory and associate that memory with this key. Once this is done, each thread can access this thread-specific memory by simply using the key associated with the memory. The pthreads library sets a limit on how many keys you can create. When you create a key, you can also specify that a destroy function be invoked to free up the thread-specific memory when the thread terminates. Java does not offer support for thread-specific data.

Execution Scheduling

Pthreads offers two fixed real-time priority-based scheduling policies called First-In-First-Out (FIFO) policy and Round Robin (RR) policy. In

```
/*Following is a code fragment to illustrate the pthread condition variable API */

#include <pthread.h>

pthread_mutex_t mutex;
pthread_cond_t condition_variable;
int flag;

pthread_mutex_lock(&mutex); /* lock the critical section */

while( !flag) {

     /* boolean variable indicating if the event has occurred */
     pthread_cond_wait(&condition_variable, &mutex);
     /* Wait for another thread to signal this thread on a change in state */

}

pthread_mutex_unlock(&mutex); /* release the lock */
```

Figure 2. POSIX condition variable API usage

FIFO policy, a thread with the highest priority runs until it blocks; there is no time-slicing. In the RR policy, a thread with highest priority runs; however, threads with equal priority are time-sliced. In addition to these polices, pthreads allows an implementation to provide any scheduling policy under the `SCHED_OTHER` policy.

Pthreads has priority inversion where a high-priority thread may be blocked for a lock held by a low-priority thread. This low-priority thread may be preempted from running by a higher priority thread that may run indefinitely. In Java, the scheduling policy is preemptive; that is, a thread with a low priority will be preempted from running by a higher priority ready to run a thread. However, Java does not specify any time-slicing of threads of equal priority. To share the execution among threads of the same priority, you can use the `Thread` class' `yield()` method.

Synchronization Scheduling

To prevent priority inversion, pthreads specifies APIs to contain the priority inversion. Java does not have any such mechanisms.

Thread Cancellation

Pthreads allows a thread to be terminated asynchronously in addition to the `pthread_exit()` API. This cancel mechanism is useful when a thread needs to be terminated from another thread. Pthreads provides a way to control whether a thread can be canceled. In addition, pthreads allows an application to clean up state when a thread is canceled.

Java also allows a thread to be canceled from another thread using the `stop()` method of the `Thread` class. When `stop()` is invoked on a thread, the target thread is forced to complete whatever it is doing abnormally. In addition, the target thread throws an exception. Java Version 1.1 specifies a new method `interrupt()` to post an interruption to a thread. The target thread does not necessarily react immediately to the interrupt. The target thread throws the exception: `InterruptedException`.

Summary

We have presented a brief comparative study of the POSIX and Java threads models describing their salient features. Although Java does not offer all of the POSIX features in the JDK 1.0.2, the Java threading model is powerful.

References

Gosling, James; Joy, Bill; Steele, Guy. *The Java Language Specification*. Addison-Wesley, 1996.

Flanagan, David. *Java in a Nutshell*. O'Reilly & Associates, Inc., 1996.

Lea, Doug. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1996.

Chary Tamirisa, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Internet: chary@austin.ibm.com. Mr. Tamirisa was the team leader for the threads package on AIX and OS/2® DCE. He has also worked in the fields of communication protocols, system software, and National Language Support. Mr. Tamirisa has an MS in Computer Science from McGill University and a BTech in Electrical Engineering from the Indian Institute of Technology in Madras, India.*

Atlanta Olympics WOMplex

By Andy Stanford-Clark

This article describes the major components of the Atlanta Olympics WOMplex and attempts to give a flavor of some cutting-edge technology developed for the project.

For the summer of '96, a team of IBM programmers and engineers, based in Southbury, Connecticut, built the official Internet Web site for the Atlanta Olympic Games. The project was immense in every respect—from the amount of content in the site, to the volume of traffic through the site, to the amount of equipment deployed, and the number of hours worked by the team.

WOMplex is an umbrella term that describes the combination of the Web Object Manager (WOM) Internet content and application server, the scalable architecture, the requisite scalable parallel processing hardware, the content authoring environment, the load-balancing systems, the user-level applications on the Web site, and the subsystems that support those applications. In other words, a WOMplex is a scalable, manageable, globally distributable Internet content and application server comprising both hardware and software.

Statistics

The URLs http://www.atlanta.olympic.org and http://results.atlanta.olympic.org together constituted the largest Web server in the world. During the 17 days of the Olympics, the site received just over 192 million hits, with peak daily hits of 16.9 million. Drilling down to the "Sneak Peek" application, which provided a constantly changing montage of captured images from 38 video feeds from the Games in Atlanta, the servers handled up to 21,000 images per hour. What makes these figures especially impressive is that every page served was built dynamically from its component objects at the instant the page was requested.

To provide this level of service 24 hours a day required non-trivial amounts of equipment. The Atlanta site was served by 50 nodes of a distributed IBM RS/6000™ Scalable POWERparallel™ system, spread across five locations in four countries. Most of the processors were at the primary WOMplex site at one of Integrated Systems Solutions Corporation's (ISSC's) major Facilities Management Centers in Southbury. This site provided the required infrastructure for a project of this scale—Uninterruptable Power Supplies(UPS), good physical security, high-bandwidth connectivity into major U.S. Network Access Points (NAPs), and raised floor.

Several nodes—typically some portion of an RS/6000 SP frame—joined the WOMplex at four mirror sites:

- Cornell Theory Center at Ithaca, New York
- IBM U.K. Laboratories at Hursley, U.K.
- University of Karlsruhe near Mainz, Germany
- Keio University near Tokyo, Japan

Each mirror site replicated most of the content of the Olympic site. Notable exceptions were the results system and the rapidly changing Sneak Peek images, which were served only from the primary site at Southbury.

Andy Stanford-Clark

Presented at Get Connected Technical Interchange '96, IBM Hursley U.K., October 1996

System Architecture for Atlanta WOMplex

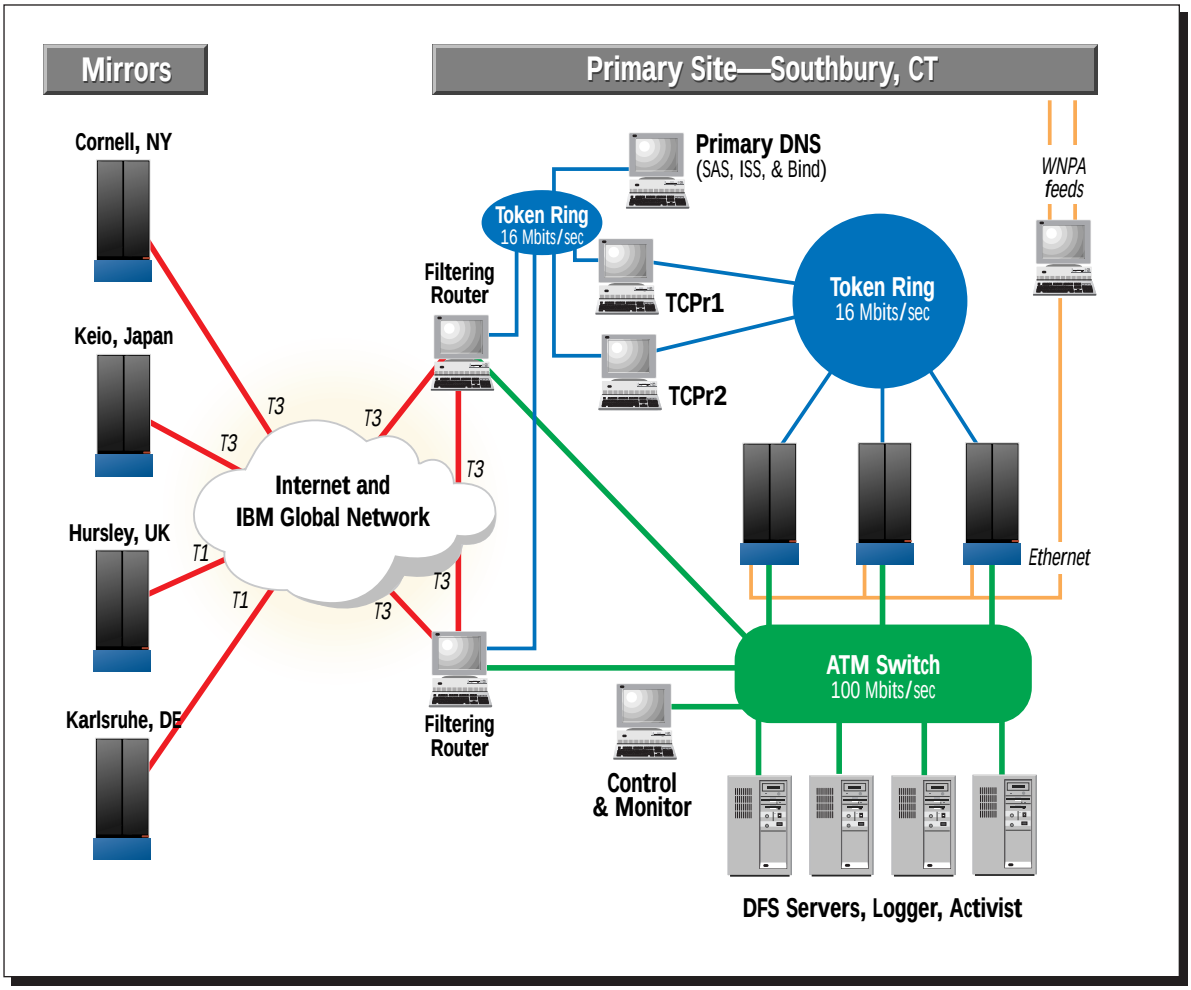


Figure 1. System architecture used for the Olympic games in Atlanta

During the last few months before the games, tickets for seats to events at the games were sold over the Internet by a server based on the IBM Net.Commerce server. In all, this generated over $5 million in ticket sales.

System Architecture

Figure 1 shows the overall system architecture for the Atlanta WOMplex. The "cloud" combines the Internet and the IBM Global Network. T3 connections are DS3 links providing 45 Mbits per second. Filtering routers are Cisco® 7500 Series routers with sets of carefully crafted filter rules to allow only the desired packets in and out from desired sources and destinations. TCPr1 and TCPr2 are the TCP routers, or WOMsprayers—these will be discussed later. Together SAS, ISS, and Bind form the name resolution service. The WNPA feed is a 56 Kbits per second leased-line connection to Atlanta, which provided the feed of results to the databases that were used to drive the Internet Results System. Several leased lines were installed for redundancy. The roles of the machines labeled Logger and Activist are discussed below.

Infrastructure

The IBM RS/6000 SP system provided the WOMplex infrastructure. At the main Southbury site, three frames of the RS/6000 SP were filled with mainly "thin" nodes (equivalent to the RS/6000 Model 390). We did not use the RS/6000 SP high-speed switch in this application—all communication between processors was done over the Asynchronous Transfer Mode (ATM) network.

Control workstations for the main SP frames were in the machine room near the RS/6000 SP. Access to the SP control workstations at the remote mirror sites was handled through X-Windows sessions running across the Internet, using

Secure SHell (SSH) (described below) for secure connections. The Operations Center in Southbury could control every processor in the WOMplex—right down to power cycling.

ATM Network

The outbound network for data from the Web site was routed over a 100 Mbits per second ATM network. ATM uses centralized switching technology to provide a high-bandwidth network. The site networking was highly complex, with multiple Token-Ring, Ethernet™, and ATM networks connected to the Internet through two filtering routers. IBM 8260 switching hubs enabled this level of connectivity. A 16 Mbits per second Token Ring handled the inbound networking. *Note:* Network requirements for a Web site are highly asymmetrical—one packet that contains an HTTP request can provoke the transfer of potentially many megabytes of data from the servers.

Transarc DFS Cell

Transarc's Distributed File System (DFS) ensured that all WOMplex Web servers had access to up-to-date information. (For those familiar with Andrew File System (AFS®), think of DFS as the "grown up" AFS.) DFS is based on the fileset concept, a collection of files handled in the same manner as a single group. Each fileset in a DFS cell resides on a nominated server; it may have mirrors on other nominated servers. Fileset replication happens in two ways: on a timer interval or (often more useful) on a manual "push" from the primary server, when the application or user knows that a batch of updates has been completed.

Each WOM server in the WOMplex was a DFS client configured to request files from the nearest DFS server. At remote mirror locations, the clients retrieved their data from a local server that was kept in sync with the main servers in Southbury. One big advantage of DFS in the Web server environment is client caching. During the Olympics, most clients achieved a 98% cache hit rate.

The primary DFS server in Southbury was distributed across three RS/6000 Model 570 servers and two RS/6000 SP wide nodes. It used 128 GB of IBM 7137 RAID storage, configured with replication and mirroring to give 32 GB of effective storage with three-way redundancy in addition to the RAID failsafe.

The entire cell was administered centrally from Southbury. Problems with filter rules on routers and TCP/IP routing across the Internet make setting up a very large distributed DFS cell a complex process; although when it is running, the system works extremely well. The single DFS cell spanned six locations: Southbury (the main server), Cornell, Keio, Hursley, Karlsruhe (the mirror sites), and Atlanta, where the content was being generated for the Web site.

Internet Connectivity

To ensure the availability of enough bandwidth for the Olympics, four DS3 circuits (T3—45 Mbits per second) were connected into four major U.S. Network Access Points (NAPs): Mae-East, Mae-West, Sprint NJ, and Ameritech™ Chicago. Regardless of the origin of a request, by using a routing algorithm that minimized hop counts, the response was sent to the major NAP closest to the source of the request. Most of the world's major non-U.S. service providers connect directly into one of these four NAPs, so the site was well connected. Multiple circuits also protected the site against any link failure. Packets were automatically routed over other links or could be manually routed around trouble spots, which occurred from time to time at various NAPs.

Security

The entire Atlanta Internet site, the software development environment, and all DFS infrastructure were located outside the IBM firewalls. Consequently, site security was always a high priority for everyone because such a high visibility site was an obvious target for hackers.

High security must be built into a site from the ground up.

High security must be built into a site from the ground up; it cannot be retrofitted onto an existing site. The filtering routers provided the main protection against invasion, and the filter rules for these machines were a major design and maintenance effort. Interworking with the remote mirror sites involved setting filter rules at the remote sites to allow only User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) packets on specific port numbers in and out from specified Internet Protocol (IP) addresses.

A major problem for the primary site at Southbury was the need for extensive access to systems inside the DMZ ("De-Militarized" Zone) at the remote sites; the rest of the world could only have port 80 World Wide Web access. Heterogeneous router technology across the world ensured that many people learned a lot about writing filter rules in a short space of time!

