IBM'S MAGAZINE FOR AIX DEVELOPERS JUNE 1998

AIXpert

FOCUS ON CLIENT/SERVER PROGRAMS

Client/Server Programming on AIX

Inside this Issue...

- Applets and Servlets
- Selecting Nodes for RS/6000 SP
- Porting Smart Cards to AIX

AIXpert BACK TO CONTENTS

IBM

AIXpert DIGITAL VERSION IBM

TABLE OF CONTENTS

Click here to view a full-size version of this issue's cover.

Click here to go to a complete version of this issue for printing.

COMMENTARY

This Ain't Your Grandpa's Client/Server
By George Noren

AIX

Managing Software Licenses
By Jeff Simon

Porting Smart Cards to AIX
By George Kraft IV

CLIENT/SERVER

Applets and Servlets: A Smooth Blend
By Greg Flurry and Jeff Jilg

Chat Applet Program Listings
*See CHAT directory on this CD (AIXPERT/98JUNE/CHAT for files)

Printing in a Network Station Environment
By Eddie Ho, Ravi Mandavi, Steve Heracleous, and Claude Bechard

Multithreaded Programming with Java RMI
By Chary Tamirisa

Selecting Nodes for the IBM RS/6000 SP
By Clive Harris

Shared Memory Programming on the RS/6000 SP
By David Klepacki and Xianneng Shen

Q&A

AIX Questions
Compiled by Jeff Simon

CALL TO ACTION

Explore the Power of the RS/6000 SP Server

JUNE 1998

AIXpert BACK TO MAIN MENU

Explore the Power of the RS/6000 SP Server

Solution Partnership Centers are a benefit of the IBM Solution Developer Program. The IBM Solution Developer Program provides marketing support to solution developers, helping them reach new customers and expand into new markets worldwide.

IBM

IBM Solution Partnership Center
2929 Campus Drive Suite 275
San Mateo, CA 94403
Phone: 650-524-5246
Fax: 650-524-5310
E-mail address: spcw@spc.ibm.com
Web: http://www.spc.ibm.com.

IBM Solution Partnership Center
404 Wyman Street – North Entrance
Waltham, MA 02154
Phone: 1-800-678-4249 or
781-895-2610

If you are a commercial software developer and you want to know how your application runs on the new RS/6000 SP large scale server, visit the IBM Solution Partnership Center (SPC). At the SPC, you can explore the power and performance of the SP server for transaction processing and discover how the new version of AIX provides a solid foundation for a variety of application—including applications for the world of e-business and business intelligence.

The SPCs offer private labs to commercial software developers for porting, enabling, and testing their applications. These labs are equipped with a wide range of tools and technologies from industry-leading hardware and software providers. While visiting a Solution Partnership Center, developers can work with their applications on various platforms without investing in any additional equipment or staff. The SPCs also offer ongoing technical seminars and hands-on workshops related to critical technology issues.

For additional information about the Solution Partnership Centers, visit the Web site: http://www.spc.ibm.com/

Information about the IBM Solution Developer Program is available at the Web site: http://www.developer.ibm.com/

```
package test;

import java.applet.*;
import java.awt.*;
import java.util.*;
import java.io.*;
import java.net.*;

/**
 * This applet was generated by a SmartGuide.
 *
 */
public class ChatApplet extends Applet implements
java.awt.event.ActionListener {
      ChatListener chatListen = null;
      Socket client = null;
      BufferedReader in = null;
      InputStreamReader inISR = null;
      private Button ivjButton1 = null;
      private Button ivjButton2 = null;
      private Button ivjButton3 = null;
      private Button ivjButton4 = null;
      private TextArea ivjChatRecord = null;
      private Dialog ivjDialog1 = null;
      private Label ivjLabel1 = null;
      private Label ivjLabel2 = null;
      private Label ivjLabel3 = null;
      private Panel ivjPanel1 = null;
      private Panel ivjPanel2 = null;
      private Button ivjSendButton = null;
      private TextField ivjTextField1 = null;
      private TextField ivjToServer = null;
      PrintWriter out = null;
      String host = "flurry";
      int port = 80;

      String version = "1.07";


/**
 * Method to handle events for the ActionListener interface.
 * @param e java.awt.event.ActionEvent
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
public void actionPerformed(java.awt.event.ActionEvent e) {
      // user code begin {1}
      // user code end
      if ((e.getSource() == getSendButton()) ) {
            conn0(e);
      }
      if ((e.getSource() == getButton1()) ) {
            conn1(e);
      }
      if ((e.getSource() == getButton3()) ) {
            conn2(e);
      }
      if ((e.getSource() == getButton2()) ) {
            conn3(e);
      }
      if ((e.getSource() == getButton4()) ) {
```

```
                conn4(e);
        }
        // user code begin {2}
        // user code end
}

/**
 * conn0:  (SendButton.action.actionPerformed(java.awt.event.ActionEvent) --
> ChatApplet.sendText())
 * @param arg1 java.awt.event.ActionEvent
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void conn0(java.awt.event.ActionEvent arg1) {
        try {
                // user code begin {1}
                // user code end
                this.sendText();
                // user code begin {2}
                // user code end
        } catch (java.lang.Throwable ivjExc) {
                // user code begin {3}
                // user code end
                handleException(ivjExc);
        }
}

/**
 * conn1:  (Button1.action.actionPerformed(java.awt.event.ActionEvent) -->
Dialog1.show())
 * @param arg1 java.awt.event.ActionEvent
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void conn1(java.awt.event.ActionEvent arg1) {
        try {
                // user code begin {1}
                // user code end
                getDialog1().show();
                // user code begin {2}
                // user code end
        } catch (java.lang.Throwable ivjExc) {
                // user code begin {3}
                // user code end
                handleException(ivjExc);
        }
}

/**
 * conn2:  (Button3.action.actionPerformed(java.awt.event.ActionEvent) -->
Dialog1.dispose())
 * @param arg1 java.awt.event.ActionEvent
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void conn2(java.awt.event.ActionEvent arg1) {
        try {
                // user code begin {1}
                // user code end
                getDialog1().dispose();
                // user code begin {2}
                // user code end
        } catch (java.lang.Throwable ivjExc) {
                // user code begin {3}
                // user code end
```

```
                handleException(ivjExc);
        }
}
/**
 * conn3:  (Button2.action.actionPerformed(java.awt.event.ActionEvent) -->
ChatApplet.connect())
 * @param arg1 java.awt.event.ActionEvent
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void conn3(java.awt.event.ActionEvent arg1) {
        try {
                // user code begin {1}
                // user code end
                this.connect();
                // user code begin {2}
                // user code end
        } catch (java.lang.Throwable ivjExc) {
                // user code begin {3}
                // user code end
                handleException(ivjExc);
        }
}

/**
 * conn4:  (Button4.action.actionPerformed(java.awt.event.ActionEvent) -->
ChatApplet.disconnect())
 * @param arg1 java.awt.event.ActionEvent
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void conn4(java.awt.event.ActionEvent arg1) {
        try {
                // user code begin {1}
                // user code end
                this.disconnect();
                // user code begin {2}
                // user code end
        } catch (java.lang.Throwable ivjExc) {
                // user code begin {3}
                // user code end
                handleException(ivjExc);
        }
}

/**
 * This method was created by a SmartGuide.
 */
public void connect( ) {

        String newPort = null;
        URL chatURL = null;

        // get alias
        String alias = ivjTextField1.getText();
        ivjDialog1.dispose();

        System.out.println("Entering connect()");
        try {
                // point to server
                chatURL = new URL("http", host, port, "/servlet/ChatServlet");
        } catch (MalformedURLException e) {
                System.out.println("URL problem: " + e);
        }
```

```
      System.out.println("Formed URL");
      try {
            // contact server
            InputStream serverIn = (InputStream) chatURL.getContent();
            BufferedReader bSI = new BufferedReader(new
InputStreamReader(serverIn));
            System.out.println("Contacted server");
            newPort = bSI.readLine();
            int thePort = new Integer(newPort).intValue();
            System.out.println("Contacted servlet and got a response of " +
newPort);

            if (thePort == 0) {
                  // can't connect to server
                  ivjChatRecord.append("Sorry, cannot connect to server;
try later.\n");
                  return;
            }

            // attach to new server port
            client = new Socket(host, thePort);

            // get i/o streams
            inISR = new InputStreamReader(client.getInputStream());
            in = new BufferedReader(inISR);
            out = new PrintWriter(client.getOutputStream(),true);

      } catch (IOException e) {
            System.out.println("IOException:  " + e);
      }

      // send the name string
      out.println(alias);

      // establish a listener for the server
      chatListen = new ChatListener(in, ivjChatRecord);
      chatListen.start();

      return;
}
 /**
 * This method was created by a SmartGuide.
 */
public void disconnect( ) {

      // stop the server
      out.println("QUIT");

      // now wait for the listener to quit
      chatListen.stopListening();

      // close all the things
      try {
            in.close();
            out.close();
            client.close();
      } catch (IOException e) {
            System.out.println("IOException:  " + e);
      }

      return;
```

```
}
/**
 * Gets the applet information.
 * @return java.lang.String
 */
public String getAppletInfo() {
      return "test.ChatApplet created using VisualAge for Java.";
}
/**
 * Return the Button1 property value.
 * @return java.awt.Button
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private Button getButton1() {
      if (ivjButton1 == null) {
            try {
                  ivjButton1 = new java.awt.Button();
                  ivjButton1.setName("Button1");
                  ivjButton1.setLabel("Connect");
                  // user code begin {1}
                  // user code end
            } catch (java.lang.Throwable ivjExc) {
                  // user code begin {2}
                  // user code end
                  handleException(ivjExc);
            }
      };
      return ivjButton1;
}

/**
 * Return the Button2 property value.
 * @return java.awt.Button
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private Button getButton2() {
      if (ivjButton2 == null) {
            try {
                  ivjButton2 = new java.awt.Button();
                  ivjButton2.setName("Button2");
                  ivjButton2.setLabel("OK");
                  // user code begin {1}
                  // user code end
            } catch (java.lang.Throwable ivjExc) {
                  // user code begin {2}
                  // user code end
                  handleException(ivjExc);
            }
      };
      return ivjButton2;
}
/**
 * Return the Button3 property value.
 * @return java.awt.Button
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private Button getButton3() {
      if (ivjButton3 == null) {
            try {
                  ivjButton3 = new java.awt.Button();
                  ivjButton3.setName("Button3");
                  ivjButton3.setLabel("Cancel");
```

```
                        // user code begin {1}
                        // user code end
                } catch (java.lang.Throwable ivjExc) {
                        // user code begin {2}
                        // user code end
                        handleException(ivjExc);
                }
        };
        return ivjButton3;
}

/**
 * Return the Button4 property value.
 * @return java.awt.Button
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private Button getButton4() {
        if (ivjButton4 == null) {
                try {
                        ivjButton4 = new java.awt.Button();
                        ivjButton4.setName("Button4");
                        ivjButton4.setLabel("Disconnect");
                        // user code begin {1}
                        // user code end
                } catch (java.lang.Throwable ivjExc) {
                        // user code begin {2}
                        // user code end
                        handleException(ivjExc);
                }
        };
        return ivjButton4;
}
/**
 * Return the ChatRecord property value.
 * @return java.awt.TextArea
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private TextArea getChatRecord() {
        if (ivjChatRecord == null) {
                try {
                        ivjChatRecord = new java.awt.TextArea();
                        ivjChatRecord.setName("ChatRecord");
                        // user code begin {1}
                        // user code end
                } catch (java.lang.Throwable ivjExc) {
                        // user code begin {2}
                        // user code end
                        handleException(ivjExc);
                }
        };
        return ivjChatRecord;
}

/**
 * Return the Dialog1 property value.
 * @return java.awt.Dialog
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private Dialog getDialog1() {
        java.awt.GridBagConstraints constraintsLabel3 = new
java.awt.GridBagConstraints();
        java.awt.GridBagConstraints constraintsTextField1 = new
```

```
java.awt.GridBagConstraints();
      java.awt.GridBagConstraints constraintsButton2 = new java.awt.GridBagConstraints();
      java.awt.GridBagConstraints constraintsButton3 = new java.awt.GridBagConstraints();
      if (ivjDialog1 == null) {
            try {
                  ivjDialog1 = new java.awt.Dialog(new java.awt.Frame());
                  ivjDialog1.setName("Dialog1");
                  ivjDialog1.setLayout(new java.awt.GridBagLayout());
                  ivjDialog1.setBounds(23, 284, 405, 99);
                  ivjDialog1.setTitle("Chat Connection");

                  constraintsLabel3.gridx = 0; constraintsLabel3.gridy = 0;
                  constraintsLabel3.gridwidth = 1; constraintsLabel3.gridheight = 1;
                  constraintsLabel3.anchor = java.awt.GridBagConstraints.WEST;
                  constraintsLabel3.weightx = 0.0;
                  constraintsLabel3.weighty = 0.0;
                  constraintsLabel3.insets = new java.awt.Insets(10, 2, 0, 0);
                  ((java.awt.GridBagLayout)
getDialog1().getLayout()).setConstraints(getLabel3(), constraintsLabel3);
                  getDialog1().add(getLabel3());

                  constraintsTextField1.gridx = 1;constraintsTextField1.gridy = 0;
                  constraintsTextField1.gridwidth = 3;constraintsTextField1.gridheight = 1;
                  constraintsTextField1.fill = java.awt.GridBagConstraints.HORIZONTAL;
                  constraintsTextField1.anchor = java.awt.GridBagConstraints.EAST;
                  constraintsTextField1.weightx = 1.0;
                  constraintsTextField1.weighty = 0.0;
                  constraintsTextField1.insets=new java.awt.Insets(10, 0, 0, 2);
                  ((java.awt.GridBagLayout)
getDialog1().getLayout()).setConstraints(getTextField1(), constraintsTextField1);
                  getDialog1().add(getTextField1());

                  constraintsButton2.gridx = 0; constraintsButton2.gridy = 1;
                  constraintsButton2.gridwidth = 1; constraintsButton2.gridheight = 1;
                  constraintsButton2.fill = java.awt.GridBagConstraints.HORIZONTAL;
                  constraintsButton2.anchor = java.awt.GridBagConstraints.WEST;
                  constraintsButton2.weightx = 0.0;
                  constraintsButton2.weighty = 1.0;
                  constraintsButton2.insets = new java.awt.Insets(0, 2, 0, 10);
                  ((java.awt.GridBagLayout)
getDialog1().getLayout()).setConstraints(getButton2(), constraintsButton2);
                  getDialog1().add(getButton2());

                  constraintsButton3.gridx = 1; constraintsButton3.gridy = 1;
                  constraintsButton3.gridwidth = 3; constraintsButton3.gridheight = 1;
                  constraintsButton3.fill = java.awt.GridBagConstraints.HORIZONTAL;
                  constraintsButton3.anchor = java.awt.GridBagConstraints.CENTER;
                  constraintsButton3.weightx = 1.0;
                  constraintsButton3.weighty = 1.0;
                  constraintsButton3.insets = new java.awt.Insets(0, 10, 0, 0);
                  ((java.awt.GridBagLayout)
getDialog1().getLayout()).setConstraints(getButton3(), constraintsButton3);
                  getDialog1().add(getButton3());
                  // user code begin {1}
                  // user code end
            } catch (java.lang.Throwable ivjExc) {
                  // user code begin {2}
                  // user code end
                  handleException(ivjExc);
            }
      };
      return ivjDialog1;
}
```

```
/**
 * Return the Label1 property value.
 * @return java.awt.Label
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private Label getLabel1() {
      if (ivjLabel1 == null) {
            try {
                  ivjLabel1 = new java.awt.Label();
                  ivjLabel1.setName("Label1");
                  ivjLabel1.setText("Send This:");
                  // user code begin {1}
                  // user code end
            } catch (java.lang.Throwable ivjExc) {
                  // user code begin {2}
                  // user code end
                  handleException(ivjExc);
            }
      };
      return ivjLabel1;
}

/**
 * Return the Label2 property value.
 * @return java.awt.Label
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private Label getLabel2() {
      if (ivjLabel2 == null) {
            try {
                  ivjLabel2 = new java.awt.Label();
                  ivjLabel2.setName("Label2");
                  ivjLabel2.setText("Message Area:");
                  // user code begin {1}
                  // user code end
            } catch (java.lang.Throwable ivjExc) {
                  // user code begin {2}
                  // user code end
                  handleException(ivjExc);
            }
      };
      return ivjLabel2;
}

/**
 * Return the Label3 property value.
 * @return java.awt.Label
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private Label getLabel3() {
      if (ivjLabel3 == null) {
            try {
                  ivjLabel3 = new java.awt.Label();
                  ivjLabel3.setName("Label3");
                  ivjLabel3.setText("Enter your Chat Alias:");
                  // user code begin {1}
                  // user code end
            } catch (java.lang.Throwable ivjExc) {
                  // user code begin {2}
                  // user code end
                  handleException(ivjExc);
```

```
            }
      };
      return ivjLabel3;
}

/**
 * Return the Panel1 property value.
 * @return java.awt.Panel
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private Panel getPanel1() {
      java.awt.GridBagConstraints constraintsLabel1 = new java.awt.GridBagConstraints();
      java.awt.GridBagConstraints constraintsToServer = new
java.awt.GridBagConstraints();
      java.awt.GridBagConstraints constraintsSendButton = new
java.awt.GridBagConstraints();
      java.awt.GridBagConstraints constraintsButton1 = new
java.awt.GridBagConstraints();
      java.awt.GridBagConstraints constraintsButton4 = new
java.awt.GridBagConstraints();
      if (ivjPanel1 == null) {
            try {
                  ivjPanel1 = new java.awt.Panel();
                  ivjPanel1.setName("Panel1");
                  ivjPanel1.setLayout(new java.awt.GridBagLayout());

                  constraintsLabel1.gridx = 0; constraintsLabel1.gridy = 0;
                  constraintsLabel1.gridwidth = 1; constraintsLabel1.gridheight = 1;
                  constraintsLabel1.anchor = java.awt.GridBagConstraints.WEST;
                  constraintsLabel1.weightx = 0.0;
                  constraintsLabel1.weighty = 1.0;
                  constraintsLabel1.insets = new java.awt.Insets(0, 2, 0, 0);
                  ((java.awt.GridBagLayout)
getPanel1().getLayout()).setConstraints(getLabel1(), constraintsLabel1);
                  getPanel1().add(getLabel1());

                  constraintsToServer.gridx = 1; constraintsToServer.gridy = 0;
                  constraintsToServer.gridwidth = 7; constraintsToServer.gridheight = 1;
                  constraintsToServer.fill = java.awt.GridBagConstraints.HORIZONTAL;
                  constraintsToServer.anchor = java.awt.GridBagConstraints.CENTER;
                  constraintsToServer.weightx = 1.0;
                  constraintsToServer.weighty = 1.0;
                  constraintsToServer.insets = new java.awt.Insets(0, 0, 0, 2);
                  ((java.awt.GridBagLayout)
getPanel1().getLayout()).setConstraints(getToServer(), constraintsToServer);
                  getPanel1().add(getToServer());

                  constraintsSendButton.gridx = 7; constraintsSendButton.gridy = 1;
                  constraintsSendButton.gridwidth = 1;
constraintsSendButton.gridheight = 1;
                  constraintsSendButton.anchor = java.awt.GridBagConstraints.EAST;
                  constraintsSendButton.weightx = 0.0;
                  constraintsSendButton.weighty = 1.0;
                  constraintsSendButton.insets = new java.awt.Insets(0, 0, 0, 2);
                  ((java.awt.GridBagLayout)
getPanel1().getLayout()).setConstraints(getSendButton(), constraintsSendButton);
                  getPanel1().add(getSendButton());

                  constraintsButton1.gridx = 0; constraintsButton1.gridy = 1;
                  constraintsButton1.gridwidth = 1; constraintsButton1.gridheight = 1;
                  constraintsButton1.fill = java.awt.GridBagConstraints.HORIZONTAL;
                  constraintsButton1.anchor = java.awt.GridBagConstraints.WEST;
```

```
                    constraintsButton1.weightx = 0.0;
                    constraintsButton1.weighty = 1.0;
                    constraintsButton1.insets = new java.awt.Insets(0, 2, 0, 0);
                    ((java.awt.GridBagLayout)
getPanel1().getLayout()).setConstraints(getButton1(), constraintsButton1);
                    getPanel1().add(getButton1());

                    constraintsButton4.gridx = 2; constraintsButton4.gridy = 1;
                    constraintsButton4.gridwidth = 1; constraintsButton4.gridheight = 1;
                    constraintsButton4.anchor = java.awt.GridBagConstraints.CENTER;
                    constraintsButton4.weightx = 1.0;
                    constraintsButton4.weighty = 0.0;
                    constraintsButton4.insets = new java.awt.Insets(0, 2, 0, 2);
                    ((java.awt.GridBagLayout)
getPanel1().getLayout()).setConstraints(getButton4(), constraintsButton4);
                    getPanel1().add(getButton4());
                    // user code begin {1}
                    // user code end
             } catch (java.lang.Throwable ivjExc) {
                    // user code begin {2}
                    // user code end
                    handleException(ivjExc);
             }
      };
      return ivjPanel1;
}

/**
 * Return the Panel2 property value.
 * @return java.awt.Panel
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private Panel getPanel2() {
      java.awt.GridBagConstraints constraintsLabel2 = new java.awt.GridBagConstraints();
      java.awt.GridBagConstraints constraintsChatRecord = new
java.awt.GridBagConstraints();
      if (ivjPanel2 == null) {
             try {
                    ivjPanel2 = new java.awt.Panel();
                    ivjPanel2.setName("Panel2");
                    ivjPanel2.setLayout(new java.awt.GridBagLayout());

                    constraintsLabel2.gridx = 0; constraintsLabel2.gridy = 0;
                    constraintsLabel2.gridwidth = 1; constraintsLabel2.gridheight = 1;
                    constraintsLabel2.fill = java.awt.GridBagConstraints.HORIZONTAL;
                    constraintsLabel2.anchor = java.awt.GridBagConstraints.WEST;
                    constraintsLabel2.weightx = 1.0;
                    constraintsLabel2.weighty = 0.0;
                    ((java.awt.GridBagLayout)
getPanel2().getLayout()).setConstraints(getLabel2(), constraintsLabel2);
                    getPanel2().add(getLabel2());

                    constraintsChatRecord.gridx = 0; constraintsChatRecord.gridy = 1;
                    constraintsChatRecord.gridwidth = 4;
constraintsChatRecord.gridheight = 1;
                    constraintsChatRecord.fill = java.awt.GridBagConstraints.BOTH;
                    constraintsChatRecord.anchor = java.awt.GridBagConstraints.CENTER;
                    constraintsChatRecord.weightx = 1.0;
                    constraintsChatRecord.weighty = 1.0;
                    ((java.awt.GridBagLayout)
getPanel2().getLayout()).setConstraints(getChatRecord(), constraintsChatRecord);
                    getPanel2().add(getChatRecord());
```

```
                        // user code begin {1}
                        // user code end
                } catch (java.lang.Throwable ivjExc) {
                        // user code begin {2}
                        // user code end
                        handleException(ivjExc);
                }
        };
        return ivjPanel2;
}

/**
 * Return the SendButton property value.
 * @return java.awt.Button
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private Button getSendButton() {
        if (ivjSendButton == null) {
                try {
                        ivjSendButton = new java.awt.Button();
                        ivjSendButton.setName("SendButton");
                        ivjSendButton.setLabel("Send");
                        // user code begin {1}
                        // user code end
                } catch (java.lang.Throwable ivjExc) {
                        // user code begin {2}
                        // user code end
                        handleException(ivjExc);
                }
        };
        return ivjSendButton;
}

/**
 * Return the TextField1 property value.
 * @return java.awt.TextField
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private TextField getTextField1() {
        if (ivjTextField1 == null) {
                try {
                        ivjTextField1 = new java.awt.TextField();
                        ivjTextField1.setName("TextField1");
                        // user code begin {1}
                        // user code end
                } catch (java.lang.Throwable ivjExc) {
                        // user code begin {2}
                        // user code end
                        handleException(ivjExc);
                }
        };
        return ivjTextField1;
}

/**
 * Return the ToServer property value.
 * @return java.awt.TextField
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private TextField getToServer() {
        if (ivjToServer == null) {
                try {
```

```
                    ivjToServer = new java.awt.TextField();
                    ivjToServer.setName("ToServer");
                    // user code begin {1}
                    // user code end
              } catch (java.lang.Throwable ivjExc) {
                    // user code begin {2}
                    // user code end
                    handleException(ivjExc);
              }
      };
      return ivjToServer;
}

/**
 * Called whenever the part throws an exception.
 * @param exception java.lang.Throwable
 */
private void handleException(Throwable exception) {

      /* Uncomment the following lines to print uncaught exceptions to stdout
*/
      System.out.println("--------- UNCAUGHT EXCEPTION ---------" +
exception);
      // exception.printStackTrace(System.out);
}

/**
 * Handle the Applet init method.
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
public void init() {
      super.init();
      try {
              System.out.println("Hey, starting the ChatApplet version
"+version);
              java.awt.GridBagConstraints constraintsPanel1 = new
java.awt.GridBagConstraints();
              java.awt.GridBagConstraints constraintsPanel2 = new
java.awt.GridBagConstraints();
              setName("ChatApplet");
              setLayout(new java.awt.GridBagLayout());
              setSize(426, 240);

              constraintsPanel1.gridx = 0; constraintsPanel1.gridy = 0;
              constraintsPanel1.gridwidth = 1; constraintsPanel1.gridheight = 1;
              constraintsPanel1.fill = java.awt.GridBagConstraints.BOTH;
              constraintsPanel1.anchor = java.awt.GridBagConstraints.CENTER;
              constraintsPanel1.weightx = 1.0;
              constraintsPanel1.weighty = 0.0;
              ((java.awt.GridBagLayout)
this.getLayout()).setConstraints(getPanel1(), constraintsPanel1);
              this.add(getPanel1());

              constraintsPanel2.gridx = 0; constraintsPanel2.gridy = 1;
              constraintsPanel2.gridwidth = 1; constraintsPanel2.gridheight = 4;
              constraintsPanel2.fill = java.awt.GridBagConstraints.BOTH;
              constraintsPanel2.anchor = java.awt.GridBagConstraints.CENTER;
              constraintsPanel2.weightx = 1.0;
              constraintsPanel2.weighty = 1.0;
              constraintsPanel2.insets = new java.awt.Insets(10, 2, 2, 2);
              ((java.awt.GridBagLayout)
this.getLayout()).setConstraints(getPanel2(), constraintsPanel2);
```

```
                this.add(getPanel2());
                initConnections();
                // user code begin {1}
                System.out.println("ChatApplet version "+version+" started");
                String hostname = getParameter("hostname");
                String hostport = getParameter("hostport");
                if (hostname != null) { host = hostname; }
                if (hostport != null) { port = new Integer(hostport).intValue();
}
                // user code end
        } catch (java.lang.Throwable ivjExc) {
                // user code begin {2}
                // user code end
                handleException(ivjExc);
        }
}

/**
 * Initializes connections
 */
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void initConnections() {
        // user code begin {1}
        // user code end
        getSendButton().addActionListener(this);
        getButton1().addActionListener(this);
        getButton3().addActionListener(this);
        getButton2().addActionListener(this);
        getButton4().addActionListener(this);
}

/**
 * main entrypoint - starts the part when it is run as an application
 * @param args java.lang.String[]
 */
public static void main(java.lang.String[] args) {
        try {
                java.awt.Frame frame;
                try {
                        Class aFrameClass =
Class.forName("uvm.abt.edit.TestFrame");
                        frame = (java.awt.Frame)aFrameClass.newInstance();
                } catch (java.lang.Throwable ivjExc) {
                        frame = new java.awt.Frame();
                }
                test.ChatApplet aChatApplet = new test.ChatApplet();
                frame.add("Center", aChatApplet);
                frame.setSize(aChatApplet.getSize());
                frame.setSize(400,240);
                aChatApplet.init();
                aChatApplet.start();
                frame.setVisible(true);

                aChatApplet.destroy();
        } catch (Throwable exception) {
                System.err.println("Exception occurred in main() of
java.applet.Applet");
        }
}

/**
 * This method was created by a SmartGuide.
```

```
 */
public void sendText( ) {
      //System.out.println("would send now - client");
      String request = ivjToServer.getText();
      out.println(request);
      return;
}
}
```