Kerberos and authentication servers were used to secure DFS. Sophisticated password schemes for

the machines were generated and then frequently changed. The apparent inconvenience of complex password schemes was easy to justify in the high profile environment of the Olympics.

Transmission of system control information, including machine root passwords, across the Internet was a potential major exposure. Early adoption of Secure SHell (SSH) for all terminal communication between machines solved this problem. AIX® and Windows® 95 clients for SSH allowed developers to use SSH as a direct replacement for the telnet they would have used otherwise. A version of the SSH client that worked through a SOCKS firewall allowed developers inside the IBM firewall to access the servers and development lab machines.

ISS provided another level of security with the "caching, routing name server," which was deployed as the primary name server for the Olympics site. In this Domain Name Server (DNS) emulation mode, ISS had authority over one or more specified subdomains. If ISS received a name resolution request for a subdomain over which it did not have authority or a name that was not being used for load balancing, then ISS passed the request to one of two backend, ordinary, name servers—typically knowledgeable DNSs that could resolve nearly any request.

The IP address of the original request determines the backend name server that receives the request. ISS configuration has a table of subnet mask specifications. The client IP address, or name server, requesting the name resolution is matched against this table. If a match occurs, the client is considered internal, and passed to one backend DNS; otherwise, it is considered external and sent to the other one.

From the practical standpoint, even with a distributed, multi-site operation, the appropriate client machines can access all of the main site's internal server names and addresses, whereas the rest of the world can only access the limited set of names and addresses (possibly only the Internet address) that the DNS administrator made available. The ISS name server also blocks the DNS Zone Transfer operation, a popular method to access lists of internal machines that are considered hacking targets.

The WOM system has been iteratively developed to support high-volume Web sites for sporting events.

The WOM Web Server

The Atlanta Web site used the latest version of IBM's Web Object Manager system, known as the *WOMserver*. The WOM system has been iteratively developed to support high-volume Web sites for sporting events and IBM external Web sites since 1994. Notable events have included Wimbledon® Tennis, Masters Golf, U.S. Open™ Tennis, and the ACM Chess Challenge (Kasparov versus Deep Blue).

Object-oriented Perl was the language used for the Atlanta server. Supporting applications and subsystems on the site were written in C, Perl, and Java™. Several prototype versions of the WOMserver written entirely in Java were also deployed in the WOMplex for experimental, demonstration, and testing purposes. A specialized binary server, based on the Apache Web server and run on a separate port number, handled binary objects—GIF, JPEG, AVI, MPEG, WAV, Java classes, and Bamba audio and video. All references to binary objects in the HTML source dynamically served as pointers to this port (for example, http://www.atlanta.olympic.org:8080/).

The data flow for entering authored content into the WOMserver had several steps. Content was authored by several types of systems, including traditional HTML editors, image editing tools, and a new generation of WOMtools for directly authoring into the WOM internal object format. This content was then "checked in" to the WOM database. Other systems, based on IBM Digital Libraries technology, generated Web pages from dynamically changing data, such as sports results.

WOMwire was the protocol used for communication among the various tools, including several batch upload tools. This protocol defines the way in which objects and their metadata are transmitted to the WOM system. The WOM database used the DB2/6000™ relational model to store the data and metadata object.

The WOMserver can access data and metadata stored in several formats. For the Atlanta WOMplex, the data and metadata were exported as flat files, called *servables*, from the DB2® database into the DFS filesystem. The DB2 export mechanism also triggered the DFS fileset replication system to push the updated files out to all servers at each WOMplex site.

Web-based applications for the Atlanta site would traditionally have been written as Common Gateway Interface (CGI) programs. However, they were written for the WOMserver in an object-oriented Perl or Java environment. They were run inside the server to avoid the overheads associated with executing CGI applications on traditional Web servers.

WOM architecture allows an executable program to generate any part of a served page;

the applications also fit within this paradigm. Detailed discussion of the WOM Network Application Platform is outside the scope of this article.

The WOMserver is *multi-homed*—it can support several distinct Web sites concurrently in the same server. This makes the WOMplex, with global load-balancing systems, a potential home for many Web sites at the same time—ideal for providing content-hosting services.

Centralized Logging

An important aspect of WOMplex operation is collecting HTTP access and error logs from the servers distributed across the world and bringing them into a central location for monitoring and analysis. The "Grim Reaper" performs this function. The Grim Reaper, a tree-structured plumbing of sockets and UNIX® pipes, streamed log records from each server to a confluence point at each site, then across the Internet to a central collection point on a machine called the *logger* (see Figure 1).

When everything is streaming nicely, the task of the Grim Reaper is straightforward. The problems occur when one of three events occurs:

- The servers produce log data faster than the local Grim Reaper can take it from them.
- Network delays or failures cause holdups in the Grim Reaper plumbing, causing a back-log of data.
- The final receiver of the logs cannot cope with the rate at which they are arriving from the Grim Reaper.

Any of these scenarios, coupled with the simple mandate that no log records can be discarded, make the Grim Reaper's job very difficult. As a result, the Reaper is a sophisticated technology. At any stage, log streams can be diverted into a file on a local disk to hold data, which for some reason, cannot be sent on. Later, when connections are restored or congestion eases, files are spooled back into the network. The Grim Reaper also can deal with requirements such as off-line transfer of batched log data (for example, late at night with less network traffic) and on-the-fly data compression for transmission across slower network connections.

After arriving at the logger machine, the log streams were transferred across a high-speed protocol Common Link Access to Workstation (CLAW) link to an 8-way CMOS 390 Squadron mainframe. The logs were then pre-processed by a series of pipes and filters, then implemented using MVS® Pipes. Individual log records were added to a table in a DB2/MVS database. From here, SQL queries could extract useful information from the logs, including hit counts and rates, online data mining using Activist, and real-time "click tracking." The raw log data was spooled to tape for later analysis.

Content Authoring Environment

The WOMwire protocol defines how authoring and site management tools communicate across the wire with the WOM database. The client part of the WOMwire protocol was implemented as an ActiveX control. The authoring tools, developed for the Windows 95 and Windows NT™ environments in Visual Basic®, C, or C++, could trivially access the WOM database by including the WOMwire control.

WOM objects can contain text, graphics, sound, video, page layout information, or executable code. These objects and their associated metadata can be checked in and out of the system, similar to a programmer's source code control system. The WOMbler created and edited content and managed metadata. WOMexplorer was used to navigate and reorganize a WOMplex site.

WOM objects can contain text, graphics, sound, video, page layout information, or executable code.

Each page on the site fits into an information taxonomy, described by a set of taxonomy tags. These tags position the page somewhere in an *information space*, which has a different hierarchy than the *data space* hierarchy imposed by the HTML links between a set of pages. This exciting new approach to Web site design is outside the scope of this article.

A Visual Basic application called the *Wu-tool classifier* was used to classify existing pages on the site. Users could classify each page into the information taxonomy by choosing from sets of predefined categories that applied to the page in question.

With the WOMpix tool, users could process images by cropping, resizing, reformatting, and labeling, before checking them into the WOM database. WOMpix dealt with many thousands of images that appeared on the site during the Olympics. It also allowed metadata, such as the source agency and photographer, to be recorded with the image. WOMpix could also generate thumb-nail images and image water-marking.

Load Distribution

Load distribution and balancing within the WOMplex uses a two-tier architecture. At each site, local load balancing distributes incoming traffic toward a single IP address across multiple backend WOMservers. Between sites, a global load-balancing system allows logical Web sites to be moved between physical server locations.

Interactive Session Support

Both tiers of the load-balancing system use software called Interactive Session Support (ISS). This is the interactive component of the RS/6000 SP LoadLeveler™ package for batch and interactive load balancing across a set of RS/6000 SP nodes or across other UNIX workstations, notably RS/6000, HP™, Sun®, and Silicon Graphics.

ISSmonitor collates agent responses, starts and stops remote agents (using inetd), and reacts appropriately if agents do not report in for an extended period of time.

Each server runs the ISSagent, which runs a user-defined metric that determines how busy the server is. The metric can be any executable program or pipeline of UNIX commands, providing it returns an integer at the end. For example, a popular choice is `ps -ef | wc -l`, which reports the number of processes currently running on the system and provides a good measure of machine activity. Far more complex metrics can be contrived, including an executable program to generate the metric value. The metric used for the Olympics counted the number of established socket connections on port 80 of each server.

ISSagents periodically report to a central ISSmonitor, which collates agent responses, starts and stops remote agents (using `inetd`), and reacts appropriately if agents do not report in for an extended period of time. The ISSmonitor also uses Internet Control Message Protocol (ICMP) ping packets to ensure that servers are still "alive" before recommending them for use. The ISSmonitor transfers a *load profile* of server activity to the WOMsprayer via a socket. The WOMsprayer uses the load profile to compile a weighted round-robin schedule for incoming traffic. Any server whose ISSagent does not respond is eliminated immediately from the load-balancing schedule to prevent incoming traffic from hitting a non-responsive server and receiving `Connection Refused` error message.

In another operational mode, ISS works as a load-balancing Domain Name Server (DNS). A generic Internet name, such as www.atlanta.olympic.org, is assigned to the cluster of servers. When a client machine or another name server in the hierarchical DNS system requests the IP address for that name, ISS ensures that the name server returns the IP address of the least heavily loaded server to provide the required service. ISS can work in DNS mode in one of two ways.

ISS can cooperate with an existing DNS, by regularly updating the "name to address mapping" database of the name server. Although this works well for low volume, long-lived connections, it is not responsive enough for high-volume, short-lived transactions usually experienced by a Web server.

ISS also operates as a high-performance, load-sensitive DNS, totally replacing the existing name server. This mode of operation can handle Web server traffic and provide a weighted round-robin mechanism for distributing high-volume load evenly across the backend servers.

ISS also offers an enhanced security, caching name server facility, described in the Security section above.

Local Load Balancing—WOMsprayer

Load-balancing software technology from IBM Watson Research has several names: TCP router, Encapsulated Cluster, WOMsprayer, ShockAbsorber, and Network Dispatcher. The *WOMsprayer*, a network router between two physical networks, offers a single IP address to the world. When a TCP connection is made to that address on the inbound network, the WOMsprayer software modifies the Medium Access Control (MAC) address field of the incoming connection request packet and transfers the packet to its outbound network. The MAC address change redefines the final packet destination, and the WOMsprayer arranges for it to be delivered to one of several backend servers.

The backend machines have unique IP addresses and also have the IP address of the WOMsprayer aliased onto their inbound network interface. Consequently, when this modified connection request packet arrives at a backend machine, it appears that the packet was destined for this server; the server does the required "accept" to create the TCP session. The WOMsprayer maintains a table of currently active connections, so any further incoming packets for that TCP session, such as packet acknowledgments, are routed to the correct backend server.

A weighted round-robin schedule determines which backend server receives an incoming session request. This schedule is constructed from the server activity profile that the ISSmonitor periodically supplies to the WOMsprayer. If ISS has not heard from a server or has reason to believe that a server is having problems, the

server is quickly excluded from the WOMsprayer schedule. No new connections are routed to that server until it rejoins the cluster.

The backend server's default route serves as the network path from the server to the client browser. This route generally does not use the same machine as the WOMsprayer. Even if the same server is used, the WOMsprayer does not intercept the returning packets.

Global Load Balancing

Two mechanisms allow Web traffic to be spread effectively across the multiple sites comprising the WOMplex: ISS-DNS and the WOMbat.

ISS-DNS

In its DNS emulation mode, ISS allows the DNS administrator to configure a table of Internet domain names to IP address mappings. This differs from an ordinary DNS in that more than one IP address may be associated with any given Internet name. An ISSagent runs on each server that is designated to serve any given name. According to the health, load, and the configured balancing policy for that name, incoming resolution requests for that name receive the IP address of the most appropriate server.

In the WOMplex, the ISSagents run on the machine that hosts the WOMsprayer for each site. Two balancing policies can be configured: Round and Fail.

Round. This policy does a weighted round-robin of the IP addresses in the list associated with that name. It is distributed according to two factors:

- The server activity profile, as determined from the input provided by the ISSagents
- A server weighting factor, allowing for different server capacity at each site—different numbers of WOMserver nodes in the RS/6000 SPs at each site, in the case of the WOMplex

Fail. This attempts to use the first IP address in the list—the "preferred" site—until that site becomes unavailable. Then, ISS switches to a weighted round-robin of the remaining IP addresses in the list, the same as in the Round policy. This system allows a single, logical Web site (defined by a specific Internet name) to be distributed across multiple physical locations, which could be a cluster of servers controlled by a WOMsprayer. Adjustments to the system configuration can be made dynamically, allowing extra servers to be switched-in, if necessary, to cope with excess load. The system enables automatic fail-over to other servers if a server experiences downtime, which can be fail-over to other servers from a preferred or primary server or omission of unresponsive servers from the round-robin schedule.

WOMbat

WOMbat uses ping to "echo-locate" a client machine and determine the nearest WOMplex site in network round-trip terms. With this knowledge, WOMserver can then divert the client to its nearest server site, and hopefully receive better service, thanks to the better network connectivity to that site.

When a client browser visits a WOMplex site for the first time, the server knows it has not seen that client before, because the client does not have a browser *Cookie* for that site. The server generates a Cookie, a unique key to identify that client in the future, and sends it to the browser to use whenever it visits that site. The Cookie is also stored in the Global Profile Daemon (GPD), a database of profile information, keyed by browser Cookie. The server also invokes— asynchronously in the background—the WOMbat Driver before composing the HTML page requested by the client and returning it to the client.

The WOMbat Driver sends details of the client's IP address and its browser Cookie to a WOMbat daemon at each site in the WOMplex. The WOMbat daemon sends a volley of ICMP echo request packets (ping) to the client IP address and measures the round-trip time of any responses that return within a pre-specified time-out period.

WOMbat daemons send back to the WOMbat Driver the average network response time taken to ping the client machine. The WOMbat Driver waits for a specified length of time for the WOMbat daemons to respond, then determines which server is closest to the client by comparing the responses. Server availability information also can be factored into this calculation, so that the closest server might be rejected in favor of a more distant, but less busy server. The user's profile in the GPD contains the identity of the preferred server.