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;

public
class ChatServlet extends HttpServlet {

  int basePort = 0;
  int chatClient = 0;

  PrintWriter clientOut[] = null;
  int clientPort[] = null;

  String version = "1.03";

  public void init(ServletConfig config)
       throws ServletException
  {

    super.init(config);

    // here perhaps would set up the maximum number of clients
    int maxLength = 4;
    // initialize the servlet state
    clientOut = new PrintWriter[maxLength];
    clientPort = new int[maxLength];
    for (int i=0; i<clientOut.length; i++) {
      clientOut[i] = null;
      clientPort[i] = 0;
    }
    // set up the base port number for communications
    basePort = 9000;
  }

  public void destroy() {
    // close well known port
    log("about to destroy");
  }
  public void doGet(HttpServletRequest req, HttpServletResponse res)
       throws ServletException, IOException
  {

    int me;

    // set up for returning information
    log("entering doGet");
    ServletOutputStream bout = res.getOutputStream();
    res.setContentType("text/plain");
    // get client number and port (find unused output slot)
    synchronized (clientPort) {
      for (me=0; me<clientPort.length; me++) {
     if (clientPort[me] == 0) {break;}
      }
      // if found an open slot
      if (me < clientOut.length) {
     // reserve the slot
     clientPort[me] = basePort + me;
      }
    }
```

```
    // if did not find an open slot
    if (me == clientOut.length) {
      // indicate can't support another client
      log("doGet, overbooked");
      bout.println(""+0);
    } else {
      // send port back to applet
      log("doGet, accepting another client");
      bout.println(""+clientPort[me]);
      // start a new server thread to listen to client
      ServerListener chatter = new ServerListener(me, clientPort, clientOut,
this);
      log("doGet, about to start chatter");
      chatter.start();
    }
    // finish off
    bout.flush();
    return;
  }
}


class ServerListener extends Thread {

  int me;
  int myPort;
  PrintWriter clientOut[] = null;
  int clientPort[] = null;
  HttpServlet servlet = null;

  ServerSocket ssClient = null;
  Socket socket = null;
  InputStreamReader inISR = null;
  BufferedReader in = null;

  String version = "1.04";

  public ServerListener (int index, int[] port, PrintWriter[] out,
HttpServlet it) {
    super();

    // set up information
    it.log("ServerListener version "+version+": port="+port[index]);
    me = index;
    clientPort = port;
    myPort = clientPort[me];
    clientOut = out;
    servlet = it;
  }

  public void run()
  {

    boolean keepon = true;

    ServerSocket ssClient = null;
    Socket socket = null;
    InputStreamReader inISR = null;
    BufferedReader in = null;

    try {
      // wait for a client to connect
```

```
      servlet.log("ServerListener "+version+ " waiting for client connection
to: "+myPort);
      // get server socket
      ssClient = new ServerSocket(myPort);
      servlet.log("ServerListener: got ServerSocket("+myPort+")");

      // wait for a client to connect
      servlet.log("ServerListener: waiting for client connection client
"+me);
      socket = ssClient.accept();
      servlet.log("ServerListener: got client connection client");

      // get input and output streams (output with autoflush)
      inISR = new InputStreamReader(socket.getInputStream());
      in = new BufferedReader(inISR);
      synchronized (clientPort) {
     clientOut[me] = new PrintWriter(socket.getOutputStream(),true);
      }

      // get alias
      String alias = in.readLine();

      // send welcome message
      clientOut[me].println("Welcome to Chat 1.0, " + alias);

      // service messages
      while (keepon) {

     // get input from client
     servlet.log("ServerListener: waiting on client "+me);
     String request = in.readLine();

     // if quitting
     if (request.startsWith("QUIT")) {
       // finish up
       servlet.log("ServerListener: about to send to client");
       clientOut[me].println("Goodbye, " + alias);
       servlet.log("ServerListener: sent QUIT to "+me);
       keepon = false;
     } else {
       // send input to all clients
       synchronized (clientPort) {
         for (int i=0; i<clientOut.length; i++) {
           // if there is a client in this slot
           if (clientOut[i] != null) {
            // send to client
            servlet.log("ServerListener: about to send to client "+i);
            clientOut[i].println(alias + ": " + request);
            servlet.log("ServerListener: sent to client");
           }
         }
       }
     }
      }

      // close everything
      servlet.log("ServerListener: about to quit "+me);
      synchronized (clientOut) {
     clientOut[me].close();
     clientOut[me] = null;
     clientPort[me] = 0;
      }
```

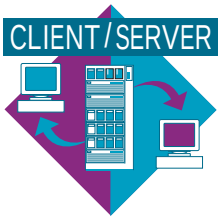
```
        in.close();
        inISR.close();
        socket.close();
        ssClient.close();

      } catch (IOException e) {
        servlet.log("ServerListener IOException: " +  e);
      }
    }

  }
```

The Chat Listener

```
package test;

import java.io.*;
import java.net.*;
import java.awt.*;
/**
 * This class was generated by a SmartGuide.
 *
 */
public class ChatListener extends Thread {
      BufferedReader inStream = null;
      TextArea outTarget = null;

      boolean done = false;


/**
 * ChatListener constructor comment.
 */
public ChatListener() {
      super();
}
/**
 * This method was created by a SmartGuide.
 * @param in java.io.BufferedReader
 * @param in2 java.awt.TextArea
 */
public ChatListener ( BufferedReader in, TextArea ta) {
      inStream = in;
      outTarget = ta;
}
/**
 * ChatListener constructor comment.
 * @param target java.lang.Runnable
 */
public ChatListener(Runnable target) {
      super(target);
}
/**
 * ChatListener constructor comment.
 * @param target java.lang.Runnable
 * @param name java.lang.String
 */
public ChatListener(Runnable target, String name) {
      super(target, name);
}
/**
 * ChatListener constructor comment.
 * @param name java.lang.String
 */
public ChatListener(String name) {
      super(name);
}
/**
 * ChatListener constructor comment.
 * @param group java.lang.ThreadGroup
 * @param target java.lang.Runnable
 */
public ChatListener(ThreadGroup group, Runnable target) {
      super(group, target);
```

```
}
/**
 * ChatListener constructor comment.
 * @param group java.lang.ThreadGroup
 * @param target java.lang.Runnable
 * @param name java.lang.String
 */
public ChatListener(ThreadGroup group, Runnable target, String name) {
      super(group, target, name);
}
/**
 * ChatListener constructor comment.
 * @param group java.lang.ThreadGroup
 * @param name java.lang.String
 */
public ChatListener(ThreadGroup group, String name) {
      super(group, name);
}
/**
 * This method was created by a SmartGuide.
 */
public void run( ) {

      // set up to run as long as user wants
      done = false;

      // if sufficient information to proceed
      if ((inStream != null) && (outTarget != null)) {
            // run forever
            while (!done) {
                  // wait for input
                  try {
                        String message = inStream.readLine();
                        // write it out
                        outTarget.append(message + "\n");
                        // check for user finished
                        if (message.startsWith("Goodbye")) {
                              // indicate we are finished
                              done = true;
                        }
                  } catch (IOException e) {
                        System.out.println("Server socket problem in
ChatListener: " +  e);
                  }
            }
      }

      // indicate finished
      done = true;
      return;
}
/**
 * This method was created by a SmartGuide.
 */
public void stopListening() {

      // wait for the run method to finish
      while(!done) {}

      return;
}
}
```

Applets and Servlets: A Smooth Blend



By Greg Flurry and Jeff Jilg

What problems are encountered in Java client/server programming? How can you use servlets to reduce communications overhead? Can applets and servlets be used to complement each other in a robust design? This article addresses these issues and more.

Java™ servlets require some careful planning and consideration during the design and development stage. Some issues such as initialization, multithreading, and synchronization using a Java applet/servlet-based chat application are described in this article. The article also includes a detailed breakdown of the chat application programming logic, coupled with a discussion of design choices.

We used Windows 95 and VisualAge™ for Java for developing this application, so we included a discussion of the development and debugging environment. After we developed the applications, the servlet portion was executed on AIX® to take advantage of the robust Java engine provided there. Future development could take place on AIX since the VisualAge for Java environment was demonstrated at the March 1998 JavaOne™ Conference, although product availability was not announced.

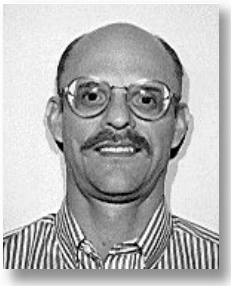
Java servlets appear destined to fill the role of CGI bin in many future Web server environments. Although many articles describing the importance of servlets and basic servlet development are available, most introductory articles ignore many of the interesting problems in designing and developing servlets. This article skips servlet basics and proceeds directly to some "intermediate" issues related to servlet design, and offers some tips on developing and debugging servlets.[1]

Design Considerations

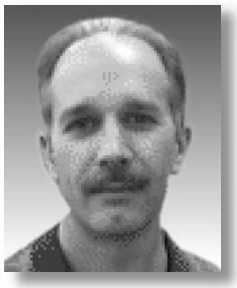
The goal of our application called Chat is to allow several users of a Web browser to enter information into a "chat room" and enable all members of that chat room to see everything that any member enters. This article examines the servlet portion of a Web-based chat application that was implemented using a Java applet and a Java servlet.

Why use a servlet in this application? Figure 1 shows the Chat design in which each individual user fires up an applet running in a browser and wants to chat. But where does the applet get the list of Chat participants? Most likely that list comes from a server.

Since each user's applet must connect to all other applets, the number of required connections $((n^2-n)/2)$ increases exponentially. For example, this design for four users requires six connections. The applet for Chat design must implement the complexity of discovering new connections and maintaining them, which makes it potentially unsuitable for low-end clients. This design may also create network bandwidth problems.



Greg Flurry



Jeff Jilg

[1] For basic information about servlets, see "Using Java Servlets on AIX" in the March 1998 issue of AIXpert.

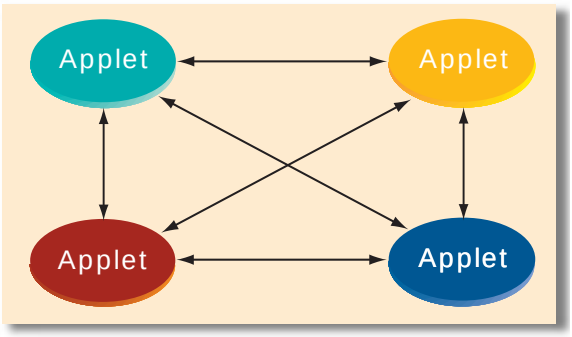


Figure 1. Applet-only Chat application

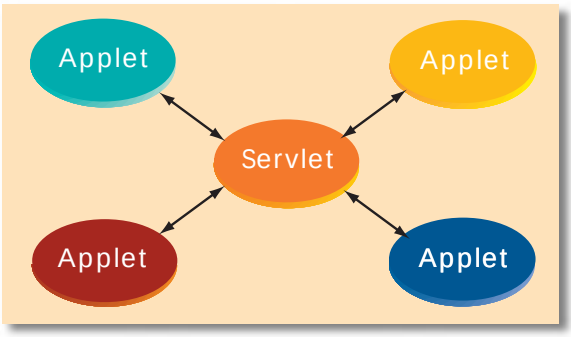


Figure 2. Applet-Servlet Chat application

Now consider the Chat design shown in Figure 2. In this example, each applet connects only to a central server; there is no need for the applet to even know about other participants. The applet is now light-weight and well suited for low-end clients. And there is only a single connection for each Chat participant—the number of connections *(n)* increases only linearly. Four clients require only four connections in this design. This decreases network bandwidth consumption, especially for large groups. For 10 users, the design uses just 10 connections compared to 45 connections required for the applet-only Chat.

Other advantages are also apparent. For example, the Chat server (in this case, the servlet running in a Web server) could support multicast datagram sockets. These sockets allow the sender to send a datagram to multiple receivers in a single transmission for potential further bandwidth reduction. This design uses unicast connection-based sockets. The Chat server could also support multiple chat rooms, maintain a log of all the chat room activity, or even perform load balancing transparently to the clients. However, our example does not implement these functions.

The Chat Application

The Chat application uses applets and a servlet as clients and server, respectively, to enable messages to be passed back and forth by users. A similar design could be incorporated into such network computing applications as remote debugging or system management consoles, or even Internet telephony (for example, for multiparty calls).

The following sections provide an overview of the Chat architecture, the states of the architecture, and termination of the application. The Java code in this article can be found in separate files on the CD-ROM. You can also copy and paste from Adobe Acrobat® by choosing the "ABC" icon and highlighting the code in Acrobat with your mouse.

Architecture Overview

At a high level, the Chat application runs an applet in a Web browser that presents a graphical user interface (GUI), shown in Figure 3. The GUI allows the user to enter text to send to the Chat server and to

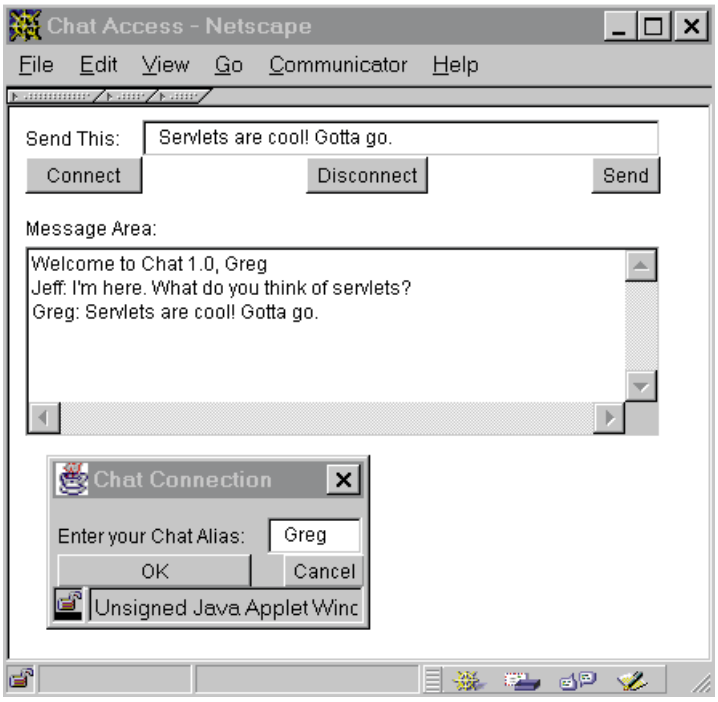


Figure 3. An Applet and its connect dialog

display text received from the Chat server. A connection-based socket handles communication between the applet and the servlet. The primary applet thread sends input from the user to the Chat server. For sending and receiving messages simultaneously, the applet spawns a second thread to listen for output from the servlet.

The servlet also involves multiple threads. The primary thread runs the standard `HttpServlet.doGet()` method, which is invoked for each instance of `ChatApplet` that connects to the Chat server. The `doGet()` method spawns a new thread that interacts with the corresponding applet on the client side; the `doGet()` thread then returns (ends). Any time a server-side listener receives input from a client, it sends that input, prefaced by an alias unique to that client, to all the existing client-side listeners. Details of this operation are described below.

Initialization

Figure 4 shows the HTML file that a user references to begin the Chat application. This file typically would be stored on the same server as the Chat servlet, although that is not required.

The parameters used to indicate the server hostname and port number allow the Chat administrator to easily modify this information as required. This also proved to be a useful debugging aid because it was easy to switch between testing the applet against the JavaSoft™ *srun* servlet environment and the Lotus Go™ 4.6.1 servlet environment.

Figure 5 shows part of the code for the `ChatApplet` class that runs in the primary applet thread. The GUI is constructed using VisualAge for Java with standard Abstract Windowing Toolkit (AWT) classes and the Java 1.1 event model (this code is not shown).

```
<html>
<title>Chat</title>
<body>
<h1>Chat 1.0</h1>
<applet code="test.ChatApplet.class" width=400 height=200>
<param name="hostname" value="flurry">
<param name="hostport" value="80">
</applet>
</body>
</html>
```

Figure 4. Chat application HTML file

```
public class ChatApplet extends Applet implements
java.awt.event.ActionListener {
  ChatListener chatListen = null;
  Socket client = null;
  BufferedReader in = null;
  InputStreamReader inISR = null;
  private Dialog ivjDialog1 = null;
  private TextArea ivjChatRecord = null;
  private TextField ivjAlias = null;
  private TextField ivjToServer = null;
  PrintWriter out = null;
  String host = "flurry";
  int port = 80;

  public void init() {
    super.init();
    try {
```

Figure 5. ChatApplet class code segments (continued on following page)

```
      System.out.println("ChatApplet version "+version+" started");

      /* lots of GUI initialization code not shown */

      // process the applet parameters
      String hostname = getParameter("hostname");
      String hostport = getParameter("hostport");
      if (hostname != null) { host = hostname; }
      if (hostport != null) { port = new Integer(hostport).intValue(); }
    } catch (java.lang.Throwable ivjExc) {
      handleException(ivjExc);
    }
  }

  public void connect( ) {

    String newPort = null;
    URL chatURL = null;

    // get alias
    String alias = ivjAlias.getText();
    ivjDialog1.dispose();

    System.out.println("Entering connect()");
    try {
      // point to server
      chatURL = new URL("http", host, port, "/servlet/ChatServlet");
    } catch (MalformedURLException e) {
      System.out.println("URL problem: " + e);
    }

    System.out.println("Formed URL");
    try {
      // contact server
      InputStream serverIn = (InputStream) chatURL.getContent();
      BufferedReader bSI = new BufferedReader(new
InputStreamReader(serverIn));
      System.out.println("Contacted server");
      newPort = bSI.readLine();
      int thePort = new Integer(newPort).intValue();
      System.out.println("Contacted servlet and got a response of " +
newPort);

      if (thePort == 0) {
        // can't connect to server
        ivjChatRecord.append("Sorry, cannot connect to server; try later.\n");
        return;
      }

      // attach to new server port
      client = new Socket(host, thePort);

      // get i/o streams
      inISR = new InputStreamReader(client.getInputStream());
      in = new BufferedReader(inISR);
      out = new PrintWriter(client.getOutputStream(),true);

    } catch (IOException e) {
```

Figure 5. ChatApplet class code segments (continued on following page)

```
        System.out.println("IOException:  " + e);
      }
      // send the alias string
      out.println(alias);

      // establish a listener for the server
      chatListen = new ChatListener(in, ivjChatRecord);
      chatListen.start();

      return;
    }

    public void sendText( ) {
      String request = ivjToServer.getText();
      out.println(request);
      return;
    }

    public void disconnect( ) {

      // stop the server
      out.println("QUIT");

      // now wait for the listener to quit
      chatListen.stopListening();

      // close all the things
      try {
        in.close();
        out.close();
        client.close();
      } catch (IOException e) {
        System.out.println("IOException:  " + e);
      }

      return;
    }

  }
```

Figure 5. ChatApplet class code segments

The GUI (Figure 3) presents Connect, Disconnect, and Send buttons, a `TextField` for user input to be sent to the Chat server, and a `TextArea` that displays all output sent from the Chat server.

By pressing Connect, a Chat connection `Dialog` (Figure 3) appears containing a `TextField` where the user enters an alias to identify that user in the chat room; the `Dialog` also contains an OK button. Pressing OK invokes `ChatApplet.connect()`, shown in Figure 5. First, `connect()` retrieves the alias for the user, then creates a URL using the applet parameters. These parameters point to the Chat servlet; `connect()` gets the content of the URL if the servlet has been initialized. If the Chat servlet has not been initialized, the servlet environment invokes the `init()` method for the `ChatServlet` class (Figure 6) that runs in the primary servlet thread.

The `init()` method in this example sets the length of a `PrintWriter` array and an `int` array. The array of `PrintWriters` communicates with the clients that attach to the server. The array of `ints` holds the numbers of the ports used in the socket connections. Since `ChatServlet` also uses this array as an indication of whether clients are attached, it is initialized to all

zeros. In addition, `init()` sets up the base number of the ports used for communications between the clients and the server.

If `ChatServlet` has been initialized, the servlet environment invokes the `doGet()` method. `ChatServlet.doGet()` first sets up for HTTP communication with the applet, then searches the port array for an open slot. Note that searching through the array and reserving a slot in the array are "synchronized."

Synchronization is critical to ensure the integrity of the port and `PrintWriter` arrays. Given the nature of the servlet environment, it is possible for multiple instances of `ChatServlet.doGet()` to run simultaneously, each trying to access the array of ports to find an open slot and reserve a slot. Without synchronization, two instances could end up finding the same empty slot, or possibly not finding an open slot—even though one really exists.

It is even more likely that one or more server-side listeners could be accessing the `PrintWriter` array and then try to use a `PrintWriter` simultaneously—with unpredictable (but most likely unpleasant) results. Although it is possible to synchronize the entire `doGet()` method, synchronizing only the sections dealing with the arrays provides better responsiveness. Synchronization is critical to maintain integrity of the `ChatServlet` application.

Following the search, `ChatServlet.doGet()` returns a string that contains either the port number to use for communications with the server, or zero, indicating that no more open slots are available for communications with the server. If `ChatServlet` has an open slot, `doGet()` starts a new thread running the `ServerListener` class to interact with the Chat applet.

It is necessary to spawn a separate thread to listen to the applet. Whenever a servlet `service()` method is invoked, the servlet environment spawns a new thread in which to run the method. During testing, however, we found that a Web server does not complete the HTTP transaction with the Web browser until the service method returns. The Web browser expects more input from

```
public
class ChatServlet extends HttpServlet {

  int basePort = 0;
  int chatClient = 0;
  PrintWriter clientOut[] = null;
  int clientPort[] = null;

  String version = "1.03";

  public void init(ServletConfig config)
       throws ServletException
  {

    super.init(config);

    // here perhaps would set up the maximum number of clients
    int maxLength = 4;
    // initialize the servlet state
    clientOut = new PrintWriter[maxLength];
    clientPort = new int[maxLength];
    for (int i=0; i<clientOut.length; i++) {
      clientOut[i] = null;
      clientPort[i] = 0;
    }
```

Figure 6. ChatServlet class code (continued on following page)

```
    // set up the base port number for communications
    basePort = 9000;
  }

  public void destroy() {
    log("about to destroy");
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
       throws ServletException, IOException
  {

    int me;

    // set up for returning information
    log("entering doGet");
    ServletOutputStream bout = res.getOutputStream();
    res.setContentType("text/plain");
    // get client number and port (find unused output slot)
    synchronized (clientPort) {
      for (me=0; me<clientPort.length; me++) {
        if (clientPort[me] == 0) {break;}
      }
      // if found an open slot
      if (me < clientOut.length) {
        // reserve the slot
        clientPort[me] = basePort + me;
      }
    }

      // if did not find an open slot
    if (me == clientOut.length) {
      // indicate can't support another client
      log("doGet, overbooked");
      bout.println(""+0);
    } else {
      // send port back to applet
     log("doGet, accepting another client");
     bout.println(""+clientPort[me]);
      // start a new server thread to listen to client
      ServerListener chatter = new ServerListener(me, clientPort, clientOut,
this);
      log("doGet, about to start chatter");
      chatter.start();
    }
    // finish off
    bout.flush();
    return;
  }
}
```

Figure 6. ChatServlet class code

the Web server and thus "spins." We chose to start another thread that interacts with the client so that the servlet's service method can return, thus completing the HTTP transaction as far as the Web server and Web browser are concerned.

Notice the `ChatApplet.connect()` call to `URL.getContent()` in which `URL.getContent()` returns an `Object`. The actual class is really a subclass of `Object`, specific to the content type. Testing showed that the actual class returned depends on the environment in which `ChatApplet` runs. IBM's VisualAge for Java and Sun's Java Development Kit (JDK) for Windows™ returned the same class, but Netscape™ Communicator returned a different class. Fortunately, both were subclasses of `InputStream`. We have not tested this in other browsers, and not all will necessarily return subclasses of `InputStream`, although it seems likely they would do so. Based on the observation, the `Object` is cast as an `InputStream`.

The `InputStream` is used to create a `BufferedReader` so that `connect()` can easily read a string returned from the Chat servlet. As mentioned above, the string is either "0", which indicates a failure, or a port number for contacting the server. If it receives a port number, `connect()` attaches to the server socket and retrieves an `InputStream` and `OutputStream` for receiving strings from and sending strings to the server. It then sends to the server the alias retrieved from the `Dialog`. Finally, `connect()` creates a new thread running a `ChatListener` class (shown in Figure 7) which receives input from the server using the `BufferedReader` and then echos it to the `TextArea`. When the `ChatServlet` thread instantiates the `ServerListener` thread (see Figure 8), it passes the port array and `PrintWriter` array to the constructor and an index into the array for this particular instance. It also passes an object reference to the servlet, which allows events to be logged in the `ServerListener` as needed (see more on this below). Once it has begun running, `ServerListener.run()` creates a `ServerSocket` listening on the port indicated by the index, and waits for `ChatApplet` to attach to the socket. After accepting the attachment, `ServerListener` gets the input and output streams necessary for communication. It places the `PrintWriter` created using the output stream into the `PrintWriter` array so that it can be accessed by all instances of `ServerListener`; this action must be synchronized for the reasons described above. `ServerListener` then reads the alias sent by `ChatApplet` and sends a "welcome" message to the `ChatListener` instantiated by the `ChatApplet`. `ServerListener` then begins its "steady state" operation.

Steady State

After the initialization activities, the `ChatApplet` thread waits for input from the user so it can send text to the `ServerListener` thread. The `ServerListener` thread waits for the text from the `ChatApplet` thread and echos input to all the `ChatListener` threads in the session. A `ChatListener` thread receives text from multiple `ServerListeners` and echos that text to the `ChatApplet TextArea`.

> ***VAJava is an excellent integrated development environment for developing Java applications, and especially good for developing user interfaces.***

To send text, the user must type into the `TextField` (Figure 3) presented by the `ChatApplet` and press the "Send" button, which invokes `ChatApplet.sendText()`, shown in Figure 5. Then, `sendText()`retrieves the text from the `TextField` and writes it to the socket's output stream, sending it to the corresponding `ServerListener` instance.

The `ServerListener.run()` method (Figure 7) waits for input from the `ChatApplet`. It receives the string and checks to see if the user wants to `QUIT`. If not, `run()` sends the received string to all the `ChatListeners` connected to the server by entering a loop and checking for valid `PrintWriters`. This action must be synchronized for the reasons described above. Then, `run()` writes the string to

```
public class ChatListener extends Thread {
  BufferedReader inStream = null;
  TextArea outTarget = null;

  boolean done = false;

  public ChatListener (BufferedReader in, TextArea ta) {
    inStream = in;
    outTarget = ta;
  }

  /* NOTE: other constructor forms not shown, i.e., the form
     with no parameters required for compatibility with
     JavaBeans
     */

  public void run( ) {

    // set up to run as long as user wants
    done = false;

    // if sufficient information to proceed
    if ((inStream != null) && (outTarget != null)) {
      // run forever
      while (go) {
        // wait for input
        try {
          String message = inStream.readLine();
          // write it out
          outTarget.append(message + "\n");
          // check for user finished
          if (message.startsWith("Goodbye")) {
            // indicate we are finished
            done = true;
          }
        } catch (IOException e) {
          System.out.println("Server socket problem in ChatListener: " +  e);
        }
      }
    }


// indicate finished
        done = true;

    return;
  }

  public void stopListening() {

    // wait for the run method to finish
    while(!done) {}

    return;
  }
}
```

Figure 7. ChatListener class code

each valid PrintWriter. Notice that the synchronization object is the port array rather than the PrintWriter array, yet we check for PrintWriters before writing to a client. The servlet must synchronize on the port array because that is the mechanism used to accept or reject clients. Due to nondeterministic scheduling, the port could be allocated, yet the socket connection is not yet made; thus, the PrintWriter is not yet instantiated.

The ChatListener.run() method (Figure 8) waits for input from ServerListeners. It simply gets a string from the input stream and appends it to the TextArea. The run() method also checks each received string for the indication to terminate. Since the ChatListener is the only entity writing to the ChatApplet TextArea, there are no synchronization issues.

Termination

The steady state operation described earlier continues until the user presses the Disconnect button, shown in Figure 3. This invokes ChatApplet.disconnect() (Figure 5), which sends the QUIT string to the server indicating that it wants to terminate the communication. disconnect() then waits for the ChatListener to stop listening and terminate by calling ChatListener.stopListening(), which

```
class ServerListener extends Thread {
  int me;
  int myPort;
  PrintWriter clientOut[] = null;
  int clientPort[] = null;
  HttpServlet servlet = null;

  ServerSocket ssClient = null;
  Socket socket = null;
  InputStreamReader inISR = null;
  BufferedReader in = null;
  String version = "1.04";

  public ServerListener (int index, int[] port, PrintWriter[] out, HttpServlet
it) {
    super();

    // set up information
    it.log("ServerListener version "+version+": port="+port);
    me = index;
    clientPort = port;
    myPort = clientPort[me];
    clientOut = out;
    servlet = it;
  }

 public void run()
  {

    boolean keepon = true;

    ServerSocket ssClient = null;
    Socket socket = null;
    InputStreamReader inISR = null;
    BufferedReader in = null;

    try {
      // wait for a client to connect
```

Figure 8. ServerListener class code (continued on following page)

```
servlet.log("ServerListener "+version+ " waiting for client connection to:
"+myPort);
      // get server socket
      ssClient = new ServerSocket(myPort);
      servlet.log("ServerListener: got ServerSocket("+myPort+")");

      // wait for a client to connect
      servlet.log("ServerListener: waiting for client connection client "+me);
      socket = ssClient.accept();
      servlet.log("ServerListener: got client connection client");

      // get input and output streams (output with autoflush)
      inISR = new InputStreamReader(socket.getInputStream());
      in = new BufferedReader(inISR);
      synchronized (clientPort) {
        clientOut[me] = new PrintWriter(socket.getOutputStream(),true);
      }

      // get alias
      String alias = in.readLine();

      // send welcome message
      clientOut[me].println("Welcome to Chat 1.0, " + alias);

      // service messages
      while (keepon) {

        // get input from client
        servlet.log("ServerListener: waiting on client "+me);
        String request = in.readLine();
       // if quitting
        if (request.startsWith("QUIT")) {
          // finish up
          servlet.log("ServerListener: about to send to client");
          clientOut[me].println("Goodbye, " + alias);
          servlet.log("ServerListener: sent QUIT to "+me);
          keepon = false;
        } else {
          // send input to all clients
          synchronized (clientPort) {
            for (int i=0; i<clientOut.length; i++) {
              // if there is a client in this slot
              if (clientOut[i] != null) {
                // send to client
                servlet.log("ServerListener: about to send to client "+i);
                clientOut[i].println(alias + ": " + request);
                servlet.log("ServerListener: sent to client");
              }
            }
          }
        }
      }

      // close everything
      servlet.log("ServerListener: about to quit "+me);
      synchronized (clientOut) {
        clientOut[me].close();
        clientOut[me] = null;
```

Figure 8. ServerListener class code (continued on following page)

```
        clientPort[me] = 0;
        }
        in.close();
        inISR.close();
        socket.close();
        ssClient.close();

      } catch (IOException e) {
        servlet.log("ServerListener IOException: " +  e);
      }
    }