The next time the user requests a page from the Web server, the server daemon interrogates the GPD using the browser Cookie as a key, and discovers that the preferred server for that client is not this server. The WOMserver then builds the requested page for the client, but rewrites all embedded links to follow-on pages and to

In its DNS emulation mode, ISS associates more than one IP address with any given Internet name.

embedded images as URLs on the preferred server rather than on local links. For example, if the preferred server was www.hursley.atlanta.olympic.org and the local link would normally have been the following:

```
<IMG SRC="/data/image1.gif">
```

then, the link sent back to the client is rewritten as follows:

```
<IMG SRC="http://www.hursley.atlanta.
olympic.org/data/image1.gif">
```

When the client browser follows any of these links, they are retrieved from the preferred server. From that point on, all links will point back to that server, so the client has effectively been "moved" to that server.

WOMplex separates logical services from physical servers, which allows a Web site to be hosted from any site in the WOMplex.

Virtual Addressing

AIX and other UNIX operating systems can alias multiple IP addresses onto a single network interface; that is, the server can "pretend" to have several different IP addresses on the same network at the same time. This was used two ways in the Atlanta WOMplex: multi-homing and service virtualization.

Multi-Homing. A single WOMserver instance can serve several Web sites. The IP site addresses are aliased onto the server's Internet network interface. The WOMserver configuration has separate "trees" of URL data for each hosted address. When a Web page request arrives, the WOMserver interrogates the socket connection to identify which IP address it was sent to, then serves the appropriate page from the correct data tree. Many "standard" Web servers consider this a standard feature, such as Virtual Hosting in Apache.

Service Virtualization. WOMplex separates logical services from physical servers, which allows a Web site to be hosted from any site in the WOMplex. IP aliasing within a site separates logical from physical and frees any dependence on specific hardware. All externally visible IP addresses are aliases, not the "real" base IP address of any machine. Therefore, if a server fails, the service IP addresses supported by the failed server can be transferred readily to another machine. One additional requirement, apart from adding the IP address aliases, is to flush the arp cache on the nearest network router, which forces it to reestablish the new IP to MAC address mappings. The System Network Management Protocol (SNMP) interface to the router can often automate this operation.

Global Profile Daemon

The Global Profile Daemon stores and caches client profile information, using a client browser Cookie as a retrieval key. Any WOM application or the WOMserver itself can store profile information in GPD. Profile data is held as a file of lines containing `name = value` pairs. Keys used for the Atlanta WOMplex were constructed from the concatenation of the following:

- Three randomly chosen letters
- Cookie creation timestamp (seconds since Epoch, plus an offset)
- Cyclic Redundancy Check (CRC) checksum digits

GPD uses a multi-level cache to balance the global distribution requirements of profile data and fast access to user profiles by the WOMserver. Closest to the WOMserver, GPD maintains an in-memory cache with a Least Recently Used (LRU) policy; therefore, people who are known to be currently moving around the Web site are most likely to be in memory.

To locate profiles efficiently, the directory structure has directory names 'A' through 'Z'—three levels deep. For example, the profile with the key ADF1234567123 would be found in the file with the following full path: /dfs/profiles/A/D/F/3217654321FDA.

Note that the name given to the profile file was the reverse of the key string, that is, 3217654321FDA. This is because DFS uses its own hashing algorithm for efficient searching of directory contents—based on the first character of the filename. Consequently, the first character of the filename should be a random character. This avoids having all files in a directory starting with the same first character, and thus all falling into the same hash bucket. We chose three directory levels and 26 entries per level, knowing that DFS caching is optimized for a certain maximum directory content size.

At the next level, DFS caches recently retrieved profiles on the local disk of the server machine. At a site level, recently accessed profiles are stored in a DFS fileset that physically resides on disks at that site. This avoids trans-Atlantic connections for many profiles. Finally, the main DFS server at the primary WOMplex site in Southbury stores the global GPD repository.

When a client comes back to the Web site (at any physical location) with a previously assigned Cookie, the WOMserver requests information on that client's profile from its local (on the same machine) GPD instance. GPD checks the in-memory LRU cache and returns the information if it is there. If not, GPD must fetch the profile from the global repository because that client has not been to this physical site recently.

GPD reads the profile from the global DFS directory and copies the file to a local DFS fileset located physically on the local site. Then GPD sends a UDP broadcast to other servers at its local site informing them that the client with a specified Cookie has come to the site, and may soon request pages from them. Other GPD instances at that site receive the broadcast and go to the local DFS fileset to read the profile information into the memory cache. Thus, only the first GPD at a site has to do the trans-Atlantic journey to get the profile data of a returning client. The other servers (who are likely to receive requests from that client in the near future, thanks to the WOMsprayer) pick it up from a fileset that is local to their physical site.

The CRC digits are built into the Cookie. This makes it more difficult for anyone to tamper with a Cookie while trying to obtain profile information about another user by reverse engineering their Cookie. Cookies with invalid checksums are not accepted by the WOM system.

Applications

This section describes applications that were designed and built for the Atlanta Web site. For various reasons, not all of these went into full production on the site. They were all working in the lab, so we have included them here for the sake of completeness.

Sneak Peek

Sneak Peek was the largest "Net Cam" ever. Live video feeds from 38 locations at the Atlanta Games were fed to "The Fishbowl" in Atlanta. Here they went through a bank of "frame grabbers" and were checked into the WOM system. Images were captured approximately every five seconds from each feed. Because of the high volatility of this data, the images were not stored into the WOM DB2 database, but went directly into DFS. HTML pages to render the latest set of images and to navigate between Olympic venues were generated on-the-fly to pick up the most recent image file names.

The human operator in the Fishbowl received all feeds concurrently and could select channels that currently had activity on them. The navigation options reflected the choices presented by the dynamic HTML pages in the Sneak Peek section of the site. The operator could choose the most stunning images and build a daily "scrapbook" of images worth keeping. These images were titled, cropped, assigned metadata using the WOMpix tool, and checked into the main WOM database.

Sneak Peek images were not mirrored across the world because of their highly volatile nature—they were all served from the primary WOMplex site at Southbury. This became an extremely popular application, and many hundreds of thousands of images were downloaded.

Bamba

Bamba is IBM's streaming audio and video technology for the Web based on G.723 low bit-rate streaming standards. For more information about the technology and to download the decoders and encoders, visit the Bamba home page on the IBM AlphaWorks™ Web site (http://www.alphaworks.ibm.com/).

Bamba has three advantages over other Internet audio and video formats:

- Data is streamed, so you can view and download it simultaneously, avoiding huge waits while the whole file downloads.
- The content is encoded at a low bit rate, making it suitable for transfer over a standard 28.8 Kbit modem.
- G.723 is a standard.

Three forms of Bamba content were used at the Atlanta Olympics:

- **Audio clips:** Recordings of many interviews with athletes, Olympics organizers, and news reports. A regular Web server handles Bamba audio clips in the same way as GIF images. On the Atlanta site, they were one data type served by the binary server.
- **Video clips:** Encoded highlights of the sports events. Bamba video clips are also regular binary content that can be served by an ordinary Web server. They were served from the binary server during the Olympics.
- **Live audio:** Atlanta local radio station, WGST Atlanta, was broadcast over the Internet 24 hours a day during the Olympics. Bamba

Sneak Peek images were all served from the primary WOMplex site at Southbury.

Reflector allowed many people to listen to the radio station concurrently.

Bamba Reflector

In typical use, a small Pentium™ PC would do real-time Bamba audio encoding. It is not realistic to have hundreds or thousands of clients connecting to this machine to listen to the audio stream, so Bamba Reflector was created. The Reflector connects to the transmitter as the only client to that machine, it then accepts connections from Bamba clients. Whatever the Reflector receives from the transmitter is transferred to the clients. There must be a certain amount of buffering and flow-control to deal with clients connected over different speed links. The Reflector intelligently drops Bamba packets without losing stream synchronization for any clients who fall too far behind in receiving Bamba packets. Each Reflector can support many hundreds of client connections concurrently. One Reflector can act as a source to other reflectors, so a whole world-wide tree of Reflectors can potentially serve many thousands of clients across the globe.

The Reflector configuration can limit connections to a specified upper limit or it can be offline—not broadcasting at the moment. When a connection is not possible, it can send out an information message in one of three forms: a pre-recorded audio message built into the client, a text message sent from the Reflector, or an audio message sent from the Reflector. Various combinations are also possible.

ISS load balancing can be used in front of a bank of Bamba Reflectors to balance incoming connection requests to a single advertised Internet name across the available servers to ensure an even load across the Reflectors.

WOMplex Boiler Room

The Boiler Room monitoring system allowed statistics to be gathered from various parts of the WOMplex, its applications, and subsystems. The system structure is described below.

Each application to be monitored had a `funneld` (`funnel` daemon) client. To make this a simple task, developers had client code for Java, Perl, and C available. The protocol used throughout this system was Internet Relay Chat (IRC), transmitting data in a textual format with each monitored system using a different IRC channel number. The client code connected to a `funneld` instance ran on each server. The `funneld` instances connected to other `funnel` daemons in a tree structure. An IRC server was at the root of the tree of `funnel` daemons.

The Boiler Room monitors were Java applets embedded in HTML pages on the WOMplex Web site. These applets used IRC client classes written in Java that connected back to the IRC server. The monitor applications then tuned in to the IRC channels required to construct their displays and waited for information to arrive. A variety of dials, counters, and indicators were implemented as Java classes that could be combined in various ways to display the statistics for any system.

Since the Boiler Room system could also tail log files and receive data through sockets and Interprocess Communications (IPC) message queues, it was easy to hook up monitors for almost anything. A special *chart recorder* client logged IRC channel data for subsequent analysis.

Information Taxonomy

Early in this project we realized that a very large site needs some kind of information taxonomy, or hierarchy, mapped to the site; otherwise, it becomes completely unmanageable and almost impossible to navigate. This important issue required the skills of an information architect.

The metadata for every page, image, and multimedia clip checked into the WOM database included a set of category tags. These tags position the object in the *information space* of the Web site. This information space is dimensionally exclusive from the *data space* imposed by the HTML hyperlinks built into the documents. Tools, such as the Wu-tool, categorize pages on the site. Once the whole site has been categorized, facilities can be provided to navigate the site in information space, as well as the traditional HTML navigation. For example, someone looking at pages about Men's Canoeing might also be interested in other pages that are "close" in the information space. For example, in this case, Kayak events might be of interest.

As an interesting new area for Web site navigation and management, this will undoubtedly play an important role in the future.

Personal Home Page

The combination of generating dynamic pages in the WOMserver and identifying individual clients from their browser Cookie allowed a personal home page to be created for each visitor to the site.

Users could specify to the WOMserver the specific sports, countries, and athletes of interest to follow as the Games progressed. If they provided

Users could specify to the WOMserver the specific sports, countries, and athletes of interest to follow as the Games progressed.

this information, they could then request their personal home page (/myhome.html) from the WOMserver. The Home Page application would build a personalized page for that user, including highlights of the day for the sports, countries, and athletes of interest. This was coupled with Activist, which could suggest links of interest for the user, based on previous browsing history.

Activist

Activist is an online Web log data mining tool. Activist was a DB2 client application connected to the DB2 server running on the CMOS 390 parallel sysplex. This machine was amassing HTTP log data from all the servers in the WOMplex via the Grim Reaper. By following the click-track of each unique client (based on their browser Cookie), Activist would compile a "persona" for that client. Once this was established, it was possible to ask questions of Activist, such as "Based on my browsing patterns in the site so far, show me some links that I might find of interest that I haven't seen yet," or "Show me some links that I might like to see, based on where others like me have been."

The phrase "like me" in this context is derived from a data mining technique called *segmentation*. Segmentation allows a data set to be partitioned into several clusters of individuals who show a statistically significant similarity in certain areas of the data. In this application, if you had looked at Canoeing and Cycling, and many others had looked at Canoeing, Cycling, and Basketball, then Activist would probably offer you some Basketball links to view.

This powerful technology will prove useful in making sense of the vast amounts of log data accumulated by every Web server in the world.

Country and Sport Pages

As information such as results, news reports, news articles, news wire feeds, and photographs flowed into the site during the Games and checked into the WOM database, each page was categorized into its appropriate position in the site's information taxonomy. Every few hours, a batch job was run to compile a home page for every country participating in the games and for each sport. This home page contained links to everything recently added to the site relating to that country or sport. This provided an easy way to follow the progress of a particular country or sport and to catch up on recent happenings in that area.

Guestbook, Search, Feedback

For the Atlanta site, these standard applications were built in the WOMserver application environment and run as dynamic extensions to the server. The Guestbook and Feedback applications made a client/server connection to a remote machine running DB2 that stored the information submitted by the user. Other applications provided a password-protected Web interface to these databases to allow the information to be extracted and analyzed.

Postcards

The Postcards application allowed users to "send a postcard" of any image on the Atlanta site to someone else on the Internet using E-mail. When a full-size version of an image was requested, usually by clicking on its thumb-nail image, one option on the navigation bar of the page containing the full-size image was to "Send a Postcard."

With this option, the user could specify the recipient's E-mail address, a message to go with the Postcard, and the sender's identity. The recipient would then receive an E-mail message indicating that a "Postcard from Atlanta" was waiting. By connecting to the URL given in the E-mail message, the recipient would go to a page that was generated dynamically, based on a unique identifier that was part of the Postcard URL. The page would contain the selected image, the message from the sender, and the sender's name.

Eye on the Olympics

"Eye on the Olympics," a Java applet, provided a way to "keep an eye" on what was happening at the Olympics, without having to continuously surf around the site. The applet used IRC to connect to a server at the WOMplex site. It presented a set of icons: one for each sporting discipline. The client displayed a moving ticker-tape of news flashes. This ticker-tape had an area that could be reached by clicking on a sports icon, which contained results and news stories relating to that sport.

The server used the metaphor of a television station with several "programs"—one per discipline—each created by an automated producer. The producer's job was to choose pieces of information being fed into the Web site and assemble them in a form for presentation in the Eye on the Olympics. The server sent out a *play list* of current articles, then sent new articles that had been added since the last play list was

Segmentation allows a data set to be partitioned into several clusters of individuals who show a statistically significant similarity in certain areas of the data.

broadcast. The client applet received the play list and modified its display behavior accordingly, adding new items as they were transmitted and deleting old items that were no longer in the play list. The producers relied heavily on the information taxonomy to determine which new items being checked into the site were of interest to their programs.

The client applet used HTTP to retrieve non-text items, such as images, from the main Web site, using a URL sent over IRC. The applet also used HTTP to retrieve a file containing the initial set of current content when the applet was first invoked.