}
```

Figure 8. ServerListener class code

does not return until the `ChatListener` is finished.

The `ServerListener.run()` method (Figure 7) receives the `QUIT` string to indicate that the user wants to disconnect. The `ServerListener` sends a "goodbye" message to only the terminating `ChatListener`. This string ensures that the `ChatListener's` socket read is satisfied, allowing it to terminate. The `ServerListener` closes the output stream, then frees its slot in both the port and `PrintWriter` arrays. These actions must be synchronized. The `ServerListener` then closes the input stream and the sockets, releasing them for the next user. Finally, it returns, thereby terminating the thread.

Once the `ChatListener.run()` method (Figure 8) receives the "goodbye" message from its corresponding `ServerListener`, it then indicates it is done. `ChatListener` returns, terminating the thread. Now `ChatApplet.disconnect()` (Figure 5) closes the socket input and output streams and the socket itself. From the user standpoint, the Chat application is now finished.

Developing the Chat Application

We coded and tested both the applet and the servlet in Windows 95. We used IBM's VisualAge for Java (VAJava) and Sun's JDK tools for developing and testing the applet. VAJava is an excellent integrated development environment for developing Java applications, and especially good for developing user interfaces. VAJava offers excellent debugging facilities that allow breakpoints, real-time examination of values, and so on. VAJava was demonstrated on AIX at the JavaOne Conference in March 1998, although no general availability was announced. You will probably want to execute the servlet on AIX to take advantage of the performance on a server. We think development on Windows 95 with subsequent execution on AIX can make good use of resources when appropriate.

We use a traditional text editor and Java Servlet Development Kit (JSDK) tools to develop and test the servlet. The JSDK *srun* tool was useful for real-time feedback from the servlet. It prints output from the standard console `System.out.println()` and from servlet `log()` methods. The drawback, of course, is that *srun* is not a real Web server and cannot deliver the HTML or the applet. This is fine when using either VAJava or the JDK appletviewer tool.

The initial applet development was done totally inside VAJava. VAJava happily runs multiple instances of the applet, while allowing us to use all the facilities of its integrated development environment. Following the testing inside VAJava was the testing with appletviewer to bring in the HTML file with its parameters. Appletviewer also provides real-time feedback on applet operations using the `System.out.println()`.

We performed the last stages of testing with the target deployment environment, a Java 1.1-enabled Web browser (Netscape Communicator 4.03) for running the applet,

and a servlet-enabled Web server (Lotus Go 4.6.1) for running the servlet. Debugging can become somewhat more difficult in the real world because debugging capabilities digress to the 1970s methods of "printing" progress and values. For this reason, a few `System.out.println()` statements will appear in the applet. The results of such statements appear in the Netscape Navigator™ 4.0 Java Console, providing a simple way to track the applet progress when the browser just appears to "hang."

A few `log()` statements appear in the servlet where you might expect to find `System.out.println()`. Both work fine when running in the JSDK *srun* tool, but `System.out.println()` does nothing when running in a Web server environment. Web servers, however, keep logs of various activities, such as servlet activity, if configured to do so. The `log()` statement causes the argument string to show up in the Web server's servlet log, conveniently prefaced by a timestamp and the servlet name—a valuable aid for debugging and also useful for simple recording of servlet activity in the deployment environment.

Conclusion

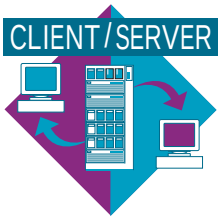
Java has captured the imagination of the network computing industry. Much has been written about Java applets for clients, but servlets prove that "Java ain't just for clients anymore!" The tips and techniques presented here should make it easier for you to begin developing and debugging your own servlets—the next step in the evolution of network computing.

If you are interested in creating an environment similar to ours, we suggest you first compile and execute the application we've provided here. Sometimes it's easier to get a sample like ours running instead of creating a brand new program and trying to both compile and configure it on a new system. *Note:* See the Table of Contents for this issue of *AIXpert* for the complete Java code for this application.

Greg Flurry, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Flurry is a senior technical staff member in the Server Development Division. His responsibilities, as part of the Systems Architecture & Technical Strategy team, include network computing and Java. He has a BS in Electrical Engineering from Vanderbilt University and an MS in the same field from the University of Kentucky.

Jeff Jilg, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. E-mail: jjilg@austin.ibm.com. Dr. Jilg is a senior architect currently responsible for Server Development Java technical strategy in the Systems Architecture & Technical Strategy team.. His MS in Computer Science from the University of Texas at El Paso complements his PhD from Texas A&M University in the same field.

Selecting Nodes for the IBM RS/6000 SP



By Clive Harris

This article describes how to select nodes for the IBM RS/6000 SP. The node types will be described in detail followed by some guidance in selecting SP nodes for particular applications.

Since the initial introduction of the IBM RS/6000 SP™, IBM has continued to announce new, faster node types with better price/performance. The SMP-based High Nodes for the RS/6000 SP are designed for commercial applications and, typically, are not used for scientific/technical applications when high floating-point performance is required. However, the new SMP nodes, based on the PCI bus, are suitable for both commercial and scientific/technical workloads.

The 332 MHz SMP Nodes, based on PCI architecture, are the most recent addition (April 1998) of node types supported by the RS/6000 SP. All previous nodes are based on Micro Channel® Architecture (MCA).

IBM now offers a variety of both uniprocessor nodes and symmetric multiprocessor (SMP) nodes for the SP system.

How to Choose an SP Node

Several node types are available for selection: Thin Node (uniprocessor or SMP), Wide Node (uniprocessor or SMP), and High Node (PowerPC™ 604e SMP). Within these types, there are various configurations, as shown in Figure 1.

With so many node choices, how can you select the most appropriate node in any particular case? There are several factors to consider.

The first step is to review the required capacity for adapters, internal disk, and memory. This can help determine whether a Thin Node (which often provides better performance) has sufficient capacity for the needed requirements. Once that is determined, consider the performance requirements and how the application can exploit the different types of nodes. Even though a Thin Node may have sufficient capacity in terms of adapters, disk, and memory, other requirements may necessitate selecting another node (for example, a need for using Micro Channel adapters). Be sure to include all Micro Channel adapters that will be required, both now and in the future. For high availability solutions, it often is necessary to include additional adapters for redundancy.

Applications have different uses for the internal disk. Commercial applications use the internal disk on each node only for the AIX® operating system, paging space, application executables, and any temporary data. Typically, any critical customer data is stored on external disks so the disks can be “twin-tailed” to provide high availability if the primary node fails. The internal disk cannot be accessed from another node if the node itself has failed. Some scientific/technical applications have a lower volume of data, and high availability is not so important. For these applications, the internal disk is the appropriate solution.

Node Type	RS/6000 Equivalent	Processor	Current / Non-current[1]
Thin 1	Model 390	POWER2 (66 MHz)	Non-Current
Thin 2	Model 39H	POWER2 (66 MHz)	Non-Current
Thin P2SC	N/A	POWER2SC (120 MHz)	Non-Current
Thin P2SC	Model 397	POWER2SC (160 MHz)	Current
Thin PCI	Model H50	PowerPC 604e (332 MHz)	Current
Wide 1	Model 590	POWER2 (66 MHz)	Non-Current
Wide RPQ	Model 59H	POWER2 (66 MHz)	Non-Current
Wide 2	Model 591	POWER2 (77 MHz)	Non-Current
Wide P2SC	Model 595	POWER2SC (135 MHz)	Current
Wide PCI	Model H50	PowerPC 604e (332 MHz)	Current
High 1 (2-way)	Model R40	PowerPC 604 (112 MHz)	Non-Current
High 1 (4-way)	Model R40	PowerPC 604 (112 MHz)	Non-Current
High 1 (6-way)	Model R40	PowerPC 604 (112 MHz)	Non-Current
High 1 (8-way)	Model R40	PowerPC 604 (112 MHz)	Non-Current
High 2 (2-way)	Model R50	PowerPC 604e (200 MHz)	Current
High 2 (4-way)	Model R50	PowerPC 604e (200 MHz)	Current
High 2 (6-way)	Model R50	PowerPC 604e (200 MHz)	Current
High 2 (8-way)	Model R50	PowerPC 604e (200 MHz)	Current

[1] Current indicates the most recent node available; non-current implies that particular node has been superseded by newer models with superior price/performance

Figure 1. Common RS/6000 SP nodes

Memory is another factor to consider regarding node capacity. The SP nodes have specific memory slots available (these are separate for Micro Channel or PCI slots) but not every combination of memory will be available, particularly if memory cards already are in use in an SP node.

Occasionally, selecting a node can be a straightforward decision. For example, some applications may require a certain number of Micro Channel or PCI adapters, or a certain amount of memory. The more difficult decisions will be based on performance and, in particular, when to use uniprocessor nodes or SMP nodes in the SP system.

Node Characteristics

Each of the various nodes have certain functions and characteristics that make them well-suited to certain applications.

The 332 MHz SMP Nodes. The design of these PCI-based nodes has high RAS characteristics and excellent attention to detail. They can coexist with all previously supported nodes within any type of SP frame, provided the frame has been upgraded for new power supply requirements.

These PCI nodes are available in two formats: PCI Thin and PCI Wide. (The remainder of this article refers to the 332 MHz SMP Nodes as PCI Thin and PCI Wide Nodes.) These nodes are similar to previous Thin and Wide nodes: they occupy the same space in a frame and the same rules apply with respect to sharing switches between frames. The attraction is better price/performance using the latest technology.

The new PCI nodes are suitable for a wide range of applications, including all commercial and most scientific and technical applications. In some cases, the PCI Wide Nodes demonstrate superior performance to the PCI Thin Nodes, even though they share the same type and number of processors. This is because the Wide Node has a second PCI bridge (or controller) that attaches the additional eight PCI slots to the internal (MX) bus.

The performance differences can be significant in some I/O-related tasks. For example, with Serial Storage Architecture

(SSA) disk performance, the data rate could increase from about 40 MB/sec on a PCI Thin Node to approximately 70 MB/sec on a PCI Wide Node. This alone might provide a good reason to choose the Wide Node rather than the Thin Node.

Micro Channel Thin Nodes. The Micro Channel Thin Nodes have four slots available with a built-in Ethernet adapter that can be used for the RS/6000 SP Ethernet, which is mandatory.

PCI Thin Nodes. The PCI Thin Nodes also have a built-in Ethernet adapter and SCSI adapter for the internal disk. In addition, the optional SP Switch adapter does not use a PCI slot: it plugs into the MX bus and does not require a PCI slot. This leaves the two PCI slots available for normal use.

Micro Channel Wide Nodes. The Micro Channel Wide Nodes do not have a built-in Ethernet adapter. One slot always will be used for this purpose, leaving seven available slots.

PCI Wide Nodes. The PCI Wide Nodes have a built-in Ethernet adapter and SCSI adapter for the internal disk. As with the PCI Thin Node, the optional SP Switch adapter does not use a PCI slot: it plugs into the MX bus and does not require a PCI slot. Ten PCI slots are available for normal use. The PCI Wide Node consists of a PCI Thin Node with an expansion cabinet—fitted at the manufacturing plant—to provide additional expansion for adapters and disk. It is not possible to field upgrade from a PCI Thin Node to a PCI Wide Node.

Micro Channel High Nodes. The High Nodes, which use only a Micro Channel bus, have 16 adapter slots, but only 14 are usable because one is used for an Ethernet adapter and one for a SCSI adapter. In many cases, each node will require an SP Switch adapter, an alternate (user) network adapter, an Ethernet or Token-Ring adapter, plus some kind of disk adapter (SSA or SCSI).

In practice, Thin Nodes can have the available slots filled quickly. When High Availability solutions (HACMP) are required, Thin Nodes almost certainly will not be appropriate.

Figure 2 shows the Micro Channel and PCI adapter capacity.

Internal Disk Capacity

Internal disks contain the AIX operating system, any temporary files, and possibly application executables. For many commercial applications, the real data will be stored on external disks (for example, SSA or RAID) outside the SP nodes to allow access from another node if the primary node fails.

Paging space can be split between internal disks (the ROOTVG Volume Group) and external disks (SSA disks). It is supported to mirror the ROOTVG Volume Group using standard AIX mirroring, which may lead to additional internal disks. Currently, the RS/6000 SP does not support an

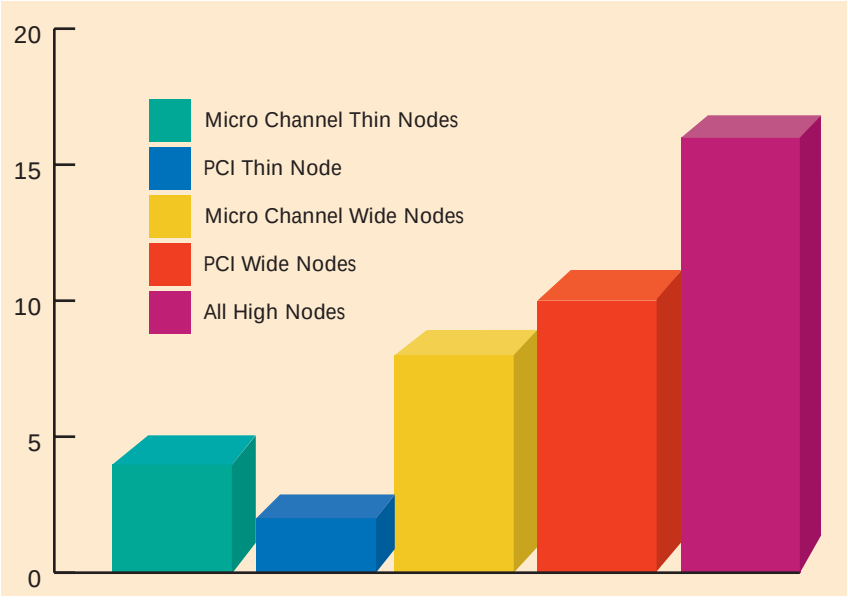


Figure 2. Micro Channel and PCI adapter slots

external ROOTVG Volume Group on SSA disks or booting off external SSA disks.

Maximum Memory Capacity

Memory cards are available for 32 MB, 64 MB, 128 MB, 256 MB, and 512 MB. Different memory cards can be used within the SP system, but certain combinations of cards cannot be used.

Thin Node. In a Thin 1 Node, only 64 MB, 128 MB, and 256 MB memory cards can be used. A Thin 1 Node allows a maximum of two memory cards of any type. The SP Thin 2 (39H) Node uses a combination of both memory cards and memory SIMMs (placed on the CPU board within the SP node). The memory card must match the CPU memory SIMMs card. Any of the 32 MB, 64 MB, 128 MB, and 256 MB memory cards can be used with the corresponding memory SIMMs on the CPU board. One of each always will be used.

Thin POWER2 Super Chip Node. This node has four memory card slots, which allows up to 1 GB of total memory. It doubles the memory capacity of a Thin Node and increases the memory bandwidth by 80%—to 1.9 GB per second. Four memory cards will give optimal memory performance in a Thin POWER2 Super Chip Node.

SP MCA Wide Node. The total number of memory cards must be two, four, or eight. If two memory cards are installed, they must be the same type of card. If more than two memory cards are installed in a Wide Node, they must be installed in matching groups of four identical memory cards. Optimal memory performance is obtained by using four memory cards in an SP Wide MCA Node.

PCI Node. This node uses memory cards specific to PCI nodes or systems. The cards cannot be interchanged with memory from MCA nodes or systems.

It is important to consider upgrade options and performance implications when choosing the initial memory configuration.

Other Non-Application Specific Selection Factors

There are several other factors that can help you decide whether to select a smaller number of more powerful nodes or a larger number of less powerful nodes. Those factors include: high availability, system and application management, application isolation, and system partitioning.

High Availability

High availability within the SP environment can be achieved in three ways:

- Provide a system that has a high degree of reliability and is less prone to error. The reliability of the standard RS/6000 components is proven, and AIX software is one of the best in the UNIX® world. In addition, the SP system, frame, and switch have additional functionality to provide higher reliability and availability.
- Provide ways within the SP to achieve concurrent maintenance when applications are not available due to regular maintenance and housekeeping work.
- Provide a failover functionality, to be instigated in the event of any kind of failure. The SP uses High Availability Clustered Multiprocessing (HACMP) software and component redundancy for this functionality. Redundancy within the SP is provided by RAID or mirrored disks, alternate networks, additional adapters, and spare or additional nodes used in the event of failure.

The new PCI nodes are suitable for a wide range of applications, including all commercial and most scientific and technical applications.

If high availability is important, consider using several less powerful nodes in combination, so if any single one fails, the rest will carry on. It is possible to build a highly

available solution in this manner without dramatically increasing the cost of the overall system. This also avoids the weakness of using a single SMP system to run all applications. A single High Node uses two drawers within the SP frame, whereas those same two drawers can house four Thin Nodes. This provides a greater degree of redundancy, and in some environments, a better solution.

These types of decisions should be based on a combination of required performance, price/performance of the different options, and a view of the level of availability required.

A system is less complex if it supports fewer applications. Also, a uniprocessor is less complex than a powerful SMP processor. If a failure occurs, you lose only part of your system. For example, losing one of the four Thin Nodes means that 75% of the total system and performance still is available. Since the possibility of four simultaneous failures causing the loss of the entire system is extremely remote, the four-node system gives several options for concurrent maintenance and housekeeping not afforded by a single node.

Two examples illustrate these considerations. Some applications, such as Lotus Notes®, will see no performance benefits from anything more powerful than a 4-way SMP system because of their design. Therefore, within the same space, you could implement either one 4-way High Node, or four Thin Nodes. The Thin Nodes provide better performance and also provide higher availability in terms of redundancy.

SAP™ is another example in which having more nodes is usually better than putting all of the workload onto one larger node. Running the application servers on separate nodes isolates and guarantees a given level of performance. Running everything on one large node (even if it were a powerful SMP-based node) will not give this level of flexibility or availability.

System and Application Management

As a rule of thumb, fewer nodes will make the SP system easier to manage. In reality, manageability depends on the type of applications running on the RS/6000 SP. For applications in which system management is not a complex task, fewer nodes will not be so important. But when the system is being used to consolidate servers, fewer nodes mean reduced management overhead.

SP system management is much easier compared to a similar number of separate servers. However, organizations often select a smaller number of nodes within the SP system. In fact, customers using their SP system for server consolidation or client/server applications (for example, SAP) have been the earliest adopters of SMP nodes.

Any parallel database environment that has mixtures of nodes is likely to make system and application management more complex. Running a parallel database on eight thin uniprocessor nodes and adding two SMP nodes into that parallel database requires more work to get adequate performance from the additional power afforded by the SMP nodes. In fact, for some parallel databases, it may be necessary to run additional workload on the SMP nodes to enable them to handle a greater workload. This will increase management and administration overhead.

Any parallel database environment that has mixtures of nodes is likely to make system and application management more complex.

The ideal in a parallel database environment is equal power nodes across the board. But in practice, as enterprises move to SMP nodes, the combination of uniprocessor and SMP nodes will occur.

Application Isolation

Isolation is one factor that may lead you to additional, less powerful nodes. Generally, UNIX systems are not good at running multiple applications on the same system. Shared resources often cause contention, and it is impossible to prevent the effect of one application on another.

For example, if applications A, B, and C run on the same UNIX system, and application A is used heavily, its increased workload on the system will have a detrimental effect on applications B and C. It is impossible to prevent this on a single (uniprocessor or SMP) system, which directly contrasts to an MVS® system, which is well-suited to running multiple applications simultaneously.

Different applications can run on different nodes within the SP system, with guaranteed performance and service levels for the applications on each separate node. This is important when running tests, development systems, or different relational database management system (RDBMS) software.

This capability is critical to SAP. Multiple nodes allow separate application servers, database servers, update and batch servers, and separate instances of SAP to run on the same SP system. A cost-effective approach may be to use a larger number of less powerful nodes to allow for application isolation rather than to run everything on a small number of SP nodes.

System Partitioning

System partitioning within the SP enables "logically" separate SP systems. SP environments can be isolated completely from each other, as in two production environments or a production and test environment.

Although SP system partitioning can assist with migration from AIX Version 3 to AIX Version 4 in an SP environment, this is not its primary purpose. The SP system no longer supports AIX Version 3. Combinations of AIX Versions 4.1, 4.2, and 4.3 on the same SP system or within a system partition are easy to run using the coexistence support in Parallel System Support Program (PSSP). For this reason, it is not necessary to partition the SP.

Today, only a few environments require system partitioning. Generally partitioning is done according to defined rules based on Switch Chip boundaries. SP Switch adapters are cabled into the SP Switch at the bottom of the SP frame and connected to various Switch Chips according to their position in the frame. The result is that certain nodes are located within a Switch Chip boundary.

The PCI Nodes follow the same rules of system partitioning and sharing a switch between frames as the existing Thin Nodes and Wide Nodes.

Frame Limitations and Node Upgrade Options

Both the old and the new tall frames support all types of nodes, including the PCI Nodes. The entry-level short frame also supports the PCI Nodes. Since the PCI Nodes require an upgraded power supply, the new tall frames include the power supply.

> ***The CPU upgrade path is popular for organizations that have High Nodes or PCI Nodes within their SP system and require more computing power.***

Both short, entry-level frames and the full-height SP frames support High Nodes (both PowerPC 604 based and PowerPC 604e based). The older entry-level High Performance Switch (LC-8 Switch) does not support High Nodes or PCI nodes. All of the other switches, such as the older, full 16-port version High Performance Switch (HiPS), and newer SP switches (both the full 16-port version and the entry-level SP Switch-8) support the High Nodes.

The PCI Nodes can be used only with the SP Switch (either the 8-port or the 16-port version) and not with the older High Performance Switch. Currently a single SP system cannot support more than 64 High Nodes.

Upgrading High Nodes (two to four or four to eight processors) is straightforward and cost-effective. Similarly, PCI Nodes can be upgraded from two to four CPUs. A PCI node begins with two processors. Thin Nodes are sold only in pairs (a full drawer).

The CPU upgrade path is popular for organizations that have High Nodes or PCI Nodes within their SP system and require more computing power.

The most difficult considerations involve organizations that are running parallel database applications, but would like to switch

to SMP nodes rather than a combination of node types. These upgrades must be carefully planned. It may not be possible to simply remove a Wide Node and replace it with a High Node, because the new High Node will take up two drawers in the SP.

Selecting Nodes Based on Performance

It is important to choose the correct nodes for the applications and have the ability to exploit the most appropriate architecture—serial or parallel, uniprocessor or SMP processor.

The performance measures shown in Figure 3 are estimates of OnLine Transaction Processing (OLTP). The numbers represent performance relative to an IBM RS/6000 Model 250, and simply show a comparison in performance between nodes.

Each type of application requires a different set of factors to consider in terms of performance of the various nodes.

Performance Considerations

It is important to differentiate between serial and parallel applications when evaluating the performance of SP nodes as well as other factors.

Serial and Parallel Applications. A single serial application will use only one node within the SP. It is common to run numerous serial applications on the same SP system, either as server consolidation or client/server applications. Server consolidation applications generally run several different applications on discrete nodes within the SP to reduce management and support costs. These applications often can be good contenders for running on an SMP node, particularly if the application has been designed and written to exploit an SMP architecture. This is not always the case with typical commercial database applications.

Each application running on an SMP node must exploit the SMP architecture, otherwise only one processor out of eight (in an 8-way SMP node, for example) is utilized. For example, Lotus Notes will typically scale only to a 4-way SMP system and will not take advantage of a 6-or 8-way system.

Node Type	Relative OLTP Performance
Thin	13.0
Thin	23.3
Thin POWER2 SuperChip (120 MHz)	5.8
Thin POWER2 SuperChip (160 MHz)	6.7
Thin PCI (2-way)	17.9
Thin PCI (4-way)	32.8
Wide	13.9
Wide	24.5
Wide POWER2 SuperChip	5.8
Wide PCI (2-way)	17.9
Wide PCI (4-way)	32.8
High 1 (2-way)	5.8
High 1 (4-way)	10.0
High 1 (6-way)	14.5
High 1 (8-way)	19.2
High 2 (2-way)	9.3
High 2 (4-way)	17.0
High 2 (6-way)	23.8
High 2 (8-way)	30.6

Figure 3. Relative OLTP performance

The High Nodes do not support scientific/technical applications. Since the floating-point performance and price/performance of the POWER2 Super Chip is superior to the PowerPC 604e processor used in the High Node, the POWER2 nodes are the obvious choice for these applications. A scientific or technical environment may need to run particular "commercial" workloads, such as file servers or communications gateways; these commercial workloads (non-floating-point workloads) can run on the SP.

The newer PCI Nodes (using the 332 MHz PowerPC processor) have good commercial (integer) and floating-point performance for various application types.

Several considerations for commercial parallel applications are written to exploit more than one node within the SP system. First, determine if the application supports an SMP (each individual node within the parallel system supports an SMP architecture). Second, make decisions about how to mix nodes within a parallel application. Selecting nodes of equal power and similar configuration may be a good

solution. However, if several POWER2 Thin Nodes already are running an application, and more power is needed by adding SMP nodes, these different types of nodes may lead to increased management/administration overhead.

Another scenario might be starting a serial application, such as a database server for an SAP solution. This serial database might run on one SMP node as it grows from two to eight processors within one High Node. For additional performance, you might implement a parallel database for an additional growth path. However, what would you add to the existing configuration of one large High Node? Initially, you may consider another 2-way or 4-way High Node, but again, this would create a performance imbalance between the two nodes—which you want to avoid.

A good solution would be an additional 8-way SMP node, but the cost may be too high. An alternative to upgrading the High Node would be to introduce a second node earlier, while it was still a 4-way or 6-way node. This would lead to two 4-processor nodes.

Scalability. The scalability of any application clearly depends on many factors. One is the ratio of CPU computing work compared to the communications workload. Nodes must be distributed evenly; and how well this is achieved depends on the compute/communications ratio.

Nodes of Varying Power. Another factor is how parallel databases handle nodes of varying power within the total configuration. Each database product that runs on the SP system continues to add more function. How a database handles different types of nodes will be a deciding factor in evaluating the options for SP system growth.

Price/Performance. The price/performance of the current nodes tends to be similar at any given time. You can select the most appropriate nodes for a particular application based on the best choice for the enterprise, rather than choosing particular nodes based purely on price/performance.

Server Consolidation (Serial Applications)

The most important consideration in a server consolidation environment is whether the specific applications support SMP processors and take advantage of such nodes. Some applications may exploit an SMP system only partially, possibly taking advantage of a 4-way or 6-way SMP system, but not an 8-way.

Server consolidation applications may be among the best for running on SMP nodes within the SP. Adopting SMP nodes can minimize the number of required nodes. But to isolate applications, you still have enough nodes within the SP for some level of potential redundancy from an availability point of view.

Server consolidation applications use only one node, and as the performance of the application grows, the maximum performance available on any individual node can be a gating factor. SMP nodes allow an application to grow from the smallest uniprocessor nodes to a powerful SMP node with greatly increased performance. Since AIX is binary compatible with each node, the same application can run unchanged from one node to another.

> ***SMP nodes allow an application to grow from the smallest uniprocessor nodes to a powerful SMP node with greatly increased performance.***

SP Switch Performance

Different nodes have varying performance characteristics with respect to the SP Switch or the High Performance Switch. This performance can impact node selection, either for nodes that act as a gateway or for scientific/technical applications in which SP Switch performance may be critical.

Performance in real life can vary dramatically, depending on tuning parameters and whether TCP/IP or User Space protocols are used across the SP Switch. Packet sizes have dramatic results on actual performance: a large packet size may give increased overall performance.

Sample Scenarios

Parallel Databases: Shared Disk Systems (I/O Shipping) Oracle Parallel Server.

For serial applications, the latest versions of the Oracle® RDBMS run on the RS/6000 SMP systems and SMP nodes within the SP. The SMP node is a good solution for serial applications.

A parallel database comprised of several SMP nodes running Oracle also may be a good solution if SMP nodes are required. A parallel database provides a highly scalable solution, supporting a multi-terabyte database. The level of scalability far exceeds a single SMP system or server.

A parallel database within an SP system can be built using uniprocessor or SMP nodes as the building blocks. The preferred option is running a parallel database with four 4-way SMP nodes rather than 16 POWER2 Wide Nodes, because this option has fewer nodes to maintain, manage, and administer.

The situation is not so clear when mixing nodes—uniprocessor and SMP nodes—within an Oracle parallel database solution. The most recent Oracle software has an intelligent optimizer that will indicate when certain nodes complete their work faster than others. It will give those nodes additional work the next time around. Over time, the SMP should accept more of the workload when compared with other less powerful nodes.

Parallel Databases: Distributed Disk Systems (Function Shipping)

DB2® Universal DataBase (UDB). In a serial database, the IBM RDBMS software (DB2/6000™) can run on SMP systems and exploit an SMP node within the SP. SMP nodes may be a good option for serial applications. UDB works differently in a parallel database, such as the Oracle Parallel Server. DB2 UDB partitions the data and spreads it across the nodes. As with all parallel databases, this is best suited to using equal power and similarly configured nodes across the entire environment.

DB2 UDB also works well with a parallel database running on SMP nodes. However, each database product achieves its objectives differently. The DB2 software maintains a catalogue node that tracks where the required data resides, and ships the SQL request to the node in question. With these types of parallel "function shipping" databases, the SQL sends a request to the remote node within the SP, as opposed to sending just the disk I/O request (I/O Shipping).

Informix XPS. PSSP 2.3 or later must be used to run a parallel Informix XPS database on SMP nodes within the SP system. The Informix parallel database uses the higher performing User Space protocol for communication across the SP Switch. The SMP nodes can be used when running a serial database.

OnLine Transaction Processing (OLTP)

Since OLTP usually involves many users, it is well-suited to an SMP environment. In effect, parallelism is built in to OLTP workloads because they typically process a large number of relatively light transactions. Response times are critical and expected to be fast. Applications such as CICS™, SAP, and other client/server implementations run well in the SP system, where individual components can run on uniprocessor or SMP nodes.

> ***A parallel database within an SP system can be built using uniprocessor or SMP nodes as the building blocks.***

When the system reaches a transaction rate of approximately 10 transactions per second, a TP Monitor can improve throughput and overall performance significantly. Good performance and scalability for OLTP applications on the SP system depend on careful system design and tuning.

The amount of I/O performed on a local disk versus I/O performed on a remote disk will affect scaling performance across nodes in a parallel database. The more disk activity that is local, the better the scaling. TPC-C transactions, for example, perform less than 15% of their I/O remotely. In contrast,

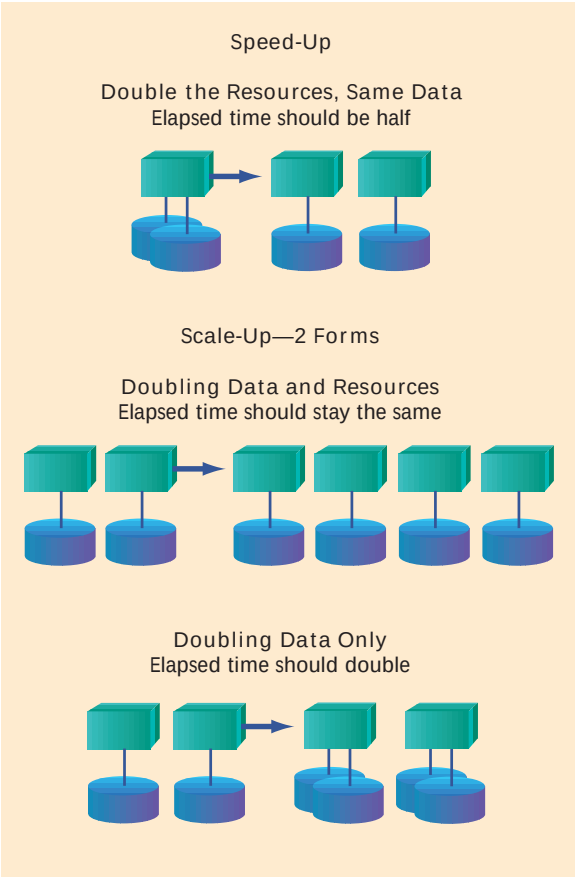


Figure 4. Speed-up and scale up

an application performing 50% of its I/O to remote disks would not scale well.

Decision Support

Decision Support System (DSS) applications, or business intelligence applications, use data warehouses or data marts for centrally storing data. Various applications access the data to gain business intelligence.

DSS applications require larger, more complex queries with fewer users. The database will run on a single node or in parallel if the most powerful SMP node is not enough. Database volumes with hundreds of gigabytes require a parallel database. Combining processors within an SMP system results in a diminishing return on investment as processors are added. However, the scaling factor for DSS workloads probably is better than for OLTP (TPC-C).

In a TPC-C workload, kernel activity requires a large percentage of time, and the operating system sometimes locks. In contrast, a DSS workload spends more time in the application, which results in better scaling, assuming that the application does not introduce an additional level of locking. Therefore, an SMP node running a TPC-D workload generally displays better scaling growth—from two to four or eight processors—when compared with a typical TPC-C type of workload. Such factors will depend on the actual application or database in use.

Figure 4 shows the processors required for scalability and performance.

SAP R/3. SAP R/3 is a well-suited and popular OLTP application. It can be advantageous to isolate some SAP components from others. For example, isolating the application servers and running them on separate SP nodes can guarantee service levels and provide redundancy. At the database server level, SAP (with Oracle, for example) can exploit a uniprocessor node, an SMP node (if more power is required), or a parallel Oracle database. High availability also can be provided for the database server; otherwise, this will be a single point of failure.

Lotus Notes/Domino™. Since Lotus Notes needs to replicate its databases between Notes® systems, it is well-suited to the performance characteristics of the SP Switch. The SP system management advantages also suit this type of application where customers often need to utilize multiple servers. Although Lotus Notes supports SMP processors, it rarely uses more than four or six processors within a single SMP system. Therefore, only an SMP node with four CPUs would be selected.

Using a single High Node with four processors for Lotus Notes may not be the only option. Within the same two drawers, you also could choose four Thin Nodes, which would provide more overall CPU performance and additional high-availability options.

The Lotus Notes Advanced Server can run more than one Lotus Notes “instance” on the same SMP system and also utilize clusters more effectively, as in the SP system. The Lotus Notes Partitioning

function allows multiple Notes servers to run on one SMP system, although each server uses its own memory section. Future requirements include the ability to "throttle back" certain Notes servers and provide dedicated resources to minimize the effect of one Notes server on another.

The Notes clustering functionality allows automatic load balancing and automatic rerouting of clients if a failure occurs. The SP system allows Lotus Notes to be implemented by using the HACMP software.

Scientific/Technical

Typical scientific/technical applications will not exploit the High Nodes; they need the higher floating-point performance of the POWER2 nodes or the PCI Nodes. The individual PowerPC 604 processors in High Nodes do not have high floating-point performance and would not be used for scientific/technical applications. The performance of the processors in the latest PCI Nodes for floating-point applications are enhanced greatly and better suited to single-precision floating-point workloads.

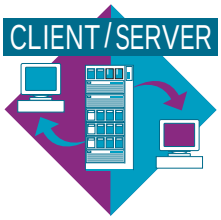
These applications take advantage of a parallel system such as the RS/6000 SP. In general, combinations of POWER2 uniprocessors are likely to be the best solution.

Summary

This article described the various SP nodes, how applications exploit them, and factors that you should consider as you select nodes for the SP system.

Clive Harris, IBM Corporation, Europe, Middle East, Africa (EMEA) SP Business Manager, EMEA Mid-Range Server Business, IBM Basingstoke, U.K.
E-mail: clive_harris@uk.ibm.com.

Printing in a Network Station Environment



By Eddie Ho, Ravi Mandavi, Steve Heracleous, and Claude Bechard

Network Station is a convenient access client for your mission-critical application environment. It is essential that printing—both locally and remotely—be part of the overall consideration in your solution deployment. This article covers all aspects of printing technology in a Network Station business environment.

IBM Network Station™ provides total support for both locally attached or network-attached printers. Network Station printing can complement your RS/6000™ printing environment and can interoperate as a client to your RS/6000 print server or vice versa. When you are accessing an application on the server, client-side printing completes the total user experience in a production environment.

Local printers can have a serial or parallel interface attached directly to a Network Station. Remote printers can be attached directly to the network with an interface card, to another Network Station, or to a serving host system. The system administrator can use Network Station Manager software to centrally define and manage the capability of each printer attached to a Network Station. Network Station Manager Release 3 provides the following:

- Supports network printing through full implementation of Line Printer Requester (LPR)/Line Printer Daemon (LPD) RFC1179 protocol for locally submitted print jobs to be sent to remote print servers as well as receiving print jobs from the remote client
- Supports a set of consistent dialog boxes for accessing printing resources with enhanced local applications, such as NC Navigator™ Browser, eSuite WorkPlace™, and terminal emulators
- Enables locally executed Java™ applications to have a full printing environment using the printing capability of Java Virtual Machine (JVM) 1.1.4
- Allows Double-Byte Character Set (DBCS) printing for both 3270 and 5250 application environments
- Supports both serial and parallel printer connections
- Supports both PostScript® and Printer Control Language (PCL) printer data stream

In general, printing technology is specific to platforms, operating systems, and applications. This fact, combined with the many different types and features of printers available today, results in complexity and difficulty in setup, migration, and support. Figure 1 shows the printing capability in a Network Station environment with heterogeneous systems.

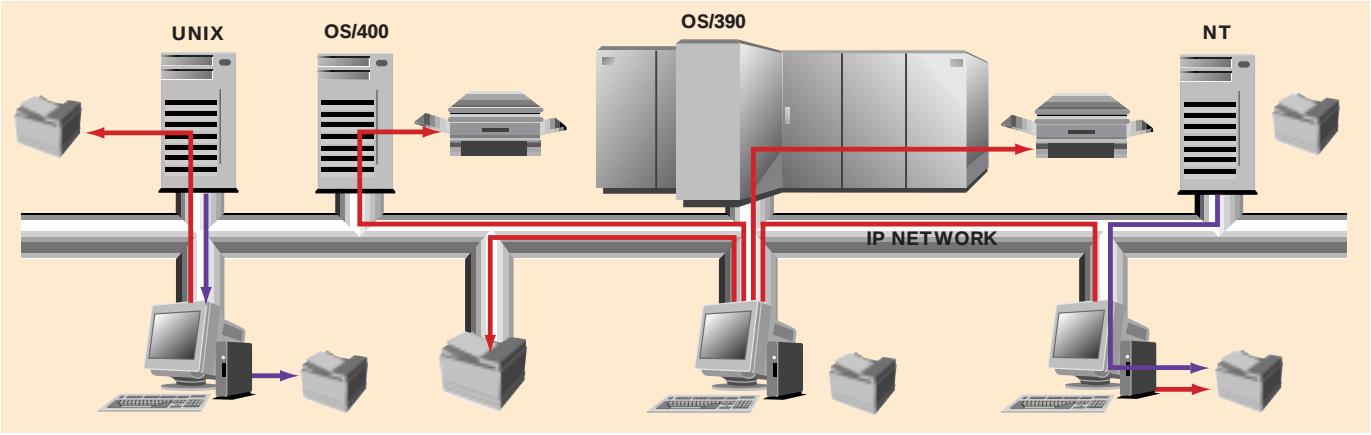


Figure 1. Typical business environment with printing support

Overview of the Print Subsystem

Network Station Release 3 technology focuses primarily on the printing function. It is structured and represents a service layer for all inbound and outbound print requests. Its major components include:

- Line Printer Requester Daemon (LPRD): A socket-based daemon that transmits outbound print jobs to remote print servers and routes data from the Print Service Module using LPR/LPD protocol
- Line Printer Daemon (LPD): A daemon for handling inbound print jobs and data routing to the Print Service Module
- Print Service Module: The core of print processing and routing, which handles all inbound and outbound print streams
- Serial Print Daemon (SERIALD): Socket-based daemon that supports data transmission from a client application, such as the Print Service Module, to a parallel or serial printer based on the socket port used
- Printer Selector: Provides the selection of preconfigured printers
- DBCS Converter: Converts internal character code to DBCS graphics

Figure 2 shows a high-level view of the overall inbound and outbound data flow of a print job stream. On the left side, the Network Station generates print requests to the external system-attached printer or network printers. The Printer Selector panel shows the choice of printers. The LPRD can support both streaming and non-streaming protocol. The protocol used is determined in real time by negotiating servers during session initialization.

Streaming protocol is more efficient and uses less memory because print data is sent in real time as it is generated. Only the RS/6000 with AIX® 4.3 and OS/400® support this mode.

The right side of the figure illustrates how print job streams originating from an application host are targeted to a Network Station-attached printer. The LPD daemon routes the incoming print data to the SERIALD daemon to print locally. The Network Station can handle concurrent local and remote printing.

The center of the figure shows a local application initiating a print request using the Print Service Module to send print data to a local or remote printer. The application prompts the user with a uniform Print Selector panel, which lists the preconfigured printers determined by the system administrator using the Network Station Manager.

After the user chooses the printer, the application uses the Print Service Module to send data either to SERIALD module for local printing or to the LPRD for remote printing. If the attached printer is

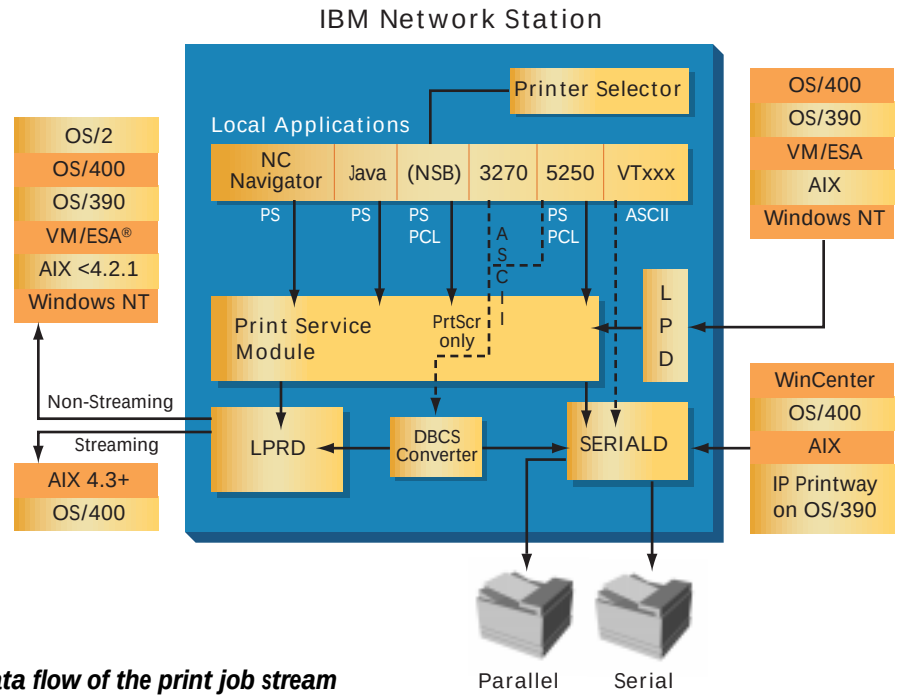


Figure 2. Data flow of the print job stream

disconnected or does not have the power on, an error dialog shows possible corrective action. Figure 3 illustrates this dataflow.

When a remote application host sends a print job to the Network Station, it uses the LPR/LPD protocol to communicate to the LPD running on the Network Station. The LPD sends the incoming print data to the SERIALD daemon using the print APIs in the Print Service Module. If the local printer is not powered on, an error dialog displays for corrective action. Figure 4 summarizes the dataflow.

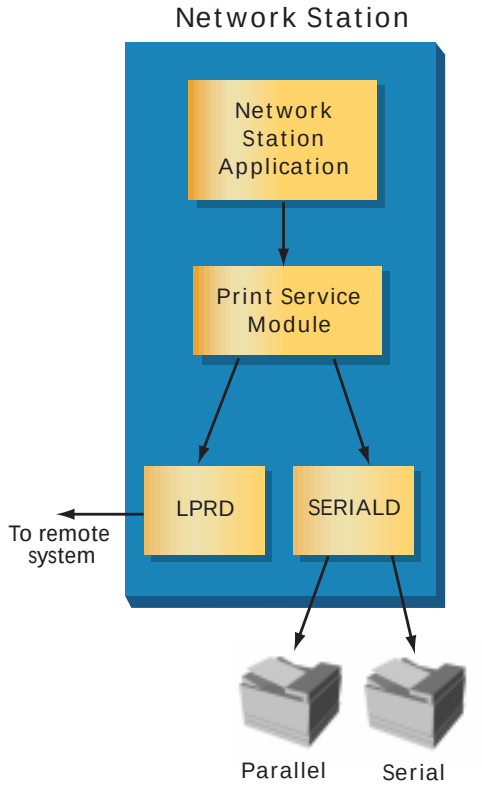


Figure 3. Dataflow to the Print Service Module

Printer Selector

The Printer Selector dialog box is used for locally initiated printing. When the end user generates a print request, a Printer

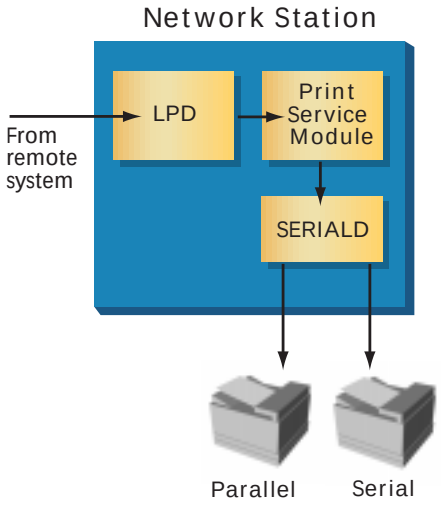


Figure 4. Dataflow to the Network Station

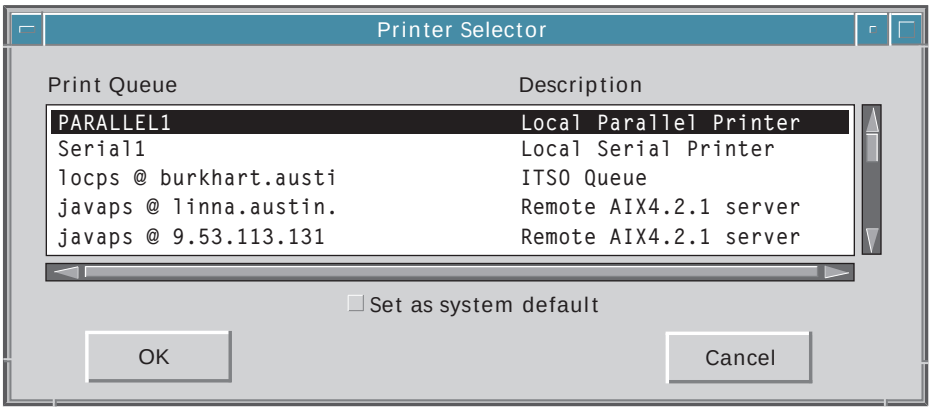


Figure 5. Dialog for printer selection

Selector dialog with a list of printers (local and remote) is displayed, as shown in Figure 5. A check box option allows users to designate a default printer. Once a default printer is set, all applications running under the same login context use this printer as the default, unless the dialog is used to reset it.

Printing Support for Applications

All Network Station applications support both local and remote printing capabilities. While all of these applications use the same Printer Selector dialog to choose printers, they use different first-level dialogs, depending on the application environment. This printer interface is provided for the following applications: 5250/3270 emulator, NC Navigator browser, and various Java applications.

Printing from the 5250 or 3270 emulator. Screen Print is triggered via the Print Screen option of the pull-down menu or using the "Print Screen" key. The emulator, which supports both single and double character sets, can generate ASCII, PCL, or PostScript print streams based on the type of printer attached. The first-level print dialog for the 3270 applications is shown in Figure 6. The Select a Printer button triggers the Printer Selector dialog.

Another major 3270 print architecture is host-directed printing, represented by TN3270e architecture. This function is currently not supported, but it is being planned for the future. This type of printing allows the S/390® host to send Systems Network Architecture (SNA) LU1 and LU3 print streams directly to a printer attached to the Network Station.

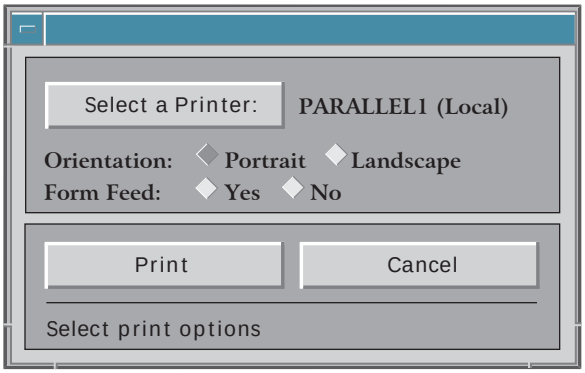


Figure 6. Select a Printer option

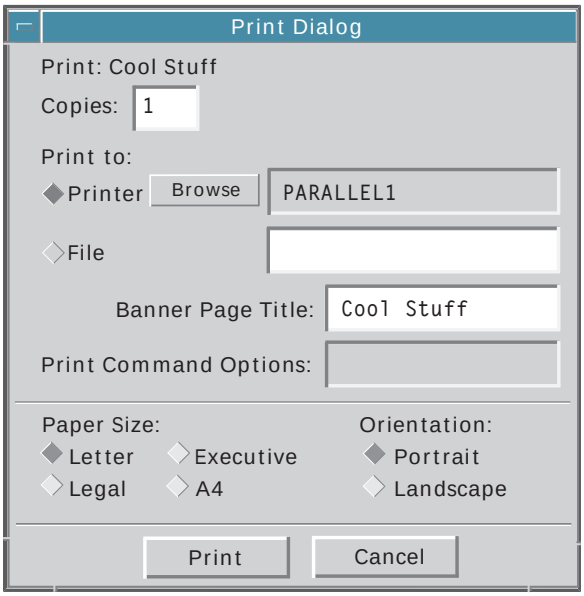


Figure 7. The Browse button

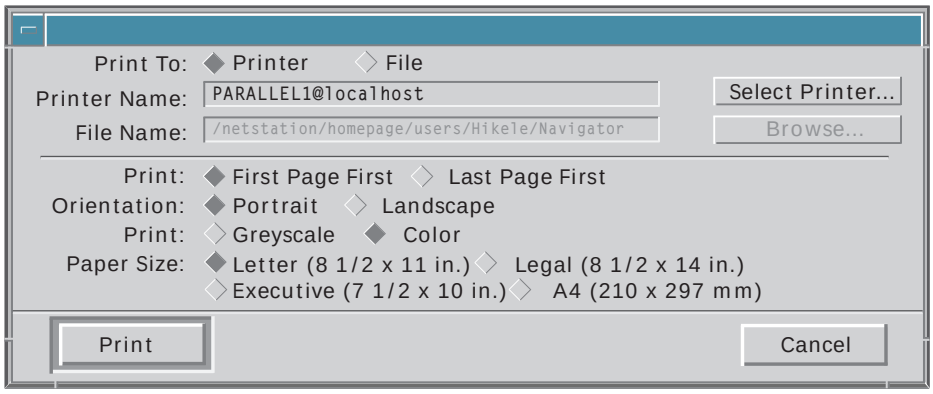


Figure 8. NC Navigator browser printing

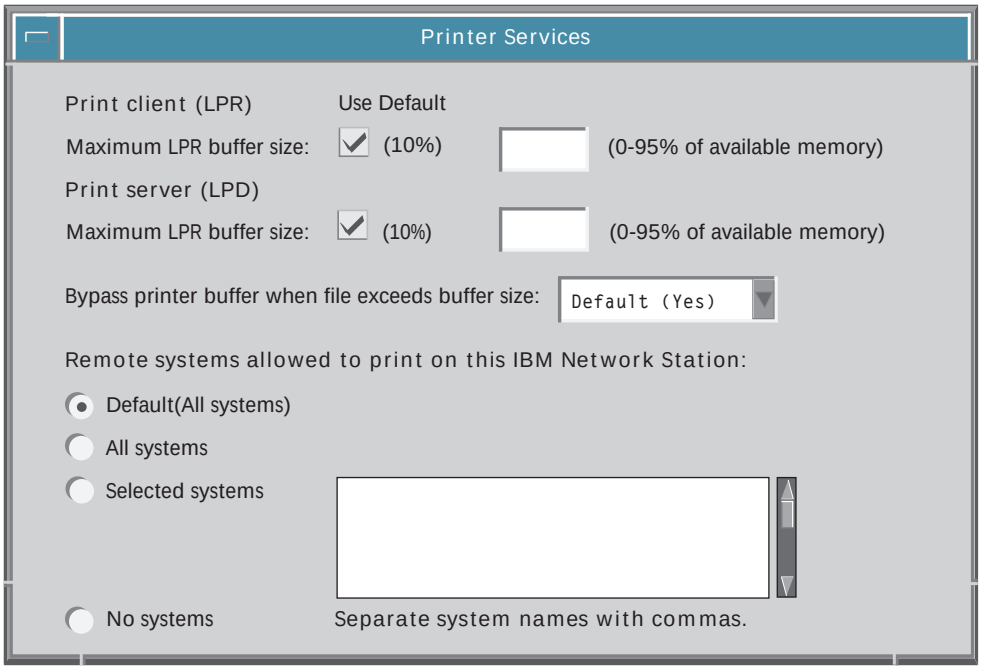


Figure 9. Caching limits for LPRD and LPD

Printing from Java applications. The Java 1.1.4 environment supports printing in PostScript format. Printing from Java applets, such as office productivity applets on the eSuite desktop, pops up the first-level print dialog (see Figure 7). Clicking the Browse button displays the Printer Selector dialog. Other print formats and orientation are available for selection.

Printing from the NC Navigator browser. The browser displays the first-level print dialog when the user selects Print from the pull-down menu File option (shown in Figure 8). This example shows a PostScript-only print stream. In this dialog, the Select Printer button brings up the Printer Selector dialog for further selection.

Network Station Manager Printer Configuration

Network Station Manager controls all resources within the Network Station domain. Using the Network Station Manager, the desktop administrator manages and updates all end-user environments. The desktop administrator also manages access to the printer table, remote host access, and LPD/LPR caching limits for the LPRD and LPD for each user (see Figure 9).

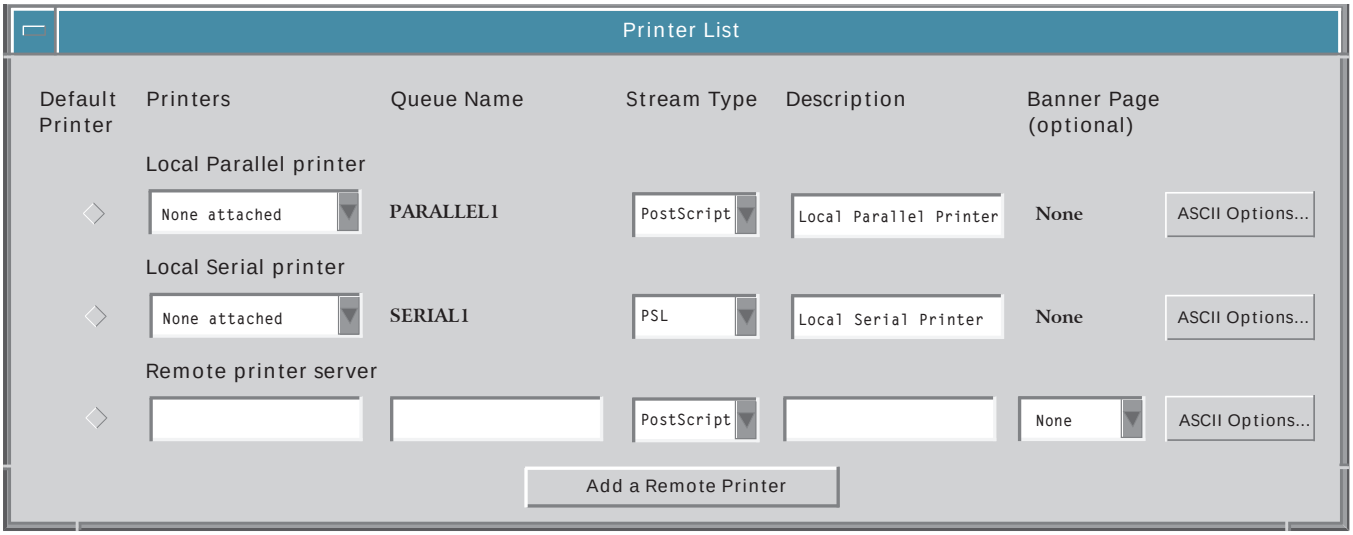


Figure 10. Printer list

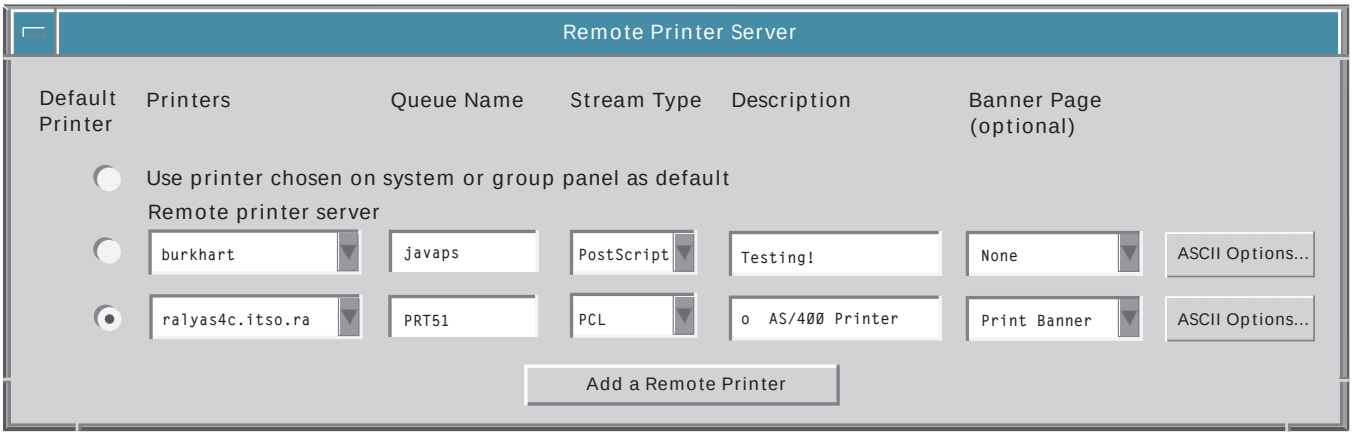


Figure 11. Remote printer server

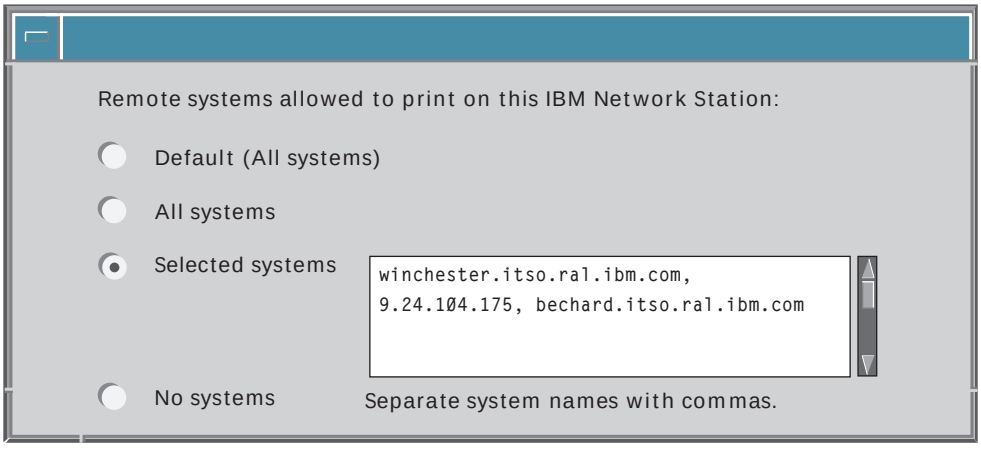


Figure 12. Print access to remote hosts

The Network Station Manager supports the printer list dialog, which contains entries for the available print queue—both local and remote. Two standard default print queues are PARALLEL1 and SERIAL1 for locally attached parallel and serial printers. It is possible to enable the local print queue and change characteristics as necessary. Figure 10 shows the printer list dialog.

The remote printer server dialog (see Figure 11) is used to make the Network Station aware of available remote printer queues. Simply complete the required fields and select the Add a Remote Printer button.

Network Station Manager also allows an administrator to control access to the local printer by remote hosts. Access may be granted or denied on a per-host or all-hosts basis (see Figure 12).

Finally, Network Station Manager is used to control cache/buffer sizes used by the LPRD and LPD daemons. In the Printer Services dialog (Figure 9), these values may be entered as a percentage of system memory. A minimum of 5 KB per daemon is recommended.

Conclusion

Network Station Release 3 provides full print capability, including locally generated print jobs or inbound print jobs from remote systems. This positions Network Station as a print-enabled network device and provides a total solution in a business environment.

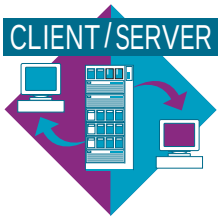
Eddie Ho, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Ho is a programming consultant in the Advanced Solution Center, Network Computer Division. He has a BS in Computer Science from the University of Wisconsin and an MS in Computer Science from North Dakota State University.

Steve Heracleous, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Heracleous is an advisory programmer in the Advanced Solution Center, Network Computer Division. He has a BS and MS in Computer Engineering from the University of Texas in Austin.

Ravi Mandava, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Mandava is an advisory programmer in the Network Computer Division Development. He has a BS in Mechanical Engineering and a MS in Computer Science from Osmania University in India.

Claude Bechard, IBM Corporation, 1001 Winstead Drive, Cary, NC 27513. Mr. Bechard is a senior technical marketing support representative at the International Technical Support Organization in Raleigh. He is responsible for producing redbooks on the IBM Network Station and the Windows NT™ platform. He is an IBM Canada employee with 29 years of service with IBM, and holds a degree in Mechanical and Industrial Engineering from the University of Montreal.

Shared Memory Programming on the RS/6000 SP



By David Klepacki and Xianneng Shen

This fourth article in a series about parallel programming on the RS/6000 SP focuses on the shared memory programming model using PAMS. The first article introduced the basic architecture of the RS/6000 SP and the three fundamental parallel programming models.[1] The second article focused on the message-passing model using MPI/LAPI,[2] and the third article focused on the data-parallel model using HPF.[3]

Shared memory programming is now possible on the RS/6000 SP™ with software called Parallel Application Management System (PAMS) from Myrias, Inc. PAMS enables programmers to develop load-balanced parallel applications from serial applications without data distribution, task management, or modification of source code.

The shared memory programming model can be subdivided into two categories: virtual shared object (VSO) and virtual shared memory (VSM). In the virtual shared object model, multiple processes globally share data objects. Regions of memory are reserved for shared access and, by necessity, involve the usual locking mechanisms to coordinate access among the processes.

This type of programming model has many implementations: Linda and its Tuplespace (Scientific Computing Associates), Treadmarks (Rice University), Global Arrays (Pacific-Northwest National Lab), and ABC++ shared regions (IBM Corporation), to name a few. To make it run in parallel, this model typically requires the addition of new function calls, data constructs, and in some cases, language extensions to an existing serial application.

The virtual shared memory model, on the other hand, assumes that the entire address space is shared. The way it shares memory pages is logically similar to that of the virtual memory manager (VMM) of the AIX® operating system.

When a page fault occurs in AIX, the memory address currently being referenced is not valid or available in main memory. It must be fetched from a paging file, usually located on a hard disk. Similarly, in a parallel computer using VSM, a page fault occurs when a referenced memory address is not valid or available in main memory. However, in this case, it might not be valid or available on the processing node, so it must be fetched from another processing node where it is available. (A processing node refers to one or more processors that share a single address space.)

With VSM, each processing node is a paging device for the parallel application and its associated virtual address space. Since the VSM model is a natural extension of the operating system's VMM, it does not

[1] Klepacki, David and Shen, Xianneng. "Parallel Programming Models on the IBM RS/6000 SP." *AIXpert*, September 1997.
[2] Shen, Xianneng and Klepacki, David. "Message-Passing on the RS/6000 SP." *AIXpert*, December 1997.
[3] Klepacki, David and Shen, Xianneng. "Data Parallel Programming with HPF on the RS/6000 SP." *AIXpert*, March 1998.

require additional constructs in an application. Often, serial applications can be made parallel on the RS/6000 SP without modifications to the source code, apart from comment-based directives to the VSM environment.

PAMS

PAMS is the only commercially supported VSM system in the marketplace today. This software implements a virtual shared memory environment for the RS/6000 SP. Using PAMS, the complex, distributed memory architecture of the RS/6000 SP looks to the programmer like one very large shared-memory multiprocessor system.

PAMS will run on any combination of RS/6000 SP node types—thin, wide, or high. Because of its design, PAMS can automatically balance the load of the application among the different performance capabilities of the nodes. Pages of memory naturally migrate to the processors that need them. Each processor in a parallel job can work at its own pace, making it unnecessary for programmers to perform any explicit data distribution or task management. In fact, the only programmer requirement is to identify the parallel opportunity in the application and to add the appropriate comment-based directives.

Memory Models

The three basic ways to share memory among processing nodes having different physical address spaces include the following:

- Uncached Store Coherent Memory (USC): Migrates a single memory region among the processors as needed
- Cached Store Coherent Memory (CSC): Replicates the memory regions as necessary, but with only one writable region
- Loop Coherent Memory (LC): Replicates the memory regions as necessary with all such regions being simultaneously writable

The USC model is simple and does not suffer any consistency problems that are associated with replicated data. However, this model does not always perform well. Its major drawback is the potential for "jitter." Jitter occurs when multiple processing nodes reference the same memory region so that this region ping-pongs among them. This happens because the USC memory model contains no other copies of the memory region. It is necessary to use semaphores or other locking mechanisms to control this condition.

> ***Using PAMS, the complex, distributed memory architecture of the RS/6000 SP looks like one very large shared-memory multiprocessor system.***

The CSC model reduces the jitter problem by having multiple read-only copies of a memory region available to all other processing nodes. Unlike the USC model where a memory region needs to be migrated for both read and write requests, the CSC model only needs to migrate memory regions for write requests.

Memory replication introduces another problem: the potential for data inconsistency. For example, updates to shared memory regions may not become visible to some copies of the memory regions in the other processing nodes when they are needed. If this occurs, the application may end up using "stale" data in the processing nodes that have the replicated memory regions.

Another form of this problem can occur, known as a *race condition*, resulting in data inconsistency among the nodes. This happens when a memory region is updated before its previous update is replicated to the same memory locations in the other processing nodes. Synchronization and memory-locking techniques must be used to circumvent these types of problems.

The USC and CSC models both suffer from *false sharing*. This occurs when multiple processing nodes concurrently request to update memory locations that do not overlap, but reside on the same memory

page. The actual memory locations being modified are different, but since they are part of the same fundamental shared memory unit (page), they are treated as if they were the same memory locations being modified; hence, the term "false" sharing. This creates a performance penalty in these models when logically there should be none. False sharing incurs unnecessary jitter in these models.

The LC model can avoid some of these problems, because different processing nodes can concurrently write to every copy of a memory region. All updated memory regions are then merged in hierarchical fashion into a single, valid memory region when needed. Memory pages do not migrate in this model; therefore, no jitter or false sharing occurs, only replication and merging. Data inconsistency and race conditions can still occur in the LC model, but only through programmer error, such as declaring a loop to be parallel when it is not.

Lastly, the LC model does allow multiple processing nodes to update the same memory locations concurrently. This is valid only when the user identifies these memory locations as being part of a global reduction operation (see Dependency Violations below).

PAMS supports all three types of memory models but prefers the LC model, which in most cases, is sufficient for many applications. In addition, the memory models can coexist in the same application. Some applications may require fine tuning of the memory management by incorporating either the USC or CSC model, but this requires additional support functions and often results in the need to modify source code. On the other hand, the LC model is implemented entirely with comment-based directives.

PAMS Development

PAMS is a complete environment that consists of precompilers, runtime libraries, and an extensive set of GUI-based tools (such as a debugger and various monitors) that support the major programming languages: C, C++, and Fortran. It is important to note that PAMS uses IBM AIX native compilers, since this retains the highest optimization of code on the RS/6000 processors. The programming interface is quite small, but deceptively powerful (see Figure 1).

Basic Directives

Although PAMS can create parallel programs, it is necessary to identify parallel capability in the source code. The LC model has two forms: blocks of code that can be performed in parallel and loops with independent iterations. Figure 2 shows an example of two parallel blocks of code using Fortran 77 syntax. Each block can be executed on a different processing node.

Blocks of code that can be executed independently are identified to PAMS with the `cpams parbegin` directive at the beginning of the blocks and the `cpams endpar` directive at the end of the blocks. Within this series of blocks, the `cpams parallel` directive identifies each independent block.

Basic Directives	/* pams pardo */	cpams pardo
	/* pams parbegin */	cpams parbegin
	/* pams parallel */	cpams parallel
	/* pams endpar */	cpams endpar
Extended Features	/* pams mergeby ... */	cpams mergeby ...
	/* pams prefetch ... */	cpams prefetch ...
Advanced Features	Store coherent memory support	
	Local system calls	
	GUI debug, monitoring, performance toolset	

Figure 1. PAMS directives and features