Conclusion

The Atlanta Olympics Web site was the largest Web site in the world and handled an immense amount of traffic during the 1996 Summer Games. The WOMplex technology was pioneered to support large sporting events on the Internet. The Olympics gave the WOMble Team a chance to push forward the technology frontiers, and has paved the way for products using WOM technology in the future.

The WOMserver was designed as an Internet application platform, of which a sporting event server was just an application. The suitability of this architecture as an application environment has been proven in the "trial by fire" experienced during the Olympics.

This experience greatly advanced load balancing and global scalability. The systems deployed for the Olympics showed that a Global WOMplex is a viable concept.

The author gratefully acknowledges the contributions of Sean Martin and David Grossman to this article, and also wishes to thank the Team at the IBM Southbury lab.

Andy Stanford-Clark, *IBM Corporation, 150 Kettletown Road, Southbury, CT 06488. Internet: andysc@vnet.ibm.com. Dr. Stanford-Clark is an Internet advanced technology consultant in IBM's Internet Division. His responsibilities include exploring emerging Internet technologies and working with customers to implement Internet-based projects. Dr. Stanford-Clark is currently in the U.S. on international assignment from IBM Hursley, U.K. He has a BS in Computing and Mathematics and a PhD in Parallel Computing from the University of East Anglia in the U.K.*

Message Passing Interface for RS/6000 SP

By Xianneng Shen, Eddie Ho, and Mike Hammill

Message Passing Interface (MPI) provides an efficient, portable, standardized interface to implement parallel programs for RS/6000 SP. This is a different programming model than the single OS environment for AIX. This article covers the basic concepts of MPI, provides some sample programs, and discusses MPI point-to-point and collective communication routines.

An RS/6000® Scalable POWERParallel™ (SP) system, a scalable parallel computer, is the high-end system of the RS/6000 product family, which ranges from laptop, desktop, workgroup server, enterprise server, to super computer using the share-nothing architecture. AIX® is the operating system used across the entire product family.

AIX provides binary compatibility for applications running on a single SP node and applications running on the rest of the RS/6000 family. The SP is a shared-nothing system of separated, full-function nodes connected by a high-performance switching network. Each node is a stand-alone RS/6000 workstation running AIX. An explicit message-passing mechanism allows communication between nodes.

The SP provides both traditional Internet Protocol (IP) and user space communication protocols. Because the user application has direct access to the SP switch adapter fabric, the user space protocol is very fast. Currently only one user application process is allowed per processor at one time in the user space protocol. A Message Passing Interface (MPI) code can be run using either IP or user space protocol on a set of uniprocessor nodes since the Symmetric Multiprocessor (SMP) node does not yet have MPI support. In this article, the terms "processor" and "node" are interchangeable.

MPI Overview

MPI is a message-passing interface that is formally specified, portable, and based on standards. The primary objective of MPI is to give programmers an efficient, portable, standard message-passing library with rich functionality. The MPI standard, which uses the most attractive features of several existing message-passing systems, was introduced at Supercomputing '93 and finalized in May 1994. Software developers have now begun implementing this standard on many platforms.

There are many reasons to use MPI, but the four most important considerations are standardization, portability, performance, and richness.

Figure 1 illustrates the basic node-to-node communication paradigm.

Basic MPI Programming and Concepts

As in other message-passing environments, you can insert calls to MPI routines in either C or FORTRAN code. Six basic MPI routines can be found in most MPI code:

- Initialize MPI by calling `MPI_INIT` at the beginning of an application
- Terminate MPI by calling `MPI_FINALIZE` at the end of the application
- Identify a particular process in your application by calling `MPI_COMM_RANK`
- Determine the total number of processors for the application by calling `MPI_COMM_SIZE`
- Send messages by calling `MPI_SEND`
- Receive messages by calling `MPI_RECV`

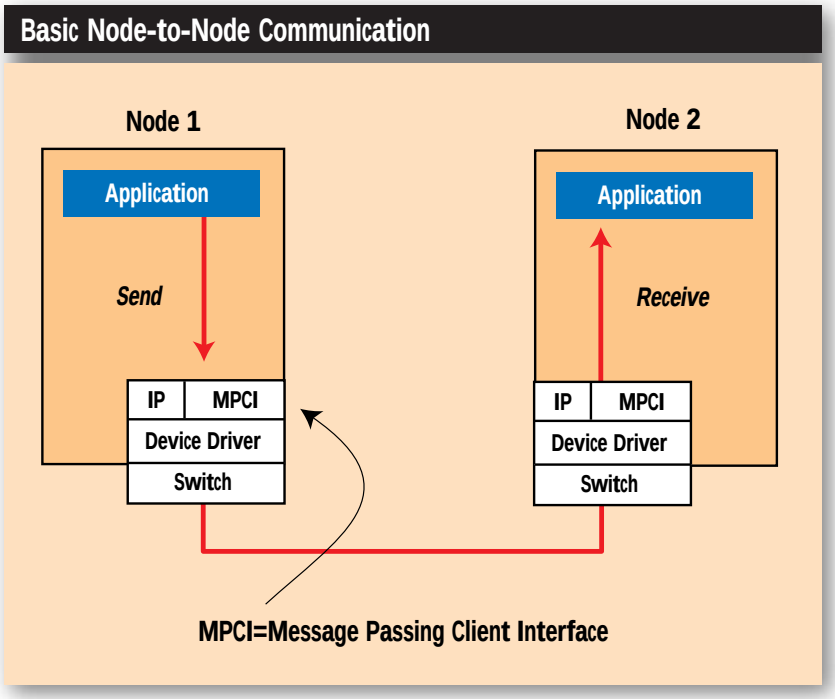


Figure 1. Node-to-node communication

```
#include <stddef.h>
#include <stdlib.h>
#include "mpi.h"
main(int argc, char **argv )
{
        char message[20];
        int i,rank, size, type=99;
        MPI_Status status;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        if(rank == 0) {
           strcpy(message, "Hello, world");
           for (I=1; i<size; I++)
           MPI_Send(message, 13, MPI_CHAR, I,
               type, MPI_COMM_WORLD);
        }
        else
          MPI_Recv(message, 20, MPI_CHAR, 0,
               type, MPI_COMM_WORLD, &status);

        printf( "Message from node =%d : %.13s\n", rank,message);
        MPI_Finalize();
}
```

Figure 2. Basic MPI routines

Figure 1 shows the six basic MPI routines described above.

In Figure 2, process zero (rank == 0) sends a message to the rest of the processes (rank != 0) in the communication world (described below). After receiving the message sent by processor 0, each processor prints its rank and the message `hello world`.

This simple example illustrates some fundamental MPI concepts including the MPI message. An MPI message contains the message and the message envelope. The message consists of the data, count, and datatype; the message envelope includes source (or destination) process rank (integer), message tag (integer), and communicator.

Communication context and the process group represent the minimum information found in a communicator. The communication context can be viewed as a hidden system tag for extra protection. The process group is a set of processes whose members are identified by their rank.

MPI Point-to-Point Communication

The most elementary form of message-passing communication involves two processes: one passing a message to the other. Although there are several ways that this might happen in hardware, logically, the communication is point-to-point: one process calls a `send` routine and the other calls a `receive` routine.

MPI Blocking Send / Receive

As with most existing message-passing systems today, MPI provides blocking as well as nonblocking send and receive. Figure 3 shows a simple send and receive flow of large messages between two nodes. Figure 4 shows the programming syntax of blocking `send`.

The blocking `send` call does not return until the message has been safely stored away so that the sender can freely reuse the send buffer. Figure 5 shows the syntax of the blocking `receive` routine.

The blocking `receive` does not return until the message has been stored in the receive buffer.

Nonblocking Send and Receive

Performance on many systems can be improved by overlapping communication and computation. One

way to achieve that is to use nonblocking communication.

A nonblocking `send` call initiates the send operation, but does not complete it. The nonblocking `send race` call will return before the message is copied from the send buffer. A separate call `mpi_wait` or `mpi_test` is needed to complete the communication—to verify that the data was copied from the send buffer.

Figure 6 shows an example message flow for nonblocking communications. Figure 7 shows the nonblocking send syntax.

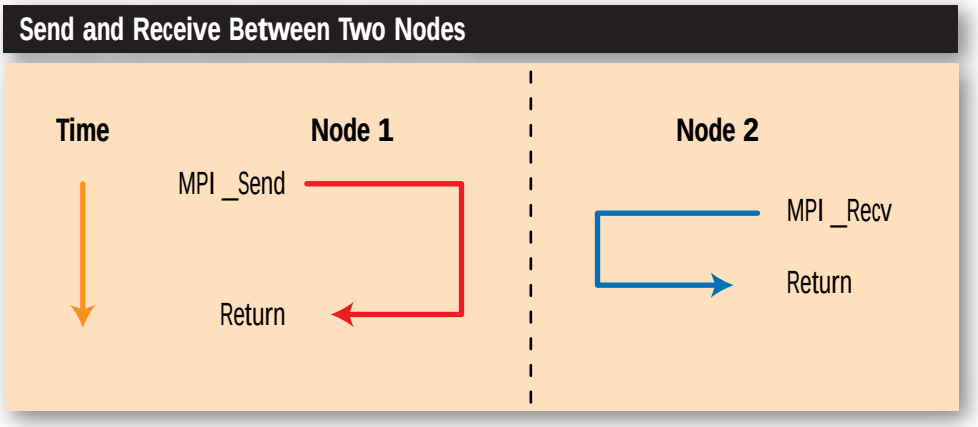


Figure 3. Send and receive between two nodes

```
MPI_Send(void* buf, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm);
```

`buf`	initial address of the send buffer (IN)
`count`	number of items in the send buffer (IN)
`datatype`	datatype of each send buffer item (handle) (IN)
`dest`	rank of the destination process in comm (IN)
`tag`	message tag (integer) (IN)
`comm`	communicator (handle) (IN)

Figure 4. Blocking send syntax

```
MPI_Recv(void* buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm,
        MPI_Status *status);
```

`buf`	initial address of the receive buffer (OUT)
`count`	number of items to be received (integer) (IN)
`datatype`	datatype of each receive buffer item (handle) (IN)
`source`	rank of the source process in comm or `MPI_ANY_SOURCE` (integer) (IN)
`tag`	message tag or `MPI_ANY_TAG` (integer) (IN)
`comm`	communicator (handle) (IN)
`status`	status object (status) (OUT)

Figure 5. Blocking receive syntax

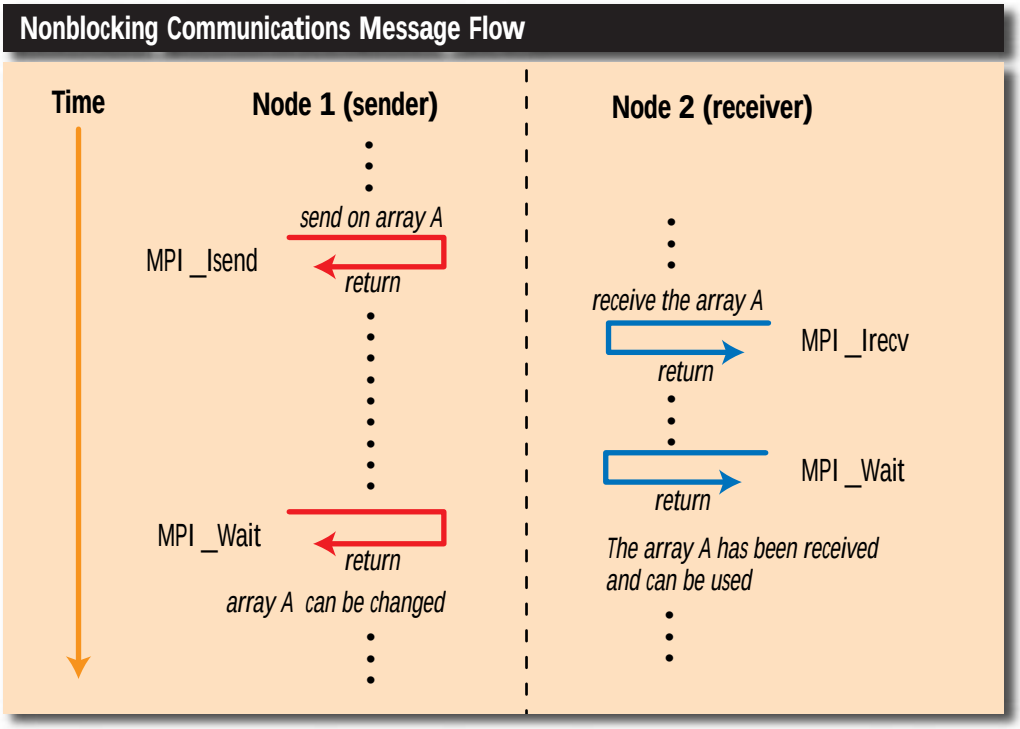


Figure 6. Nonblocking message flow

```
MPI_Isend(void* buf, int count, MPI_Datatype datatype,
     int dest, int tag, MPI_Comm comm,
     MPI_Request *request);
```

buf	initial address of the send buffer (IN)
count	number of items in the send buffer (IN)
datatype	datatype of each send buffer item (handle) (IN)
dest	rank of the destination process in comm (IN)
tag	message tag (integer) (IN)
comm	communicator (handle) (IN)
request	communication request (handle) (OUT)

Figure 7. Nonblocking send syntax

```
MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm,
        MPI_Request *request);
```

buf	initial address of the receive buffer (OUT)
count	number of items to be received (integer) (IN)
datatype	datatype of each receive buffer item (handle) (IN)
source	rank of the source process in comm or MPI_ANY_SOURCE (integer) (IN)
tag	message tag or MPI_ANY_TAG (integer) (IN)
comm	communicator (handle) (IN)

Figure 8. Nonblocking receive syntax

If the nonblocking `send` is called, the system may start copying data from the send buffer. The sender should not access any part of the send buffer after a nonblocking send operation is called until `send` completes.

Similarly, a nonblocking `receive` call initiates the receive operation, but does not complete it. The call will return before a message is stored in the receive buffer. An `MPI_Wait` call must complete the receive operation. Figure 8 shows the nonblocking receive syntax.

The receive buffer stores count consecutive elements of the type specified by datatype, starting at the address in buf. The length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, in the receive buffer.

To check the status of a nonblocking `send` or `receive`, call `MPI_Test` or to complete a nonblocking send or receive, call `MPI_Wait`.

Collective Communication

Collective communication involves all the group processes. You can build your own collective communication routines, but it may involve a lot of work and may not be efficient. Although other message-passing libraries provide some collective communication calls, none provides a set of collective communication routines as complete and robust as those provided by MPI.