```
cpams parbegin
c The following block of code is executed on one processing node
cpams parallel
       statement or subroutine 1
       statement or subroutine 2
       ...
c The following block of code is executed on another processing node
cpams parallel
       statement or subroutine 100
       statement or subroutine 200
       ...
cpams endpar
```

Figure 2. Two parallel blocks of code using Fortran 77 syntax

Then, each block of code can execute on a different processing node (see Figure 2).

Figure 3 shows an example of parallel loops using Fortran 77 syntax. The loops contain independent iterations that can be executed on different processing nodes. It also supports nested parallel loops.

A loop with independent iterations can be executed in parallel by identifying it with the `cpams pardo` directive. The PAMS environment uses a sophisticated algorithm to distribute the iterations among the processing nodes in a load-balanced fashion, especially important when employing processing nodes of varying power. Nested loops can be parallelized as shown in Figure 3. In addition, I/O statements (read/write) can be included in parallel loops (as well as parallel blocks) to achieve a parallel I/O capability.

Extended Features

The extended features of PAMS consist of the `mergeby` and `prefetch` directives (see Figure 1). The `prefetch` directive is a fine-tuning optimization that tells the PAMS environment to begin acquiring the necessary memory locations prior to their actual reference. The `mergeby` directive defines the values for storage locations that are concurrently updated by different processing nodes. This is commonly known as a *global reduction operation*.

Dependency Violations

Three types of data dependency can inhibit a loop from being executed in parallel. The first is a *reduction violation*, which appears in the following form:

```
do i = 1, n
  sum = sum + ...
enddo
```

This global sum reduction operation example is typical of iterative solution methods. This type of loop cannot be parallelized (yet) since the iterations depend on each other. The `sum` memory location is repeatedly overwritten, based upon its previous value.

The USC and CSC memory models require a semaphore lock for updating `sum` with partial sums from each processor. However, the `mergeby` directive of the LC

```
cpams pardo
       do x = 1, n
          statement or subroutine 1
          statement or subroutine 2
          ...
cpams pardo
          do y = 1, m
             statement or subroutine 100
             statement or subroutine 200
             ...
           enddo
       enddo
```

Figure 3. Parallel loops using Fortran 77 syntax

memory model makes global reduction possible without any serialization. This directive can declare `sum` to be a merge variable, so that its memory location can be independently updated with different partial sums on each replicated memory copy of the processing nodes. At the end of the loop, the replicated memory regions are merged into a single valid image; that is, all of the partial sums are added during the merging process to produce the final result in the `sum` location. Any binary commutable operation including `sum`, `maximum`, and `minimum` can be used to achieve this.

The second type of data dependency called a *store-store* violation appears as follows:

```
do i = 1, n
  ...
  x = f(i) ...
enddo
y = x + ...
```

The scalar variable `x` is not a temporary loop variable, but one that is used after the last loop iteration. This loop must be modified in order to be parallelized because the memory location `x` must be updated in sequence to retain its last value. Declaring such a loop to be parallel is a common oversight, so PAMS can perform dynamic dependency analysis (for Fortran) on all loops to detect this situation at runtime. If it is detected PAMS logs an error, but the program will continue to execute the loop serially rather than abort or produce an incorrect result.

The last dependency violation type is the well-known *store-fetch* violation:

```
do i = 1, n
  f(i) = f(i - 1) + ...
enddo
```

This type of loop cannot be parallelized since each iteration is dependent upon its previous iteration. Unfortunately, the only choice is to find an alternative algorithm that does not employ store-fetch violations in the loops.

PAMS Example

Let us consider the simple example taken from a previous article in this series[4] concerning rank sorting a vector of numbers. Figure 4 shows the PAMS program in Fortran 77 that performs this function.