MPI collective communication can be divided into three subsets:

- Barrier synchronizations
- Data movements
- Reduction computation

Barrier Synchronizations

In almost all parallel applications, explicit or implicit synchronization is required. As with other message-passing libraries, MPI provides a function call, `MPI_Barrier` (`MPI_Comm comm`), to synchronize all processes within a communicator.

Data Movements

MPI provides a set of useful data movement routines, such as `broadcast`, `gather`, `scatter`, and `alltoall`. In many cases, one process needs to send (`broadcast`) some data to all the processes under the same communicator. Each process must call `MPI_Bcast` (`void* buf`, `int count`, `MPI_Datatype datatype`, `int root`, `MPI_Comm comm`), where the process sending the data is called root. Figure 9 shows a broadcast operation for a 4-node system.

If you have an array scattered throughout all processes in the group, and you want to collect each piece of the array into a specified process, the function to use is `GATHER`. MPI provides `MPI_GATHER`. The `MPI_SCATTER` is the counterpart of `MPI_GATHER`. Figure 10 shows the scatter and gather operation; Figure 11 shows the syntax.

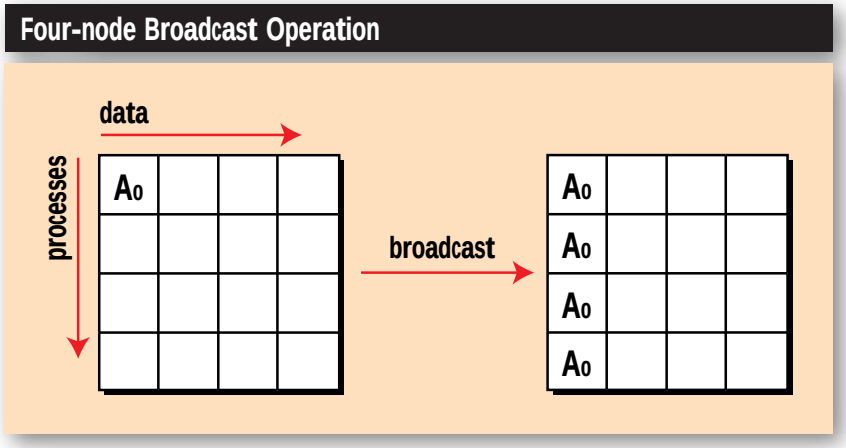


Figure 9. Broadcast operation for a 4-node system

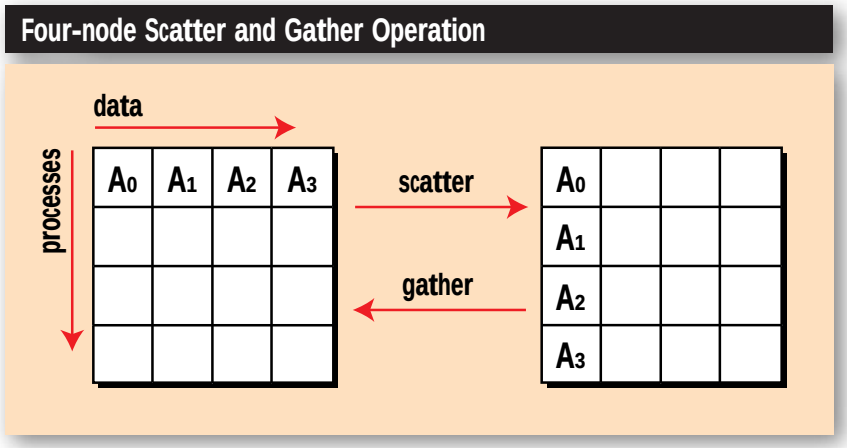


Figure 10. Scatter and gather operation for a 4-node system with four data types

```
int MPI_Gather(void* sbuf, int scount, MPI_Datatype sdatatype, void* rbuf,
   int rcount, MPI_Datatype rdatatype, int root, MPI_Comm comm)

int MPI_Scatter(void* sbuf, int scount, MPI_Datatype sdatatype, void* rbuf,
   int rcount, MPI_Datatype rdatatype, int root, MPI_Comm comm)
```

Figure 11. Scatter and gather syntax

MPI provides a vector version of MPI_Gather and MPI_Scatter routines: MPI_Gatherv and MPI_Scatterv.

An MPI_ALLTOALL call is helpful in applications such as matrix transpose and FFT. Figure 13 shows its syntax.

Reduction Routines

One of the most useful collective operations is global reductions, or combine operations. The partial result in each process in the group is combined into one specified process or all the processes using some desired basic functions. Figure 14 shows the form of each of the two global reduction primitives.

MPI has predefined operations, such as MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, and MPI_MINLOC. See the MPI standard documentation for these operations.

A scan, or prefix-reduction operation, performs partial reductions on distributed data. The scan operation returns the reduction of the data in the send buffers of nodes ranked 0,1,2,...,*n* in the receive buffer of the node ranked *n*. Figure 15 shows the syntax.

Conclusion

The message-passing programming model closely matches the SP shared-nothing architecture. The

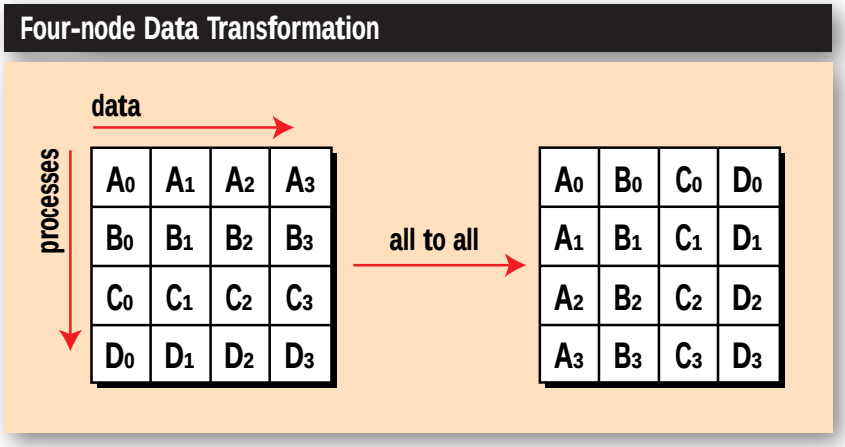


Figure 12. Data transformation on a 4-node system with multiple data types

```
int MPI_Alltoall(void* sbuf, int scount, MPI_Datatype sdatatype,
   void* rbuf, int rcount, MPI_Datatype rdatatype, MPI_Comm comm)
```

Figure 13. Syntax for a data transformation

```
int MPI_Reduce(void* sbuf, void* rbuf, int count, MPI_Datatype datatype,
 MPI_Op op, int root, MPI_Comm comm)

int MPI_Allreduce(void* sbuf, void* rbuf, int count, MPI_Datatype datatype,
   MPI_Op op, MPI_Comm comm)
```

Figure 14. Syntax for a prefix-reduction operation

```
 int MPI_Scan(void* sbuf, void* rbuf, int count, MPI_Datatype
    datatype, MPI_Op op, MPI_Comm comm)
```

Figure 15. MPI scan

message-passing program running on an SP system can achieve the highest possible performance. MPI becomes the choice of the message-passing library because of its standardization, portability, high performance, and rich functionally. Moreover, MPI continues to be improved as more functions are added. However, MPI will be backward compatible, which will enable an MPI application program developed using MPI today to run on an MPI implementation in the future.

Reference

IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference Version 2, Release 1 (GC23-3894-00).

Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*. June 12, 1995.

IBM Parallel Environment for AIX: Operation and Use Version 2.1.0. (GC23-3891-00).

Gropp, William; Lusk, Ewing; and Skjellum, Anthony. *Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

Xianneng Shen, *IBM Corporation, 522 South Road, Poughkeepsie, NY 12601. E-mail: xshen@vnet.ibm.com. Dr. Shen is a senior marketing support representative in the RS/6000 Executive Briefing Center. He has a BS and an MS in Electrical Engineering from the University of Electronic Science and Technology of China, an MS in Computer Engineering from Syracuse University, and PhD in Electrical Engineering from Syracuse University.*

Eddie Ho, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. E-mail: eddieho@vnet.ibm.com. Mr. Ho is a programming consultant in the RS/6000 Executive Briefing Center. He has a BS in Computer Science from the University of Wisconsin and an MS in Computer Science from North Dakota State University.*

Mike Hammill, *Cornell Theory Center, Ithaca, NY 14853. E-mail: mhammill@tc.cornell.edu. Mr. Hammill is the academic outreach coordinator. Mr. Hammill has a BS in Computer Science, a BA in Mathematics, and a BA in Philosophy from the University of Iowa.*

Internet Multimedia Server—VideoCharger

By Eddie Ho and Sam Juliano

Until recently, the World Wide Web was novel and entertaining, but far from dynamic. Now, thanks to a rapidly growing group of multimedia-enabling tools, such as Java, ActiveX, Shockwave, Quicktime, and video streaming, the excitement of surfing the Web is finally picking up its pace. For many interactive, content-rich environments, the scalable VideoCharger for AIX can deliver end-to-end multimedia content to the desktop using the open Web infrastructure. This article details the emerging technology.

Video and audio are the most natural form of human communications for concepts and ideas. In our society we are constantly surrounded by multiple media types, each with the intent of delivering the strongest messages to capture our mindshare.

Starting at home, television is the information delivery system. TV is a broadband, one-way broadcast system that can deliver high-quality video, but it does not provide interaction. In the business environment, most commercial applications are non-graphic and optimized for fast data transfer in the transactional environment. Today's application designer rarely considers multiple datatypes because of the programming environment and network constraints.

The World Wide Web is the emerging hypermedia information retrieval system with built-in support for multiple datatypes. This infrastructure enables the flow of multimedia content to both business and consumer environments. The Internet frontier has gradually extended to accommodate audio and video.

Internet Multimedia Challenges

Multiple requirements make video delivery across digital networks extremely challenging. The challenge is compounded by many network providers in the Internet industry, each with different levels of capability and support. Figure 1 provides a reality check in today's combined Internet and intranet environments. It shows a typical data-intensive environment where different business units access each other's information.

The availability of multimedia on the Internet poses several challenges:

Network bandwidth. Digital video requires high-speed throughput to produce acceptable quality. Most current LAN architectures were not designed to handle multiple concurrent users in a video environment. This problem is exacerbated when the client PC is attached through a slow modem connection. The bitmap-intensive pages have converted the World Wide Web into the "World Wide Waiting" room for many.

Isochronous media delivery. Digital video is sensitive to timing and cannot tolerate any jitter or latency. Although video components of a stream can tolerate some data error and loss, the audio stream is less tolerant of data irregularities. Dropped packets and jitter can be distracting and render audio totally unintelligible in the worst case.

Filesystem infrastructure. Digital video consumes great quantities of storage space. For example, a digital video file in MPEG-1 format recorded at 1.2 Mbits per second will require 1.2 GB storage for a two-hour movie. Most UNIX® filesystems are not designed to manage and stream many large files.

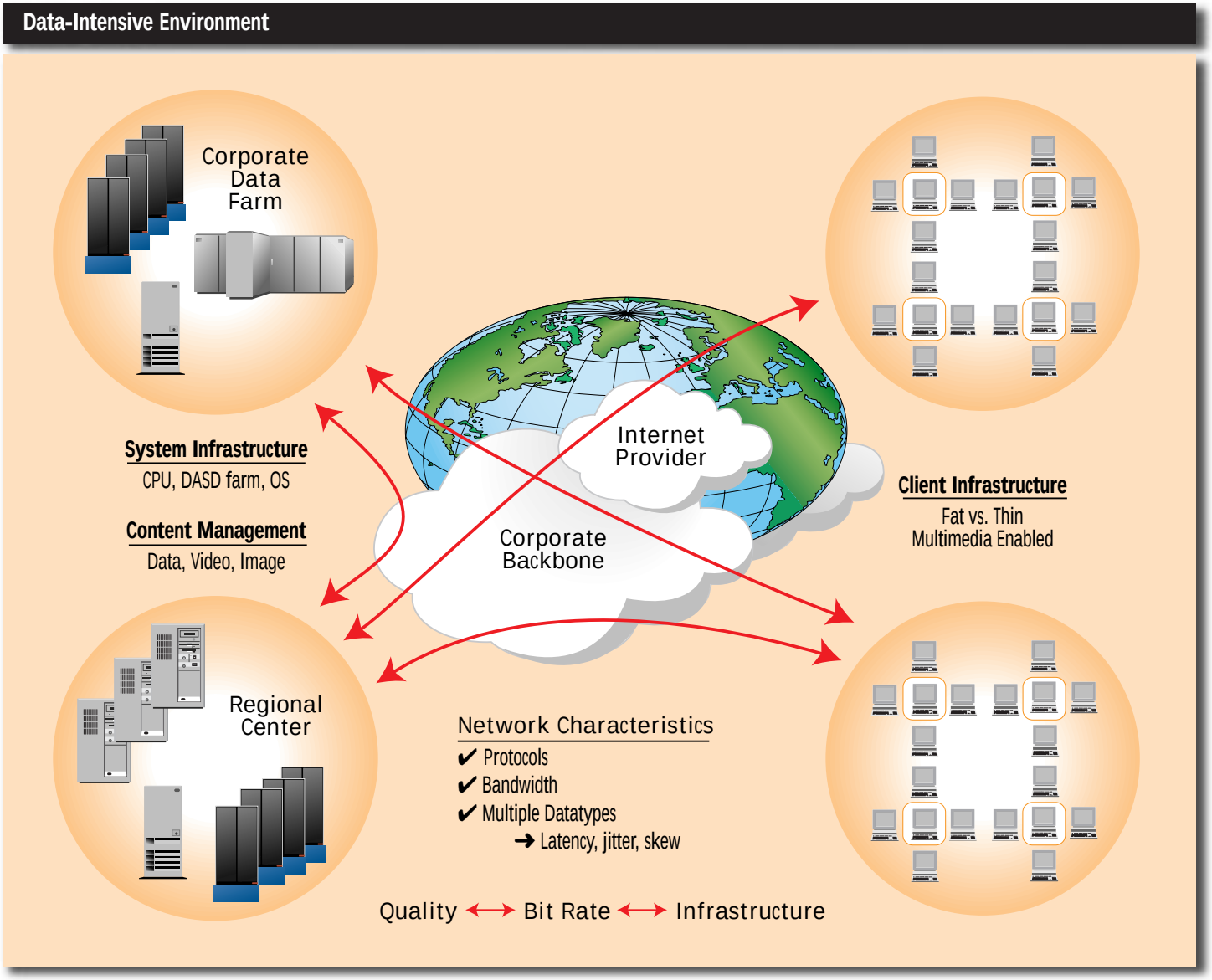


Figure 1. Typical data-intensive environment

Client considerations. MPEG decoder performance plays a key role in video quality because the stream is pushed from the server and the client decoder is required to keep up in a constant environment.