```
subroutine permute (key, rank, temp, n)
      integer n, key(n), rank(n), temp(n)
cpams pardo
      do i = 1, n
         temp(i) = key(rank(i))
      enddo
cpams pardo
      do i = 1, n
         key(i) = temp(i)
      enddo
```

Figure 4. Sorting a vector of numbers

The addition of two comment-based directives parallelizes this routine; therefore, this code still runs unchanged on a stand-alone workstation. Compare this to the High Performance Fortran (HPF) implementation[5] and notice that data distribution is not necessary when using virtual shared memory. The data movement among the processors is performed transparently during runtime by the PAMS environment. *Note:* If you have not implemented this routine using message-passing, now is a good time to do so in order to understand some differences among the three programming models.

Summary

Virtual shared memory provides parallel programmers with an environment that is conceptually similar to traditional uniprocessor development—it has a single, logical address space. Virtual shared memory can overcome the physical limitations associated with physically shared memory machines. Most physically shared memory machines cannot sustain more than 32 processors.

Virtual shared memory is intended for systems having many processors. PAMS, which is relatively new on the RS/6000 SP,

[4] Klepacki, David and Shen, Xianneng. "Data Parallel Programming with HPF on the RS/6000 SP." *AIXpert*, March 1998.
[5] Ibid.

has already demonstrated scalability up to 64 nodes. PAMS enables many applications that run on traditional symmetric multiprocessing (SMP) systems to be easily migrated to the RS/6000 SP. This exciting technology has great promise to simplify the programming process for machines that are ever increasing in size and complexity.

Other software vendors, such as Applied Parallel Research, Inc. (APR) are building tools to further enhance the usability of PAMS. APR now produces a tool that analyzes FORTRAN code for parallel opportunities and automatically inserts the appropriate PAMS directives into the source code.

Acknowledgements

The authors would like to thank Wayne Karpoff and Brian Lake of Myrias, Inc. for the many lucid discussions about virtual shared memory and PAMS, and for pioneering the next generation of parallel software development.

David Klepacki, IBM Corporation, 522 South Road, Poughkeepsie, NY 12601. Dr. Klepacki has been working in IBM's POWERparallel Systems Group since its beginning in 1991 as a computational physicist and scientific applications specialist with emphasis on performance benchmarking. Today, in addition to his technical endeavors, he also manages the parallel software tools segment for the technical marketing branch of the RS/6000 Division. Dr. Klepacki's current interests include performance programming, scalable parallel algorithms, scalable I/O, and portable high-performance computing tools. He holds a PhD in Theoretical Nuclear Physics from Purdue University as well as an MS in Electrical Engineering from Syracuse University.

Xianneng Shen, IBM Corporation, 522 South Road, Poughkeepsie, NY 12601. E-mail: xshen@us.ibm.com. Dr. Shen is a programming consultant 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 a PhD in Electrical Engineering from Syracuse University.

Porting Smart Cards to AIX

By George Kraft IV

Smart card standards have evolved as plastic card technology has begun using microprocessor silicon chips and programmable memory. This article presents an overview of smart card technology and applications.

Today, plastic cards used as "credit cards" have embossed customer name and account number, signature strip, magnetic strip, bar code strip, hologram, and in some cases, the customer's photograph. Smart card technology introduces microprocessor silicon chips and programmable memory to the already existing plastic card standards. Figure 1 shows a sample smart card.

Smart cards can electronically hold credit card account information, health insurance policies, driver's license, identification badge, or any other form of card information normally carried in a wallet. In addition, smart cards can function as an electronic purse to hold electronic cash. Electronic cash, or *e-cash*, can store an electronic representation of money that can be used for small miscellaneous expenditures such as vending machine purchases, bus rides, and metered parking.

Smart Card Chip

The smart card chip, about five millimeters in size, is embedded in the upper left corner of the plastic card. Outside the card, on the surface, is the contact plate for serial input and output (I/O) to the card reader. Cards without this contact are smart cards with an embedded antenna that usually operates within the one to fifteen centimeter range.

The chip generally has a central processing unit; EPROM, to hold its operating system; RAM; and EEPROM for resident smart card applications and data, shown in Figure 2. The job of the operating system is to receive commands from the 9600 baud serial connection, access the applications and data on the card, and send a response back to the serial connection. The smart card is a plastic "credit card" with an embedded miniature computer.

The ISO 7816 standard defines the physical characteristics of the smart card, including the I/O. This standard defines a consistent means for communicating with smart cards manufactured by different vendors. However, different cards and card manufacturers have distinguishing features.

The embedded operating systems represent the primary differences between smart cards. For this reason, smart-card developer toolkits are provided to create card-resident applications and data. This information usually "initializes" the cards for a particular application during manufacturing.[1]

George Kraft IV

[1] Visit *http://scs.bs.boeblingen.ibm.com/* to see IBM's smart card toolkit designed to support the IBM MultiFunction Card (MFC) and other card operating systems.

Figure 1. Physical characteristics of the smart card

OpenCard Framework

The OpenCard Framework (OCF) consortium held its inaugural meeting at the CardTech/SecureTech '98 trade show in Washington, D.C. during April 1998. At this meeting the consortium announced the release of the OCF Version 1.0 application programming interface (API) written in Java™. OCF was developed for host-based Java applications to be smart card aware and to be able to drive smart-card base applications. OCF provides smart-card reader device and smart-card operating system independence through Card Terminal and Card Services respectively.[2]

Host-resident, smart card-aware Java applications using OCF can be transported between host operating systems, smart-card terminal devices, and vendor-specific smart cards. To port OCF to particular platforms such as AIX®, a smart-card terminal device must be selected. Then a Java Native Interface (JNI) driver must be written to span between the host operating system and OCF running in the Java Virtual Machine (JVM). An example of this for AIX can be found in the *com/ibm/opencard/terminal/ibm5948/natives/aix* directory from the sample source downloaded from the OpenCard home page.

OCF provides a card extension service to discern smart card differences; for example, the Europay MasterCard Visa (EMV) e-cash services. OCF can call upon the registered EMV extensions for the smart-card electronic purse. OCF's auxiliary extendibility makes it flexible enough to accommodate most vendor smart cards.

Major PC and smart card companies formed the Personal Computer/Smart Card (PC/SC) workgroup in May 1996. The PC/SC specification, based on the ISO 7816 standards, is designed to interface smart card readers and cards with PCs through device-independent APIs for smart card-aware applications on Windows™ and Windows NT™.[3]

OCF-compliant applications written in Java migrate easily to Windows, because OCF can use the PC/SC drivers for smart card readers; therefore, OCF and PC/SC smart card applications coexist on the WinTel platform. OCF smart card applications running on various platforms can easily migrate to Windows; however, PC/SC applications cannot migrate away from Windows and Windows NT. See URL *http://www.opencard.org/misc/OCF-FAQ.html#PCSC* for more information regarding OCF and PC/SC compatibility.

The Java Card

We have talked about ISO 7816 physical characteristics, the OCF and PC/SC smart card-aware host applications, and various smart-card operating systems with supporting developer's toolkits to create card-resident, smart card applications.

[2] Visit *http://www.opencard.org/* for demos and more information regarding OCF.
[3] See URL *http://www.smartcardsys.com/* for more information regarding PC/SC.

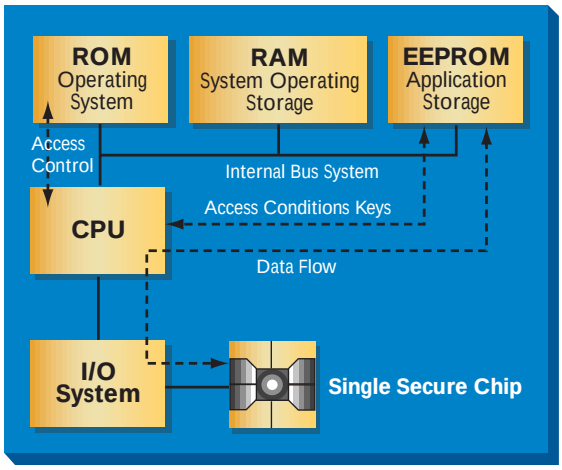


Figure 2. The smart card chip

The Java Card™, one such smart-card operating system, can run Java applications on the card.

The Java Card framework defines a reduced set of APIs for developing smart card-resident applications. Missing from the Java Card are dynamic class loading, security manager, threads and synchronization, object cloning, finalization, and large primitive datatypes (float, double, long, and char). However, the Java Card is not a stripped down Java Development Kit (JDK). Unlike the JVM on a PC, the Java Card Virtual Machine runs continuously. When the Java Card is removed from the reader and powered down, the information on the card is persistently stored in the card's EEPROM.[4] Developers must handle garbage collection manually, because the Java Card Virtual Machine does not.

Conclusion

This brief introduction to smart cards shows some advantages of programming smart card-aware applications and smart card-resident applications in Java using OCF and Java Card respectively. Although this article is not announcing any specific smart-card product releases for the AIX operating system on the RS/6000™, it suggests that a new "open" technology is ready to be utilized.

The OpenCard Framework has released the OCF Version 1.0 API for developers to create smart card-aware Java applications, and JavaSoft™ has released Java Card Version 2.0 API for developers to create smart card-resident Java applications.

George Kraft IV, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Kraft is an advisory software engineer for IBM's Network Computer Division. He recently moved from IBM's RS/6000 Division where he worked on the AIX integration of the IBM Network Station™. He has a BS in Computer Science and Mathematics from Purdue University.

[4] See URL *http://www.javasoft.com/javacard/* for more information regarding the Java Card API for smart cards.

This Ain't Your Grandpa's Client/Server!

COMMENTARY

We're talkin' client/server that sizzles! With high-performance SP servers driving clients that range from smart credit cards to Network Stations to even PCs or existing workstations—all tied together with common software written in Java™. This is e-business that *means* business! And we've got it all in this issue of *AIXpert*.

Start out with a discussion of how to develop client/server solutions using Java's RMI (remote method invocation) interfaces in a multithreaded environment. That article covers what you need to know to design and code an RMI application on AIX®, including a detailed example to help you understand the concepts. If that's not enough, continue on to the applet/servlet article. It guides you through the technology behind Java servlets—complete with a working example of a chat room client/server implementation written in Java.

Next, you need a client. We provide you with two possible clients, but you may supply your own. First is the IBM Network Station™. We explain everything you need to know about the Network Station's printing subsystem so that you can implement an effective business solution that depends on that subsystem. Or, perhaps you'd prefer a smart card client. We cover the important concepts of designing for smart cards with hopes that your application will be the one that makes this fledgling technology take off.

Finally, you need a server, and we've got that in SPades! Selecting what type of node to run on your RS/6000 SP server can be a daunting task. Read up on the choices of both Micro Channel® and PCI bus nodes described in our article and you'll have less trouble the next time you need to make that decision. And those who have been following it closely won't want to miss the fourth part of our parallel programming series—this issue covering the use of shared memory in the SP.

To round out the issue, we have an article describing a centralized software license management program, as well as the popular AIX question and answer section.

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

AIX Questions

Compiled by Jeff Simon

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.

What is the limit for the number of files that can be opened by a single process?

The limit is 2,000 file descriptors per process on operating systems prior to BOS 4.3. It is not possible to exceed this limit. `OPEN_MAX` (`/usr/include/sys/limits.h`) is the maximum number of file descriptors that can be used to create an array of file descriptors in the process user block. Again, this array size cannot be altered.

BOS 4.3.1 increased this limit to 32,767.

—Jeff Simon

What is AutoFS?

AutoFS, which is being shipped with BOS 4.3.1, includes a port of the Sun® ONC+ implementation of automatic mounting and replaces the current automount facility. AutoFS has the following three components:

- The `automount` command
- AutoFS filesystem kernel extension
- Automount daemon

AutoFS allows map files and other changes to be implemented without stopping and restarting the automount daemon.

—Jeff Simon

What is CacheFS in BOS 4.3?

Jeff Simon

The Cache File System (CacheFS) is a general-purpose filesystem caching mechanism available in AIX® 4.3 that improves NFS server performance and scalability by reducing server and network load. Designed as a layered filesystem, CacheFS provides the ability to cache one filesystem on another.

In an NFS environment, CacheFS increases the client-per-server ratio, reduces server and network loads, and improves performance for clients on slow links, such as point-to-point protocol (PPP). Any filesystem that is primarily read is a likely candidate for caching. In AIX 4.3, CacheFS only supports the caching of NFS Version 2 and Version 3 filesystems.

—Jeff Simon

I have installed BOS 4.3. When I try to run vmstat, I get the following error:

```
0551-129 knlist failed
```

This problem usually can be resolved by ensuring that the link `ls -l /unix` is set up. The link should look like this:

```
unix -> /usr/lib/boot/unix_mp
```

Set up the proper link if needed:

```
ln -s /usr/lib/boot/unix_mp
/unix
```

—Jeff Simon

What is the maximum number of continuation lines or size that the xlf <4/5> compiler can handle?

The maximum size of continuation lines for xlf <4/5> is 6700 bytes.

—Jeff Simon

Why are threads blocked inside the clnt_create() RPC call?

Generally, it is not considered safe to create threads in C++ code. For example, the use of I/O streams in the threaded C++ programs may cause problems. (`iostream.h` and `complex.h` are *not* threadsafe).

The `clnt_create()` remote procedure call (RPC) uses the `gethostbyname()` routine, which is not a thread-safe routine. This results in the `clnt_create()` call not being a true thread-safe function.

The `gethostbyname` returns a structure. However the `gethostbyname_r` function, which is thread safe, requires you to allocate and pass in the structure within your thread (plus a work structure). This makes the parameters different.

Routines that are not thread safe are subroutines that use static or global variables—routines returning a pointer to a static structure.

RPC calls on AIX are not currently thread safe. When writing RPC calls, developers must put locks around the RPC calls to ensure that only one thread comes in at a time to prevent deadlocking.

—David McCloud

Why do I have problems mounting filesystems exported from Sun servers?

IBM supports Sun NFS Version 4.03 and newer versions. On earlier versions of nfs, all NFS calls had to use port numbers that were reserved (below 1024). That is not necessary with the new versions (Version 4.2.1 and greater) unless an `nfso` option `nfs_use_reserved_ports` is set. The problem may be that the other clients are expecting the RISC client to use a reserved port.

This problem has two possible solutions:

- Turn off port checking on the PC server. This may not be allowed because of security considerations, but it could be tried if the client allows you to unset port checking.
- Use the command `nfso-o nfs_use_reserved_ports=1` to tell the RISC client to use reserved ports on its calls.

If you are using BOS 4.2.1, you must use APAR# IX71891 for this setting to work.

The `noacl` option is another NFS mount option to consider to disable the ACL RPC checks. Disabling the ACL queries can be done by mounting with this option `(mount -o intr,noacl server:/nfs_server_dir /nfsdir)`.

—David McCloud

David McCloud

How do you load a member of an archive using dlopen?

Figure 1 demonstrates how to load a member of an archive using `dlopen`.

```
compile: cc -g dltest.c -o dltest -ldl

/*———————dltest.c ———-*/
#include <stdio.h>
#include <dlfcn.h>
#include <errno.h>
int main()

{
 void *handle = dlopen("libXm.a(shr.o)",RTLD_MEMBER|RTLD_NOAUTODEFER);
/* you need to use the object module shr.o.
 use "dump -Hv /usr/lib/libXm.a" to find which module to use*/

 if (handle != 0)
 {
  void *addr = dlsym(handle,"XtToolkitInitialize");

  if (addr == (void *)0)
  {
   printf("PASS: dlopen() : ok\tHandle = %p\n",handle);
   if (errno == ENOEXEC)
    {
    printf("PASS: Find on XtToolKitInitialize ok, dlerror says:
[%s]\n",dlerror(
));
    }
 else
    perror("PASS: Find on XtToolKitInitialize ok");
  }

  else
  {
   printf("PASS: loaded libXm.a and found libXt.a\n");
   printf("INFO: %p=dlopen(), %p,=
dlsym(xx,XtToolkitInitialize)\n",handle,addr)
;
  }
  dlclose(handle);
 }
 else
 {
  if (errno == ENOEXEC)   {
   printf("Failed to load libXm.a - ok, dlerror says:
[%s]\n",dlerror());
   }
 else
   perror("Failed to load libXm.a - ok");
 }
}
```

Asthma Saudagar

Figure 1. Using dlopen to load a member of an archive

—Asthma Saudagar

What are the different logical volume types on AIX?

AIX has five logical volume types, shown in Figure 2.

Volume Type	Description
Filesystem	Holds filesystem data and metadata
Log	Holds journaled filesystem(JFS) metadata update log
Paging	Paging areas
Boot logical volume	Boot block and RAM filesystem code
Dump area	Holds panic dumps

Figure 2. AIX logical volume types

—Jeff Simon

My build line is very long and I get errors. Is there a workaround?

If the command line is too large for the linker to handle, it will give an error. To circumvent this, use the `-F` flag on the command line to insert a modified configuration file.

Copy the `/etc/xlC.cfg` to a local file `myconfig.cfg` and include the command-line options in the file under the appropriate stanza. InfoExplorer™ provides greater detail. Figure 3 shows an example.

```
xlC -Fmyconfig.cfg:stanza
-Fconfig_file:stanza | -Fconfig_file | -:stanza
```

Figure 3. Using the -F flag to insert a modified configuration file

—Asthma Saudagar

Compiled by Jeff Simon, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Simon has worked in AIX technical support since 1990 and currently works as a technical lead and system administrator for a series of technical Web sites. He has a BS in Computer Science from Southwest Texas State University and certification in AIX support.

Managing Software Licenses

By Jeff Simon

License Use Management (LUM) is an easy-to-use license management system that provides customers with a centralized method for managing software licenses.

License Use Management (LUM) is an easy-to-use license management system that provides customers with a centralized method for managing software licenses. The tools within LUM can manage access and usage of software products.

LUM enables large enterprises to collect use statistics for their software, monitor use levels, and provide overall management for their software usage. System administrators who manage systems in a heterogeneous network environment will appreciate its ability to run on multiple platforms (see Figure 1). In addition, software vendors that require their products to be license enabled will benefit. LUM provides the flexibility for try-and-buy or per-use licenses, or licenses with a specific time duration.

LUM consists of two products: License Use Management Application Developer's Kit and License Use Runtime. The Developer's Kit contains tools that allow vendors to code application programming interface (API) calls in their products and embed the code that services these calls. Vendors who purchase the Developer's Kit also receive a copy of the License Use Runtime, which contains the tools needed to manage the licenses. The vendors also get royalty-free rights to redistribute the License Use Runtime within the license-enabled application.

LUM, based on the LicensePower/iFOR™ technology, was developed at the Tivoli® Laboratory in Rome, Italy. This technology provides a client/server scheme for License Use Management. The client is the LUM-enabled product and the server is the license server.

License Use Runtime Platforms
AIX® Versions 4.1, 4.2, and 4.3
Windows NT™ Versions 3.51 and 4.0
Windows 95
OS/2 Warp™ Versions 3 and 4
Sun® Solaris® 2.5.1
HP/UX Versions 10.10 and 10.20
SGI (IRIX®) 6.2

Figure 1. Platforms that support License Use Runtime

The License Management System

The License Management System consists of a license server, runtime client code, and the associated license database. The server maintains the license database and a variety of information about license activity. The client is the LUM-enabled product, which has embedded calls that determine whether a license is available and what action the product must take based on information from the License Management System.

License Use Runtime monitors license agreements by tracking software usage. Figure 2 shows the contents of the runtime environment.

Jeff Simon

License Use Runtime Agreements	
bos.net (*)	Contains NCS 1.5.1 fileset (prerequisite)
ifor_ls.base	License Use Management Runtime base code
ifor_ls.libraries	License Use Management Runtime libraries (only needed if installing on AIX Versions 4.1 and 4.2)
ifor_ls.compat	Compatibility package (only needed if running products using nodelocked and/or concurrent nodelocked licenses, enabled with previous LUM versions)
ifor_ls.msg.*	Messages for the supported languages

Figure 2. Runtime environment

Fileset	Description
ifor_ls.ipf.*	IPF/X online documentation for the supported languages
ifor_ls.html.*	HTML documentation for supported languages
lumusg.ps	Using License Use Management Runtime Version 4 for AIX (PostScript®)
lumusg.pdf	Using License Use Management Runtime Version 4 for AIX (Adobe Acrobat®)

Figure 3. LUM filesets

The License Use Runtime software is available to download from the following ftp site: *ftp://ftp.software.ibm.com/software/lum*. Figure 3 shows filesets that are also available from this ftp site.

License Types

Software vendors can deliver to customers encrypted licenses that allow a product to be enabled for a nodelocked or network license. A nodelock license generally targets a specific node. It is typically used on a stand-alone system rather than in a client/server environment.

Nodelocked licenses can be either runtime based or non-runtime based. For the non-runtime based license, the product itself, not the License Use Runtime, manages the nodelocked license. The nodelocked license server manages the runtime-based enabled license. A product that is network license-enabled uses licenses that are stored on one or more network license servers. When a licensed program is started, License Use Runtime at the license server determines whether or not a license is available.

There are several different types of licenses available:

Nodelocked: Requires each workstation (the node) on which the license-enabled product runs to have license enrollment information (license password) locally installed

Simple nodelocked: Allows an unlimited number of simultaneous uses of the licensed application on the local machine

Use-once nodelocked: Permits a single use of a particular licensed product on a specific machine within the period for which the license is valid. Each time the product is started, one license is consumed

Concurrent nodelocked: Allows a limited number of simultaneous uses of the licensed application on the node where the application has been installed

Per server: Acts exactly like concurrent nodelocked licenses except that at any time it is possible to change into per-seat licenses

Trial period: Provides for an evaluation period that starts either when the product is enrolled or when the product is run for the first time

Network: Allows products to be stored on a network license server and shared among multiple network license clients

Steps to Reconfigure NetLS

1. `lssrc -g ncs  // Let's see which daemons are running`
   `// (llbd, glbd, nrglbd.)`
2. Stop the daemons:

```
stopsrc -s netlsd
stopsrc -s glbd
stopsrc -s llbd
```

3. Remove these files:

```
/tmp/llbdbase.dat
/usr/lib/netls/conf/cur_db      **!
/usr/lib/netls/conf/lic_db        !-- Make a backup copy of
/usr/lib/netls/conf/lic_db.bak  **!   these files, since they
                                      contain your password.
                                      Hence, you will need
                                      to restore them or
                                      get new passwords.