VideoCharger Server Solutions

The VideoCharger for AIX software is designed to deliver networked video and audio to Internet/intranet clients. This allows businesses to enhance their marketing on the Web by adding video to their Web pages for product demonstrations. The video content is delivered in real-time, thereby eliminating the need to download or save the file before video and audio are played. The delivery is flexible and adjustable to the connection speed of the client.

The combination of Low Bit Rate (LBR), Mid-band, and High Bit Rate video/audio can be streamed across the Internet concurrently to many clients. The ability to regulate video based on connection speed can help to provide a balanced mix of transactional versus multimedia content users. Figure 2 compares the bit rate to frame rate and bandwidth.

Figure 3 shows a typical Internet environment with video-streaming support.

The VideoCharger server, a two-tiered client/server design, runs on AIX 4.2. The components consist of the following:

VideoCharger Server: Enhanced HTTP server that delivers streaming video and audio over the Internet/intranet. The server can efficiently manage, store, and deliver content to the connected clients.

VideoCharger Player for Windows 95: An IBM-provided "helper" application player working together with a browser, such as Microsoft Internet Explorer or Netscape Navigator™. The player allows the user to select, receive, present, and control video streams.

Advantages of VideoCharger

The VideoCharger video delivery system has many innovations, including technologies from IBM Research and the Austin development team. There are several key areas of innovation.

Bit Rate	Frame / sec (fps)	Bandwidth	Resolution
Low Bit Rate	7.5	28.8 Kbits/sec	160 x 120
Mid-band	15	256 Kbits/sec	320 x 240
High Bit Rate	30	1.5 Mbits/sec	320 x 240

Figure 2. Comparison of bit rate, frame rate, and bandwidth

Filesystems

Digital video assets are time-based, continuous data streams of information that must be delivered to the viewer at the proper rate in order to preserve human perception of motion and sound. The proper data rate must be maintained from the server to the client; otherwise the viewer will see "glitches" in the video and hear dropouts in the audio.

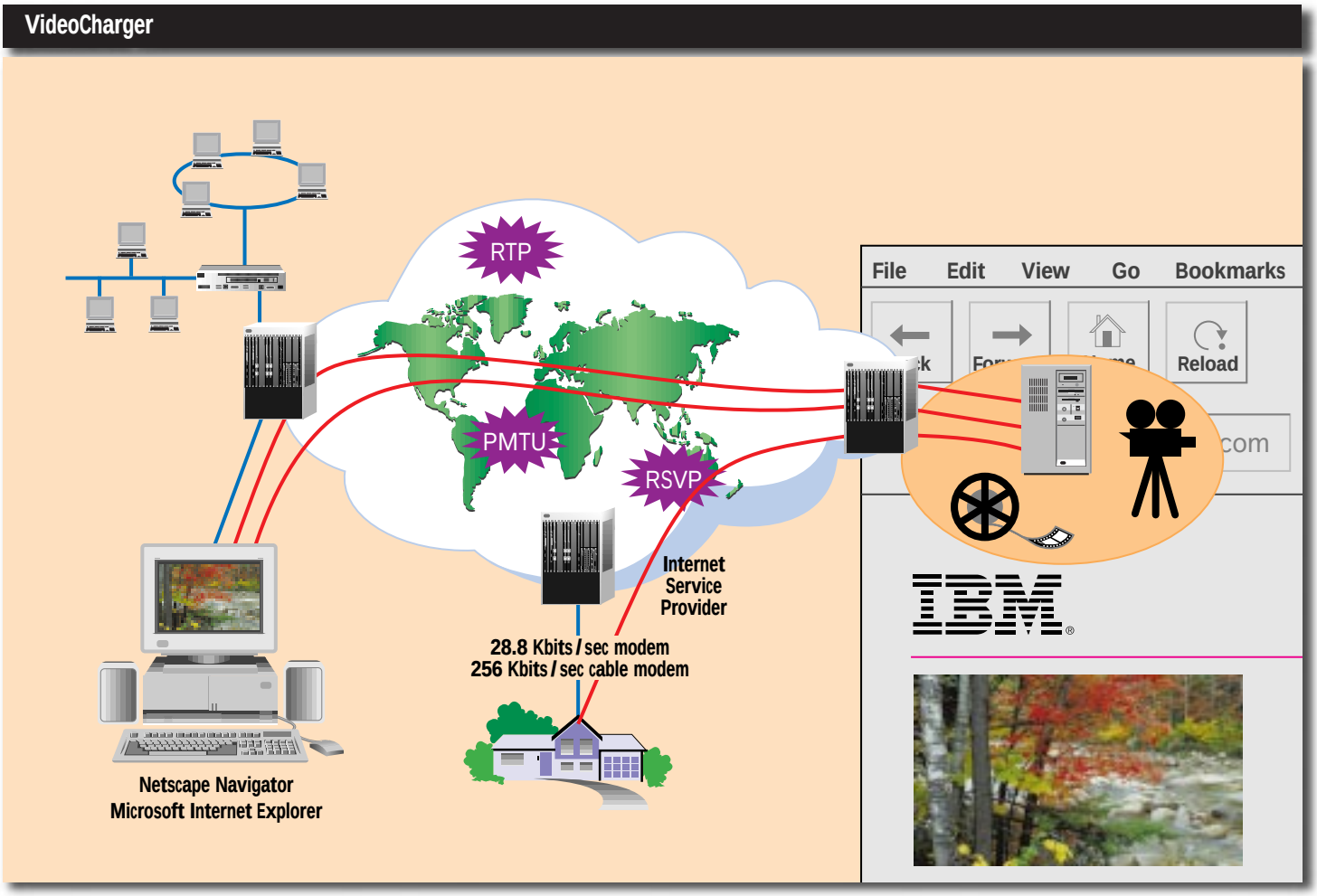


Figure 3. VideoCharger

The VideoCharger Server uses a custom filesystem known as the Multimedia Filesystem. Unlike standard UNIX filesystems designed for storing and retrieving relatively small files, the Multimedia Filesystem was designed for the efficient delivery of digital video assets, providing guaranteed bandwidth for data delivery. The file infrastructure is optimized for the storage, management, and distribution of large digital video files. This design is critical to the balance of the end-to-end delivery system. Some characteristics include the following:

Data Striping. Striping balances the workload evenly across all disks, which increases the aggregate bandwidth of the filesystem. It also permits concurrent access for multiple clients.

Large files and filesystems. Each filesystem can conceptually support up to 64,000 disks (up to 256 terabytes).

Large block reads. Reading large blocks of data (256 KB and up) reduces the number of read requests the server must handle.

Data replication. Replication spreads multiple copies of digital video files across filesystem disks to protect against unexpected disk failure. Replication factors can be assigned to the entire filesystem or to individual files.

Deadline scheduling. This ensures that digital video data delivery takes priority over the OS services.

Error recovery and online reconfiguration. Disks can be added and removed while the system is online with no data loss.

Designed for scalability. A minimal configuration can grow without a major interruption, such as doing a back-up, reformatting a disk, and recopying assets.

Multiple Data Rates

The VideoCharger Server supports clients attached to a variety of network types, such as LANs, WANs, and switching internetworking like ATM. Remote users who access the networks using modems through an Internet Service Provider (ISP) require a minimum of 28.8 Kbits/sec or higher. The server supports bit-rate sensitive content to achieve a balanced quality based on the connected line speed. This encoding is based on the industry-standard H.263 video and 6.3 mode of G723 audio from the videoconferencing industry. This technology provides clients with a 28.8 Kbits/sec modem and acceptable quality of 7.5 frames per second and an 8 kHz audio channel.

Mid-band video quality can be used by Internet/intranet clients attached through ISDN or the emerging cable modem technology. A 256 Kbits/sec connection provides a good quality video resolution of 320 x 240 pixels at about 15 frames per second.

High-quality MPEG-1 video streams recorded at 1.5 Mbits/sec provide a 352 x 240 pixel resolution at 30 frames per second, equivalent to VHS videotape in quality.

The flexible support for multiple data rates allows different clients to attach to the server at various speeds based on its supporting network capabilities. This minimizes the amount of bandwidth used and maximizes the number of concurrent clients that can be attached to the server.

New Internet Protocol Extensions

The VideoCharger Server delivers video content to the PC client using some of the emerging Internet extensions discussed below.

Real-time Transport Protocol (RTP). This new protocol extension was designed for internetworked audio and video conferences that require content synchronization, demultiplexing, media identification, and active-party identification. RTP delivers digital video with a minimum network overhead.

ReSerVation Protocol (RSVP). This protocol reserves network bandwidth in an Internet environment. It allows the request for a specific Quality of Service (QOS) class throughout multiple networks to secure a consistent bandwidth for the delivery of video content. The VideoCharger will take advantage of this extension if the network router supports this function and can better ensure end-to-end quality delivery.

IP Multicasts. This enables the VideoCharger Server to transmit IP datagrams to zero or multiple hosts with minimum streams. The host can, in turn, multicast the video asset to clients within its subnet.

Path MTU Discovery. This support can optimize the Maximum Transmission Unit (MTU) size during the initial network setup. Today's internetworking topology can have at least 15 networks. Each network has different MTU sizes, which can deteriorate the performance during transmission.

VideoCharger Server Component Architecture

The VideoCharger Server is an HTTP server enhanced with the capability to stream digital video data to the IBM-compatible PC clients. The Server can coexist with other Internet servers

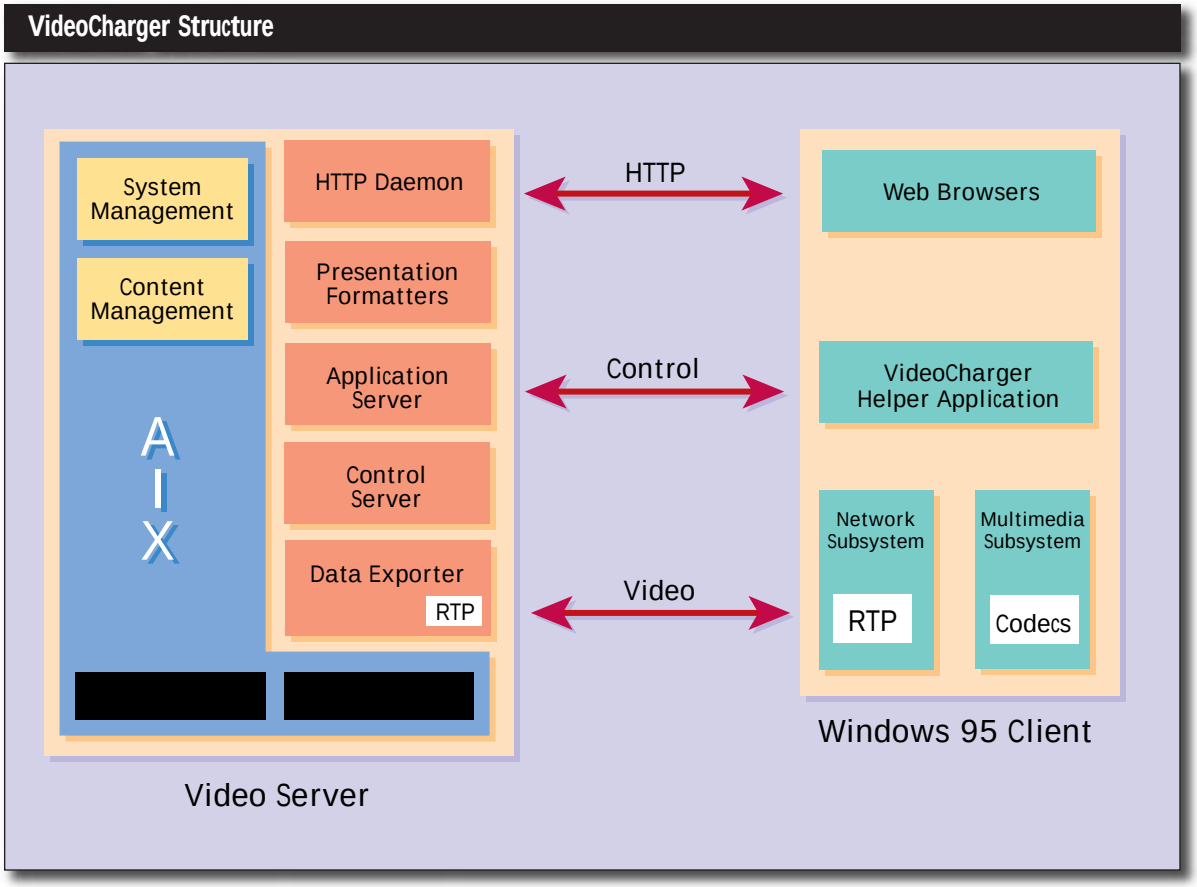


Figure 4. Components of the VideoCharger Server

within the complex using Uniform Resource Locators (URLs) in HTML documents (Web pages). The component infrastructure is scalable from a single system to multiple systems supporting hundreds of concurrent video streams. Figure 4 shows the components of the VideoCharger Server.

Each component of the VideoCharger is described below.

Data Exporter. This interface module controls the rate at which the data is pushed over the Internet. The server can support multiple data exporter modules. Each module runs on a separate server—the key component for scalable data delivery.

Control Server. This supervisory module coordinates the operation of all components and regulates the number of video streams to ensure sufficient resources.

Application Server. The module interface to the client provides video stream control functions, such as video pause, stop, start, and manipulation of the stream.

Multimedia Filesystem. This 64-bit filesystem provides real-time support for playing and recording continuous media.

The VideoCharger allows real-time control of the video stream between the server and the client. It provides support in two ways:

- Real-time feedback from the client to the server at the network level
- User interface commands (VCR buttons) sent from the client to the application server for stop, pause, pace, and position. This support is provided by three HTTP Common Gateway Interface (CGI-bin) programs called Presentation Formatters (PF):
  - — Video Selection formatter for video selection from the list
  - — Video on Demand formatter for query and search of video item
  - — Broadcast Scheduler formatter for video scheduling to be multicast

Figure 5. VideoCharger Player for Windows 95

VideoCharger Client Functions

The plug-in for the VideoCharger Player, provided with the Server product, runs on Windows 95. You can also download it free from http://www.rs6000.ibm.com/solutions/video-servers. The client can access the video content using a Web browser such as Netscape Navigator or Microsoft Internet Explorer. The Player consists of the following:

- Player helper applications
- Networking subsystem, including streaming network interface, application server interface, RTP
- Multimedia subsystem, including ActiveMovie subsystem, streaming source filter, Low Bit Rate (LBR) codec

Figure 5 shows the VideoCharger Player pop-up. Figure 6 shows the interaction between the client and various components of the servers.

Server Scalability

The VideoCharger is designed for scalability; all components of the server can be integrated in a single system for small environments, or can be scaled up using multiple systems with distributed components for a very large environment. Users can grow their configuration by adding more data exporter systems to achieve a higher level

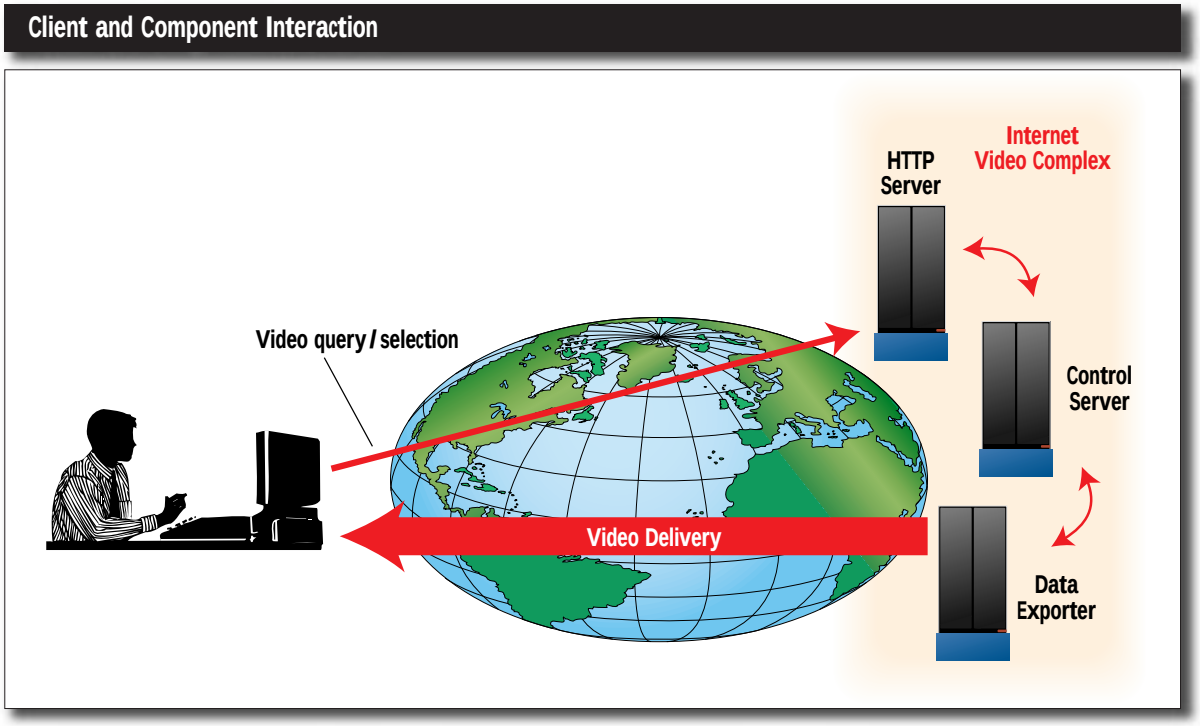


Figure 6. Interaction between client and components

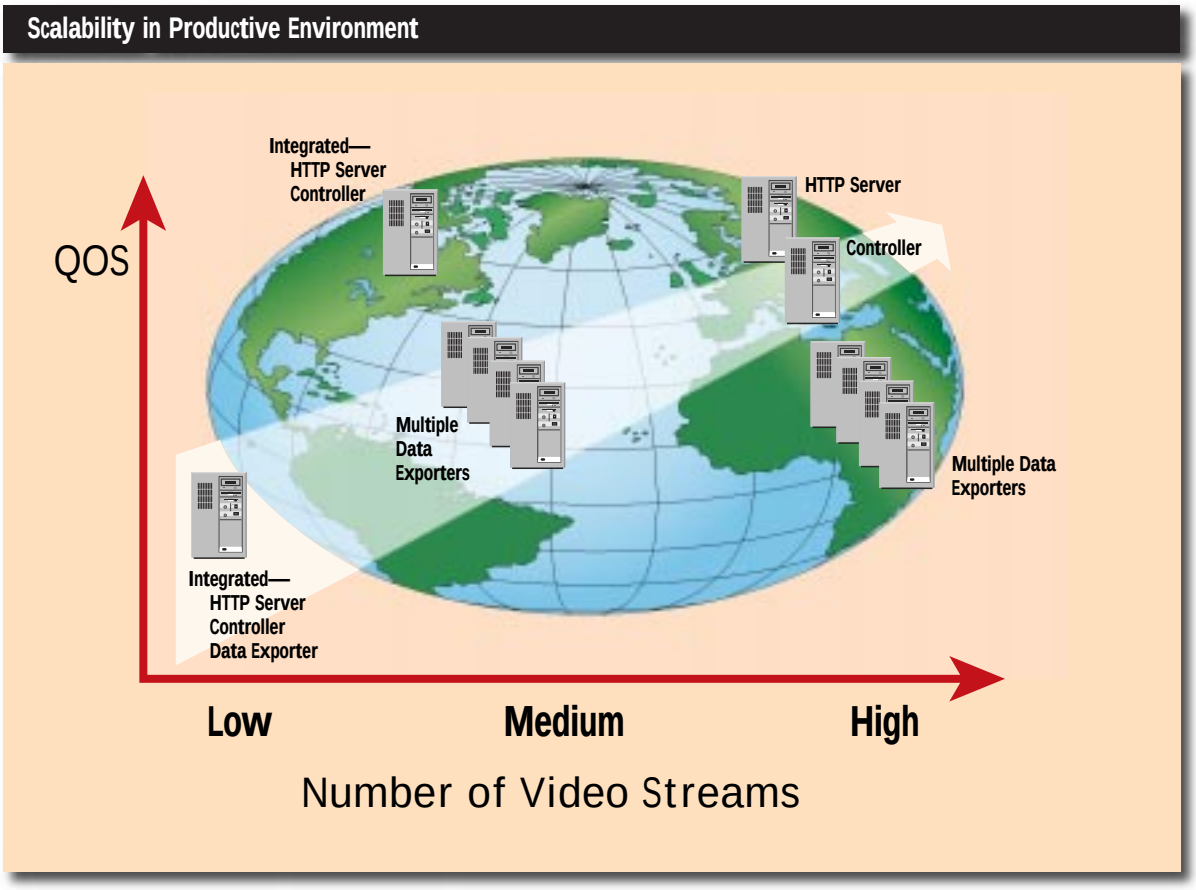


Figure 7. Scalability in productive environments

of performance with a larger number of multimedia streams. Figure 7 highlights the flexibility and scalability in production environments.

Conclusions

Multimedia is changing the way we work. It allows us to be more productive by bringing all data types to the desktop. The spiral effect of faster server systems and higher speed information networks can bring end users from anywhere closer to the real-time environment. The VideoCharger Server technology provides building blocks to launch the second generation wide area Internet-based system with dynamic content delivery from anywhere.

***Eddie Ho**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Ho is a programming consultant in the RS/6000 Executive Briefing Center. He has a BS in Computer Science from the University of Wisconsin and an MS in Computer Science from North Dakota State University.*

***Sam Juliano**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Juliano is a staff programmer in Internet Multimedia Development. He has a BS in Electrical Engineering from the University of Texas in Austin.*

AIX Questions

Compiled by Bruce Pine

The AIX Solution Provider Technical Support Group in Austin, Texas, supports software vendors who are developing or porting applications to AIX. This article is a compilation of questions that are frequently asked by vendors. The name of the responding Technical Support Group staff member appears after each response.

How do I install a nodelock license?

As root, cut and paste the following:

```
 <vendor_id> <product_password>
"<annotation>" "<version>"
```

into `/usr/lib/netls/conf/nodelock`. Here is the result of that procedure:

```
5da54a553b4c.02.09.15.31.05.00.00.00
ftzznpe9hfgqw "2" "3"
```

It may be helpful to place a comment above your license indicating expiration date/compiler; for example:

```
## C Set ++ for AIX version 3. Good from
<date> until <date>
```

Next, verify that your system's date is correct.

—Jeff Simon

How can I recover from a NetLS problem?

Here are the steps for NetLS recovery:

1. `lssrc -g ncs` shows which daemons are running (`llbd`, `glbd`, `nrglbd`)

2. Stop the daemons: `stopsrc -s netlsd`, `stopsrc -s glbd`, and `stopsrc -s llbd`

3. Remove the files shown in Figure 1.

4. Run the following:

```
/etc/ncs/lb_admin
lb_admin: use local
lb_admin: clean (y to delete bad entries)
lb_admin: use global
lb_admin: clean (y to delete bad entries)
lb_admin: quit
```

If no errors occur, then the brokers are in sync. If there are other license servers in the same cell, also execute the following:

```
     /tmp/llbdbase.dat
     /usr/lib/netls/conf/cur_db      **!
/usr/lib/netls/conf/lic_db        !-  Make a backup copy of these files, since they
/usr/lib/netls/conf/lic_db.bak  **!   contain your password. Hence, you will need to
                                      restore them or get new passwords?
/etc/ncs/glb.e     ***!--These are your 2 GLB data bases
/etc/ncs/glb.p     ***!
/etc/ncs/glb_log
/etc/ncs/glb_obj.txt
/etc/ncs/glb_site.txt  ***! This identifies a specific server (usually a faster
                           machine) in your cell that you want your password from.
/usr/lib/netls/conf/log_file
```

Figure 1. Files removed for NetLS recovery

Jeff Simon

```
# drm_admin
drm_admin: set -o glb -h ip:<hostname>
drm_admin: merge_all
drm_admin: quit
```

This causes all of the other Global Location Broker Daemons (`glbd`) in the cell to update their location databases to reflect the changes that were made.

5. Remove `netls_first_time`. Run `/usr/lib/netls/conf/netls_config`, as shown in Figure 2.

Do you want the `llbd` started automatically when the machine boots?	yes
Do you want `netlsd` started automatically when the machine boots?	yes
An initialized database already exists for the `glbd`. Do you wish to use that database when starting the `glbd` daemon?	no
1. Continue with installation without choosing a `Cell_Name`.	
2. Use the default for the system Cell Name.	
3. Create a new alternate cell for the system Cell Name.	
Please indicate your choice (1, 2, 3):	2
Daemons started automatically?	yes

Figure 2. NetLS configuration

6. Run `/usr/lib/netls/conf/netls_first_time`. This should restart the three daemons—`glbd`, `llbd`, and `netlsd` in sync.

—Jeff Simon

How can I call a C program from a FORTRAN program?

Figure 3 shows an example.

—Jeff Simon

How can I change the number of processes assigned per user?

As root, select the options shown in Figure 4 to change the number of processes assigned per user. This can also be done from the command line:

```
chdev -E -l sys0 -a maxuproc=n
```

where `n` is a number between 40 and 10,000.

—Jeff Simon

```
c program name: FortranCallingC.f
      write (6,*) "FORTRAN is not just for mathmeticians"
      call hello
      end


/* program name: Cprog.c */
void hello()
{
  printf("Most of UNIX is written in  C\n");
}

syntax of compilation:  xlc -c Cprog.c
                        xlf FortranCallingC.f Cprog.o -o <filename>
```

Figure 3. Calling a C program from a FORTRAN program

```
smitty
  System Environments
    Change / Show Characteristics of Operating System
      Maximum number of PROCESSES allowed per user
```

Figure 4. Changing the number of processes per user

Is it possible to distinguish the setting for maxdata with an executable?

Maxdata is a binder option that is set at link time; that is, `-bmaxdata:0xy0000000` where y > 2. This information can be obtained from an executable by entering the following:

```
dump -ov <executable>
```

A section called Optional Header will contain this information:

```
maxSTACK      maxDATA
0x00000000  0x50000000
```

—Jeff Simon

How do I print the scope of a class from within DBX?

Use the `which` command from within DBX; for example, `which <class_name>`

—Jeff Simon

How do I print the type of a class from within DBX?

Use the `whereis` command from within DBX; for example, `whereis <function>`

—Jeff Simon

How do I print a variable assignment inside a class?

Here are the steps to print a variable assignment of a class:

1. Compile program using `-g`, which inserts debug information into the symbol table

2. Step through code until the variable needed has been "seen"

3. Print `<variable_name>`

Note: Many DBX subcommands contain a "fastpath" by substituting the DBX subcommand with the first letter of the command; that is, `p <variable_name>`.

—Jeff Simon

How can I do a fast IPL on an SMP system?

David Stewart

Either enter the following command in `inittab` or run it from the command line:

```
mpcfg -cf 11 1    (command line)
```

A fast IPL can also be set in the Standby menu. Follow the steps described below (see the Service Guide for the R30 for more information):

1. Place the key mode switch to service mode; press the enter key on a terminal connected to the S1 or S2 line.

2. The BUMP clears the screen on the selected line, issues a prompt, and waits for a keyword.

3. Enter the keyword `sbb` to display the Standby menu. The Standby menu consists of a main menu with several options as shown below:

   0 Display Configuration
   1 Set Flags
   2 Set Unit Number
   3 Set Configuration
   4 SSbus Maintenance
   5 I2C Maintenance

Choose option 1. This will give you some flags to toggle ON or toggle OFF. Figure 5 shows option 1.

—David Stewart

Remote Authorization flag	Disabled or Enabled
BUMP Console flag	Disabled or Enabled
Autoservice IPL flag	Disabled or Enabled
Extended Tests parameter	Disabled or Enabled
Power-On Tests in Trace Mode flag	Disabled or Enabled
Power-On Tests in Loop Mode flag	Disabled or Enabled
Fast IPL flag	Disabled or Enabled
Set Electronic Mode Switch to Normal	Disabled or Enabled
Select x to exit once the flags are set.	

Figure 5. Standby menu—Option 1

How can I identify what shared libraries an executable file uses?

Use the `dump -H` command on the executable file. Figure 6 shows an example.

The Import File Strings section shows the library path searched for the shared libraries and the shared libraries that are actually loaded (in this case, `libXt.a`, `libXext.a`, `libX11.a`, and `libc.a`).

—Bill Woodward

How can I determine what shared libraries and dynamically loadable code are being used by a running process?

Since some executables (such as the X server) dynamically load executable code, the `dump -H` command may not show all of the code dependencies. In this case, the `genld` command can be used. The `genld` command is part of the `perfagent.tools` Licensed Program Product (LPP).

To run `genld`, just type `genld` and redirect the output to a file. Because this prints information for all of the system processes, you will need to edit the file and search for the process name or process id.

For example, `genld` shows that X server is using the files shown in Figure 7.

Figure 7 shows that the X server has dynamically loaded the XVideo extension (`/usr/lpp/UMS/lib/loadxv/`), and has also dynamically loaded the device-specific code for the GXT500 (`/usr/lpp/gai/60x00004002/loadddx/`) as well as the other libraries shown by `genld`.

—Bill Woodward

```
# dump -H /usr/bin/X11/bitmap

/usr/bin/X11/bitmap:

                        ***Loader Section***
                     Loader Header Information
VERSION#         #SYMtableENT     #RELOCent        LENidSTR
0x00000001       0x000000e4       0x00000b52       0x00000061

#IMPfilID        OFFidSTR         LENstrTBL        OFFstrTBL
0x00000005       0x00009d58       0x00000c65       0x00009db9

                        ***Import File Strings***
INDEX  PATH                          BASE                MEMBER
0      /usr/lib:/lib:/usr/lpp/xlC/lib
1                                    libXt.a             shr4.o
2                                    libc.a              shr.o
3                                    libX11.a            shr4.o
4                                    libXext.a           shr.o
```

Figure 6. dump -H example

```
Proc_pid:   6036   Proc_name: X
                    d04f3000  /usr/lpp/UMS/lib/loadxv/
                    d04ef000  /usr/lpp/gai/60x00004002/loadrms/
                    d04ec0a8  /usr/lib/libgair4.a/shr.o
                    d0471000  /usr/lpp/gai/60x00004002/loadddx/
                    d012e0a8  /usr/lib/libodm.a/shr.o
                    d013e0a8  /usr/lib/libcfg.a/shr.o
                    d01c40a8  /usr/lib/libdbm.a/shr.o
                    d01266f0  /usr/lib/libc.a/meth.o
                    d0000380  /usr/lib/libc.a/shr.o
                    10000000  X
```

Figure 7. Files used by X server

How can I identify what workstation type is actually being used for a graPHIGS™ application with the workstation type specified as "X", meaning to use the best available workstation type?

Since the workstation-dependent code is dynamically loaded depending on the actual workstation type selected, you must use `genld` to generate the list of shared libraries and dynamically loaded code. Then use that information to determine the actual workstation type selected.

In a typical graPHIGS application, `genld` will show a large number of shared libraries and dynamically loaded code. However, only a few are necessary to determine the actual workstation type.

Example 1: The soft graPHIGS pipeline (`/usr/lib/libgppipe.a`) is in use, and the graPHIGS rasterizer is the soft rasterizer (`/usr/lib/librengp_soft.a`). The application is using the XSOFT workstation type. See Figure 8.

Example 2: The soft graPHIGS pipeline (`/usr/lib/libgppipe.a`) is in use, but we are using a device-specific rasterizer (`/usr/lib/librengppex_ppr.a`). The application is using the XDWA workstation type with the soft pipeline introduced in AIX 4.1.4. See Figure 9.

Example 3: Here we are using a device-specific `load3dm1` module. The `load3dm1` module is the interface to the graPHIGS code running on the graphics adapter. The application is using the XDWA workstation type that was used prior to AIX 4.1.4. See Figure 10.

```
Proc_pid:   21388   Proc_name: gptest
                        [deleted irrelevant libs]
                        d17ad0c0  /usr/lib/libgppipe.a/shr.o
                        d1e3df18  /usr/lib/librengp_soft.a/shr_r.o
                        d14300c0  /usr/lib/librengp_soft.a/shr.o
                        d0241000  /usr/lpp/graPHIGS/bin/loads3d/
                        [deleted irrelevant libs]
                        10000000  gptest
```

Figure 8. XSOFT workstation type

```
Proc_pid:   21434   Proc_name: gptest
                        [deleted irrelevant libs]
                        d25270c0  /usr/lib/librengppex_ppr.a/shr.o
                        d2526000  /usr/lpp/gai/adapter4.r4/GPrasterizer/
                        d17ad0c0  /usr/lib/libgppipe.a/shr.o
                        d2525000  /usr/lpp/gai/adapter4.r4/GPpipeline/
                        d2522000  /usr/lpp/gai/adapter4.r4/load3dm3/
                        d01ac000  /usr/lpp/gai/adapter4.r4/loadrms/
                        [deleted irrelevant libs]
                        10000000  gptest
```

Figure 9. XDWA workstation type

```
Proc_pid:   21442   Proc_name: gptest
                         [deleted irrelevant libs]
                         d25a9000  /usr/lpp/gai/adapter4.r4/load3dm1/
                         d01ac000  /usr/lpp/gai/adapter4.r4/loadrms/
                         [deleted irrelevant libs]
                         10000000  gptest
```

Figure 10. XDWA workstation type prior to AIX 4.1.4

```
Proc_pid:   21418   Proc_name: gptest
                        [deleted irrelevant libs]
                        d14250c0  /usr/lib/libXi.a/shr.o
                        d013a0c0  /usr/lib/libodm.a/shr.o
                        d01790c0  /usr/lib/libgair4.a/shr.o
                        d0e690c0  /usr/lib/libXext.a/shr.o
                        d0191c30  /usr/lib/libX11.a/shr4net.o
                        d018c0c0  /usr/lib/libIM.a/shr.o
                        d017c0c0  /usr/lib/libiconv.a/shr4.o
                        d02a90c0  /usr/lib/libX11.a/shr4.o
                        [deleted irrelevant libs]
                        10000000  gptest
```

Figure 11. XLIB workstation type

Example 4: None of the other conditions apply. The application is using the XLIB workstation type. See Figure 11.

In cases where graphics adapter-specific code is loaded, the entries may be slightly different. However, for post-AIX 4.1.4 graPHIGS, the device-specific rasterizer will be named `librengppex_xxx.a`, where the `xxx` is a three-letter code specific to the device. For the pre-AIX 4.1.4 code, the `load3dml` will be consistent, although it may be loaded from a different directory.

—Bill Woodward

Bruce Pine, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758.*

POWER NOTES

Support Line Update

"An Easy Reference for Getting the Most Out of Support Line," now available on the World Wide Web, updates and replaces the information published in the November 1996 article "Talking to AIX Support Line."

Go to http://service.software.ibm.com, IBM Software Technical Support, and follow these links to the Support Line information:

1. AIX Software Technical Support

2. Choose a country: United States

3. About AIX Support Line (phone support)—under Roadmap to AIX Technical Support (English only)

Put your bookmark at this location to easily refer to AIX Support Line telephone support information.

AIX Cluster Solutions Tested

A development group at IBM Austin has taken existing software products running concurrently on a collection of RS/6000 hardware to produce a clustered environment. The types of products tested included administration, network management, load balancing, storage, printing, performance, and middleware. Dozens of test cases were performed to discover how these products achieved high availability, manageability, and scalability.

Information about these test cases and their results is available on the World Wide Web. To view this information, open the following URL: http://www.rs6000.ibm.com/resource/technology. From here, click on the "More" link or scroll down to the last section, More Tech ReSources. Select the link entitled "Testing Cluster Solutions in an RS/6000 Environment."

SUPER! '97

A User Conference for RS/6000 SP Customers

Towards Teraflops, Petabytes, and Terabits/sec

Where: Austin, Texas

When: April 27-30, 1997

Registration Fee: $300 prior to April 1, 1997 and $375 on or after April 1

Who Should Attend: Users, developers, researchers, computing center managers, and technical staff who work with high-performance parallel and clustered computing; anyone anticipating the arrival of an SP

The SUPER! '97 conference theme, "Moving Toward Teraflops, Petabytes and Terabits/Sec," highlights the established role of the RS/6000 SP in solving research and business problems with its high-performance computational, I/O, and networking power. Sessions will emphasize sharing experiences and techniques for configuring, operating, managing, and using the RS/6000 SP effectively. The conference will also feature emerging applications that take advantage of the RS/6000 SP's scalable architecture.

IBM will openly discuss future plans and requirements for the RS/6000 SP and respond to all questions. Conference activities include the following:

- Talks by distinguished speakers in the field
- Technical sessions about RS/6000 SP system software, system management and administration, application support tools, and configuration options including discussions about RS/6000 hardware and software that complement the RS/6000 SP
- Panel discussion among RS/6000 SP sites on system software installation, operation, and migration
- Open forum with IBM product development with "open mike" Q&A for attendees
- Tutorials including RS/6000 SP network tuning, high-performance FORTRAN, and RS/6000 SP filesystems
- RS/6000 SP User Group discussion on current and new requirements
- The Site Gallery that showcases how the RS/6000 SP is being used by hundreds of customers
- Informal discussion opportunities, including a special dinner excursion

How to Register

For additional conference details, including complete agenda, registration forms, and optional secure electronic registration via secure Web browser, see the SUPER! Home Page.

- **Online:** http://ike.engr.washington.edu/super/
- **E-mail:** super@ike.engr.washington.edu
- **Phone:** 800-882-8784 or 512-471-3121

For five

days in May the

Gateway to the West

will be the gateway to

cross-platform developer

heaven.

If your future is network computing, IBM's 6th annual Technical Interchange is the event of the year you can't afford to miss. Exhibits, hands-on and elective sessions, and free certification on leading IBM technologies. You'll benefit from the latest and hottest topics such as Lotus® Domino,™ Windows NT* and preparation for the Year 2000.

Join us in fun-filled St. Louis from May 11 to 15 for more developer hints, tips and tricks than a barrel of monkeys. We'll even throw in a special-edition CD of IBM's Developer Connection for Java™ and Internet tools, plus a ton of other goodies.

Check out the conference details. Visit our Website at www.software.ibm.com/ibmti or call 1 800 872-7109, ext. 4000 (outside the U.S. call 508 440-9700), for immediate enrollment.

Do it now, and experience heaven on earth.

IBM®

Solutions for a small planet™

Java World Tour '97

IBM, Netscape, Sun Microsystems, and Novell have joined forces to offer a one-day event that could change the way you write software. Forever.

The Java World Tour '97 begins in February with seminars in more than 40 cities around the world. This one-day live satellite conference is designed for commercial developers and other business and technology professionals within the software development community. It will feature five education segments with demonstrations of applications from leading Java developers.

Keynote speakers include:

- John M. Thompson, Senior Vice President, IBM Software
- Jim Barksdale, CEO Netscape Communications
- Scott McNealy, CEO, Sun Microsystems

Following the keynote speakers are sessions covering various aspects of major topics such as the networked view of the application environment, system management for the network computing environment, how to build Java applications, and electronic business. A Q&A session and demonstrations are also part of the event.

Each attendee will receive instructional and Java tools on CD-ROMs and a CBT Systems CD-ROM featuring four hours of Java education.

Registration

The early-bird registration rate is $99 (U.S.), which is valid up to five working days prior to the event. Registration at the door is $149 (U.S.).

If you are a member of the IBM Solution Developer Program, your registration fee for the Java World Tour '97 is only $49. To become a member of the Program, call 800-627-8363 or send E-mail to ibmsdp@vnet.ibm.com. You can also enroll by completing the registration form on the home page at http://www.developer.ibm.com/

World Tour '97 Locations

March 10, 1997	**Atlanta** **Baltimore** **Boston / Cambridge** **Montreal** **New York** **Orlando** **Stamford** **Toronto** **Washington / Alexandria**
April 22, 1997	**London** **Paris** **Munich** **Stockholm** **Vienna** **Milan** **Amsterdam** **Barcelona** **Zurich** **Brussels**

These rates are for North American locations only. For additional information and pricing for locations outside North America, visit our Web site at http:/www.ibm.com/javaworldtour.

Registration options include:

- **Online:** Click Registration on the Web site: http:/www.ibm.com/javaworldtour
- **Phone:** 888-554-JAVA (U.S. only) or 415-372-7073
- **Fax:** 415-525-0199
- **U.S. Mail:** The Java Education World Tour
c/o SOFTBANK Expos
P.O. Box 45295
San Francisco, CA 94145-0295

Simply AIX

By Casey Cannon, Carolyn Jones, and Scott Trent

Published November, 1996 by Prentice Hall Professional Technical Reference
Copyright 1997, 352 pp.
Paper Bound w/CD-ROM
ISBN 0-13-568882-5
$39.93

Simply AIX provides a comprehensive, friendly introduction to AIX for system administrators and new users. It presents AIX-specific information in an easy to use, accessible format. It covers AIX 4.2, the most recent release of AIX. The book also includes a CD-ROM with many useful tools and utilities.

Topics covered include the following:

- Getting AIX up and running
- Customizing your AIX system
- Connecting peripherals and establishing a connection to the Internet
- Comparing AIX with DOS and Windows environments
- Administering an AIX system

The CD-ROM includes AIX development toolkits, software evaluation releases, product demos, internal development and productivity tools from IBM, product documentation, and sample device driver code.

This book is for anyone who uses AIX or administers an AIX system, including new users and experts.

Table of Contents

Can Multitask?). Visual System Manager. Backup—and Make Restore an Option!

9. AIX Speaks Your Language: Internationalization
Why Should I Care About Speaking Your Language? I'm Convinced! How do I Get AIX to Speak My Language? This is Interesting! Tell Me More! References.

10. All the Help You Need
Unlimited Online Information. 1-800-IBM-4FAX, Lots of Information for FREE. Power Team Developer Support (1-800-627- 8363). AIX and RS/6000: Resources on the Net. AIX News Groups. RS/6000 Talk Radio.

11. Gathering Up the Pieces
Performance Tools for System Management. The AIX Development Environment. Security and System Management. FixDist—How to Get the Latest Bug Fixes from the Internet. AIX PitStop—How to Resolve Common AIX Problems.

12. IBM Developer Connection for AIX CD-ROM
What is the IBM Developer Connection for the AIX Program? CD-ROM Installation. Contents of IBM Developer Connection for AIX CD-ROM. Appendixes. AIX Man Pages.

Appendix A. Directory Commands
Appendix B. File Manipulation Commands
Appendix C. Backup Commands
Appendix D. Miscellaneous Commands
Appendix E. Connectivity Commands