/etc/ncs/glb.e     ***!---these 2 are your 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.
```

   Note: All of the above files should normally be readable by others.

4. Run the following:

```
/etc/ncs/lb_admin
lb_admin: use global
lb_admin: clean
lb_admin: quit
```

5. Remove `netls_first_time`. Run `/usr/lib/netls/conf/netls_config`.
   (This is not necessary because `netls_config` re-creates `netls_first_time`.)

6. Select the default cell.

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

8. Verify that the system date is correct.

Figure 5. Reconfiguring NetLS

Reservable: Allows the license to be reserved for the exclusive use of a user, a group, or a node—generally for a specified time

Use-once: Allows the single use of a particular licensed product within the period for which the license is valid; each time the product is started, one license is consumed

Per seat: Allows the server of a licensed client/server application to request licenses on behalf of its clients without the need for the application clients to be license enabled

License Administration Tool

The Basic License Tool (BLT) enables you to view and update information about the licenses and obtain reports about their use. The BLT has both a graphical interface and a command-line interface. `i4blt` starts the License Use Runtime Basic License Tool interactive interface.

Potential Issues in Using LUM

Since LUM and Network Computing Services (NCS) are components of a distributed network system, location broker databases may become corrupt. A location broker provides information about the network or Internet resources to clients.

Resolving NetLS problems generally requires several approaches to analyze and isolate any problems. In some cases, trial and error may be the only way to determine the real problem. That could require rebooting license servers, reconfiguring NetLS, or perhaps even upgrading the software. In general, NetLS problems are caused by the replicated data files not being in sync, corrupted license-database files, or an expired license.

For corrupted files or data files out of sync, reconfiguring from scratch may be the best option. The concept of stopping daemons, flushing out corrupted database files, and restarting NetLS is the same concept used in other technologies, such as the Distributed Computing Environment (DCE). Figure 5 shows the steps to reconfigure NetLS.

The following are some general suggestions for addressing licensing problems:

- Know which levels of `*ifor*` that you are running.

```
lslpp -l bos.rte.ifor_ls
lslpp -l ifor_ls.base.cli
lslpp -l ifor_ls.base.gui
lslpp -l *ifor*
```

- Check the expiration date of your licenses (including the system date).
- Run `i4cfg-list` to determine which daemons are running.
- If you are using a nodelocked license and no daemons are running or something other than `i4llmd` is running, then enter `i4cfg-stop`, run `i4cfg-script`, and respond with the following: 2, 1, return, n, y, y. This should start the `i4llmd` daemon.
- `i4blt -r` ... can be used to generate reports to monitor compiler usage.
- If C Set++ 3.1.4 was previously installed on a system that coexists with another C or C++ compiler that does not work, it may be necessary to run `/usr/vac/bin/replaceCSET` to establish the symbolic links.

Known Fixes

LUM requires the following fixes:

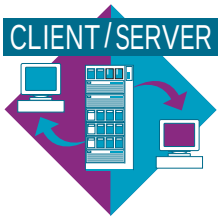
- APAR IX64408 for AIX Version 4.1
- APAR IX64105 for AIX Version 4.2
- APAR IX74933 for AIX Version 4.3

Conclusion

For more detailed information about the License Use Management, see the License Use Management white paper on the Web at *http://www.software.ibm.com/is/lum/*.

Jeff Simon, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Simon has worked in AIX technical support since 1990 and currently works as a technical lead and system administrator for a series of technical Web sites. He has a BS in Computer Science from Southwest Texas State University and certification in AIX support.

Multithreaded Programming with Java RMI



By Chary Tamirisa

This article presents an overview of the Java Remote Method Invocation (RMI) architecture, a description of its execution environment, the differences between synchronous and asynchronous method invocations, and two design choices for obtaining asynchronous behavior within the RMI framework using threads. Also included are synchronization issues specific to RMI along with a few tips. A detailed example illustrates the concepts discussed in this article. This article is based on Java RMI support in the JDK 1.1.4 running on AIX 4.3.

The Java™ Remote Method Invocation (RMI) provides a way to create distributed applications based on the client/server paradigm. RMI offers synchronous remote method invocation wherein the caller (client) waits until the remote method call returns from the server. RMI architecture is described on the JavaSoft™ Web page,[1] which provides a brief discussion of how remote method calls are mapped to the RMI threads. This article examines RMI's thread model in detail and outlines how to write asynchronous applications based on RMI.

Readers should be familiar with Java's thread support and the RMI model for distributed objects.

RMI Model

A *remote object* is one whose methods can be invoked from another Java Virtual Machine (JVM) running on the same machine or a remote machine. Typically, a server implements a remote object and a client invokes the methods in the remote object through a proxy. A proxy creates an environment in the client that allows the client to use the remote object as if it were local. Therefore, the client can invoke the methods transparently in the remote object.

The RMI system has three layers: the Stub/Skeleton, Remote Reference, and Transport. Together these three layers form the RMI runtime, which allows a client to invoke remote method calls as if they were local method calls. If a remote object becomes unreferenced, it becomes a candidate for the garbage collection.

Figure 1 shows the RMI architecture.

The Stub and Skeleton are Java classes automatically generated using the `rmic` command on the remote object implementation, or the server class. The Stub code runs in the client and the Skeleton code runs on the server. A key function of the Stub is to send the client's arguments to the server and obtain any return values. RMI uses the Java serialization of objects as the mechanism to send data from client to server and vice versa. Non-remote objects are passed by value whereas remote objects are passed by reference.

Chary Tamirisa

[1] RMI Architecture Specification. *http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.htm*

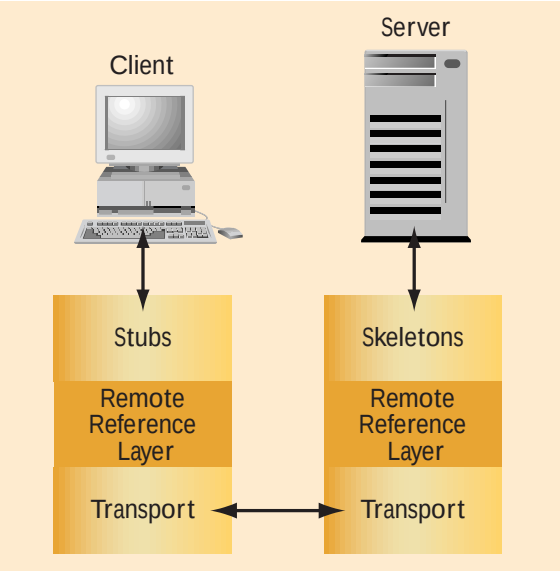


Figure 1. The architecture for the remote method invocation

The process of converting the arguments from their original form into the corresponding serialized form is called *marshalling*. The receiver converts (or deserializes) the serialized objects into the original form through a process called *unmarshalling*. Both the Stub and the Skeleton do marshalling and unmarshalling.

When a client makes a remote method call, the Stub initiates the call to the server. On the server side, the Skeleton makes the upcall to the actual implementation of the method and to marshal either the return value of the call or an exception. Every class that implements the `java.rmi.Remote` interface must have Stubs and Skeletons.

The Remote Reference layer enables the RMI architecture to be extensible in order to provide a variety of server and reference types. For example, RMI currently provides point-to-point (Unicast) server objects. However, in the future, it will allow multicast (one-to-many) reference types, which will allow replicated service.

The Transport layer listens to incoming requests, sets up connections for an incoming call, monitors the liveness of connections, and passes the connection to a dispatcher for the target remote call. Currently, the Transport layer implements the TCP protocol, although several other protocols can coexist with TCP.

RMI Execution Environment

To develop multithreaded applications based on RMI, it is important to understand the RMI execution environment. Figure 2 provides a snapshot of the threads in a Java RMI server application. Figure 3 provides a snapshot of the threads in a Java RMI client application. The snapshots are based on the Java Development Kit (JDK)1.1.4 running on AIX® 4.3.

Note: The actual implementation details may vary from platform to platform. Therefore, these threads cannot be used explicitly in your programs. Treat them as background threads created to facilitate RMI.

Server

The server side of RMI has three threadgroups: the system, the main, and the RMI runtime. A *threadgroup* defines a group of threads allowing operations such as `stop()` `suspend()`, `resume()` on all the threads in the group. Figure 2 shows the threadgroup, the thread name, its priority, and a notation of whether the thread is a daemon. A daemon thread is a background thread not expected to exit. When `main()` of the server application is invoked, two threadgroups already exist: system and main. The system threadgroup has one thread (Finalizer) and the main threadgroup consists of one thread (main). The Finalizer is a daemon thread but the main thread is not. The server application's `main()` is invoked in the main thread.

All RMI threads belong to the RMI runtime threadgroup. The RMI runtime starts when the server creates the remote object (the object whose methods can be invoked from another JVM). The RMI runtime threadgroup consists of the Pinger, Reaper, KeepAlive, LeaseRenewer, LeaseChecker, Cleaner, and a set of TCP Acceptor threads. From a programmer's viewpoint, the important set of threads are the TCP Acceptor threads. In the context of these threads, the remote methods are invoked when the Skeleton makes an upcall from the RMI's runtime to the actual method.

Client

The client-side RMI runtime does not contain all the threads that are found on

ThreadGroup	Thread Name	Priority	Is Daemon?
RMI Runtime	10 (MAX)		No
	Reaper	5	Yes
	KeepAlive	5	No
	Pinger	5	Yes
	LeaseRenewer	5	Yes
	LeaseChecker	5	Yes
	Cleaner	5	Yes
	TCP Accept-1	5	Yes
	TCP Accept-2	5	Yes
	TCP Accept-3	5	Yes
	TCP Accept-4	5	Yes
	TCP Accept-5	5	Yes
	TCP Accept-6	5	Yes
	TCP Accept-7	5	Yes
Main10		(max)	No
	Main	5	No
System		10 (max)	No
	Finalizer	1	Yes

Figure 2. Server-side RMI runtime threads

ThreadGroup	Thread Name	Priority	Is Daemon?
RMI Runtime		10 (max)	No
	Reaper	5	Yes
	Cleaner	5	Yes
	LeaseRenewer	5	Yes
Main		10 (max)	No
	Main	5	No
System		10 (max)	No
	Finalizer	1	Yes

Figure 3. RMI runtime threads in a simple client

the server side. A client that invokes just remote methods in a server has three threadgroups as shown in Figure 3. It is noteworthy that the client has no TCP Acceptor threads, because all remote methods invoked by the client are executed in the context of the current client thread. Since all remote method calls are synchronous, the current client thread blocks until the remote method call returns.

Mapping Method Calls to Threads

To use threads effectively, RMI programmers need to answer the following questions:

- Since a server consists of multiple RMI threads, how does RMI map incoming method calls to these threads?
- If the same client makes multiple calls to a server concurrently, will the server execute them concurrently or serially?
- What happens if the calls come from different clients (from different JVMs)?

RMI guarantees that calls originating from different client virtual machines will execute in different threads, which allows concurrency. However, calls originating from the same JVM are not guaranteed to

run concurrently; these calls may be invoked serially in some unspecified order.

Synchronous vs. Asynchronous Remote Methods

By definition, remote method invocations are synchronous: the client invokes a remote method and waits until the method is completed. Specifically, the calling thread waits until the remote method is completed and the results, if any, are passed back to the client.

The advantages of the synchronous call are that the client application does not have to do additional work to obtain the results of the method, and the remote call looks like a local call. If concurrent calls originating from the same JVM are mapped to different threads in the server's RMI runtime, the result is concurrency. However, RMI does not guarantee concurrency for multiple concurrent calls originating from the same JVM. If concurrent calls originating from the same JVM are mapped to the same RMI thread, the RMI remote method calls are executed serially in the server. But there is a way to handle this limitation.

An asynchronous remote method submits a request to a server and returns without waiting for the request to complete; the client can continue its execution. Java RMI does not support asynchronous semantics in its remote method calls. To get asynchronous behavior, it is necessary to create a multithreaded execution environment and invoke remote method implementations in the context of this multithreaded environment. That is, it is necessary to create a threaded execution environment and layer it over the RMI's runtime.

Asynchronous Application Design

In asynchronous application design, clients submit requests to a server in batch mode, but typically they do not wait for the remote methods to complete. The remote method invocations are asynchronous method calls. The clients may query the server to find status, or even cancel jobs as needed. The server typically notifies the appropriate client of the results after a job is completed.

The following scenario describes an asynchronous application. The client submits a remote request by invoking `submit()`. Once a job is submitted, the client can query the status of the job. When the job is completed, the server notifies the client.

Another scenario examines how a client can submit a request and cancel it later. The client creates a thread to invoke the remote `submit()` method and later creates another thread to invoke the remote `cancel()` method in the server. Figure 4 depicts this scenario.

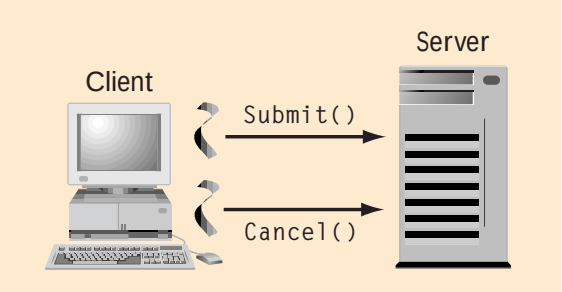


Figure 4. Two remote methods invoked by multithreaded client

On the server, let's assume that `submit()` method is running on thread TCP Accept-1 (RMI runtime thread). On which thread will `cancel()` run? If `cancel()` runs on another thread TCP Accept-2, it can interrupt thread TCP Accept-1 as shown in Figure 5. A thread can be interrupted or stopped by invoking the `interrupt()` or `stop()` method in the `java.lang.Thread` class.

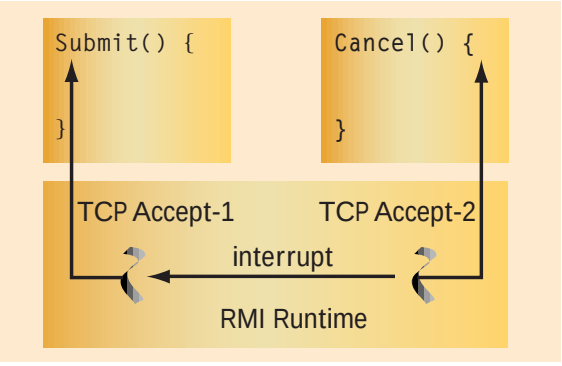


Figure 5. Server side-mapping each remote call from the same JVM to a different thread

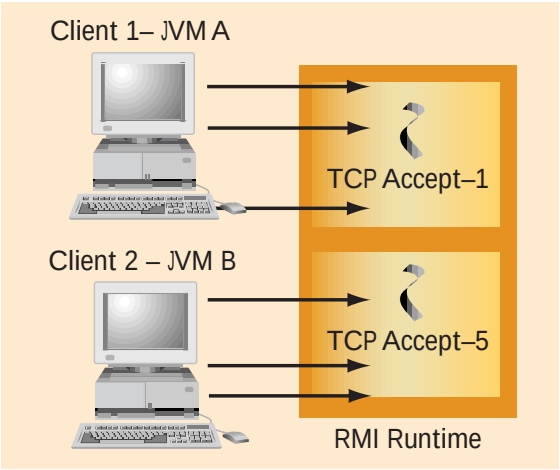


Figure 6. Server side—mapping all remote calls from the same JVM to the same TCP Accept thread

However, RMI does not guarantee mapping `cancel()` method (originating from the same client JVM) to another thread TCP Accept-2. It may map the `cancel()` request to thread TCP Accept-1, depicted in Figure 6.

It is clear that an infrastructure is needed on top of the RMI runtime to obtain asynchronous semantics. We need a new thread so that `submit()` runs on this thread rather than in the RMI runtime. Let's call this thread TS1, as shown in Figure 7. When `submit()` method is invoked on the server, the RMI runtime invokes `submit()` in one of the TCP Acceptor thread's context. From this thread, it is possible to create another thread (TS1) to distribute the actual work of the submit method `submitImpl()`. After starting this new thread, the RMI thread returns from the `submit()` method.

When `cancel()` is invoked on the server, the same or another RMI thread may invoke it. It does not matter which one. The `cancel()` method can determine which thread to cancel, cancel the thread, then return. Also, since `submit()` returns without waiting for completion, the client does not need a separate thread for `submit()` and `cancel()`. This enables asynchronous behavior from RMI.

This design is independent of the RMI implementation. A critical implementation requirement is the need for a unique identifier for each job (job ID) and a shared table (hashtable, for example) that uses this unique key to store all pertinent information associated with a job, including the thread ID. Since this table is shared across multiple clients, access to the table must be synchronized.

Callback Mechanism

The client has two choices in querying the status of the job it submitted. It can poll the server by periodically invoking the remote `status()` method, or alternately, the client can let the server invoke a client's method. The latter is known as a *callback*. To implement the callback mechanism, the client has two choices.

- Peer-to-Peer. In this model, the client and server are peers. The client also is a server and the server also is a client. Just like the server, the client exports its callback interface with a port number and hostname, then has the server look it up through the RMI Naming service. The client also extends `UnicastRemoteObject`. It is necessary to create the Stubs and Skeletons for the client side using `rmic` command.

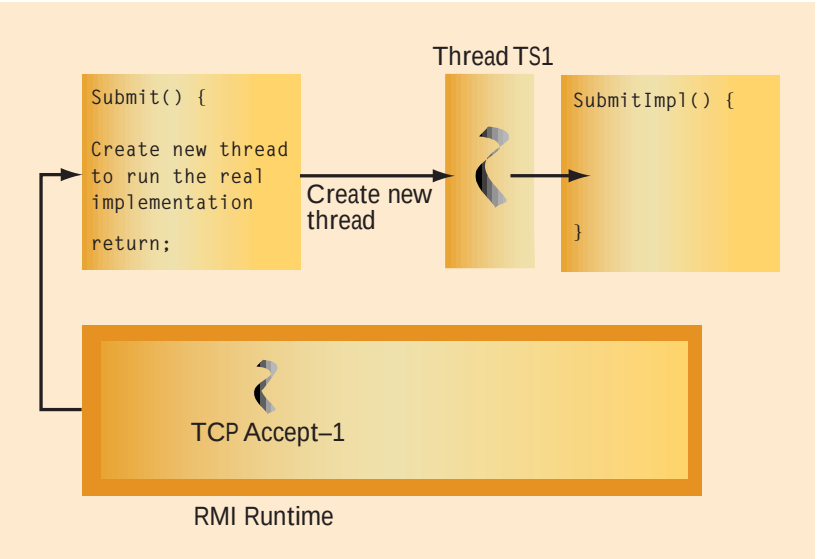


Figure 7. Application creating threads to obtain asynchronous behavior

- Callback Method. In this model, the client is not a peer because it does not export its interface to the RMI Naming and it does not extend `UnicastRemoteObject` (and hence, other clients cannot connect to it). The client implements the `java.rmi.Remote` interface and invokes `UnicastRemoteObject.exportObject()` to export the client object; it is not necessary to use the naming service. The client, of course, must implement the callback interface. The `rmic` command can generate the client-side Skeletons and Stubs in addition to the server-side counterparts. The client then can pass to the server a reference to itself, and the server can invoke the (callback) methods on the client.

The following describes how a client can export itself:

Callback Interface. The `callback` interface has one method `(jobDone())` which is invoked by the server when the job is completed. Note that the callback interface extends `java.rmi.Remote`, shown in Figure 8.

Client Class. The `PrintClient` class uses the RMI Naming service to look up the server. It then exports itself using the `exportObject()` method of `UnicastRemoteObject`, as shown in Figure 9.

Design Patterns

We have outlined two methods for obtaining asynchronous behavior. Let us apply these techniques to real applications. We will focus on the `callback` method of the client/server model, not the peer-to-peer model, because the Callback mechanism offers simplicity and retains the simpler client/server model. However, the following patterns can be used in the peer-to-peer model.

Two design patterns emerged from our experience. In the first pattern, remote methods took a long time for spawn threads to execute the actual call. In the second pattern, the remote method on the server placed the job in a queue, then woke up an executor thread to execute the job.

Spawn Threads from Remote Methods

This model assumes that the client needs asynchronous remote methods. Callbacks pass the results back to the client. Figure 10 captures this pattern.

To implement this pattern, follow these steps on the server:

- Identify the methods that need to run asynchronously.
- Create a separate thread (worker thread) each time these methods are invoked.
- Use `java.rmi.UID` class to create a unique identifier for the job.
- Record sufficient information (job ID) so the server can locate the client that originated the request. This information includes the client reference and the worker thread ID. The UID can be the key for a hashtable that stores this job ID.
- Once the job is completed, use the `callback` method of the client to send the results back to the client.

```
import java.rmi.*;
import java.rmi.server.*;

public interface Callback extends Remote {
            public void jobDone(UID id) throws RemoteException;
 }
```

Figure 8. Callback interface

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
import java.io.*;


public class PrintClient implements java.io.Serializable, Callback {

      PrintClient() {
         try {
             String url = "rmi://:2005/PRINTERS";
             Services server = (Services)Naming.lookup(url);
             UnicastRemoteObject.exportObject(this);
              ..
          }
          catch(Exception e) {}

          public void jobDone(UID id) {

      }
       /* Callback interface method */
      public void jobDone()throws RemoteException {

      }

}
```

Figure 9. Client class

The following applies to the client:

- Client implements a callback interface, such as `java.rmi.Remote interface`.
- Client exports itself using `java.rmi.server.UnicastObject.exportObject()`.
- Client submits a request to the server and records the UID returned by the server.
- Client can use the job UID to later query or cancel the job.

The advantage of this pattern is its simplicity; however, it also has a disadvantage. If the number of worker threads is directly proportional to the number of clients submitting jobs, then the number of threads created in a process places an upper limit on how many jobs can be supported simultaneously.

Queue Requests

Although this pattern is similar to the one above, it uses a queue instead of creating threads for each of the asynchronous methods, as shown in Figure 11. The remote method implementation (on the server) puts the job in a queue and wakes up a worker thread. As in the previous pattern, the remote method returns to the client with a unique identifier. When a client cancels a remote job using the UID, the server first tries to locate the job in the queue. If the server finds the job, it removes the job from the queue. If the job is not in the queue, the server then tries to see if the job is currently being executed by a worker thread, and if so, interrupts the thread,thereby, canceling the job.

If the server cannot find the job, it returns an appropriate status to the client. Similar to the previous pattern, the server uses a `callback` method to return results to the client.

We have outlined a single queue and one worker thread associated with it.

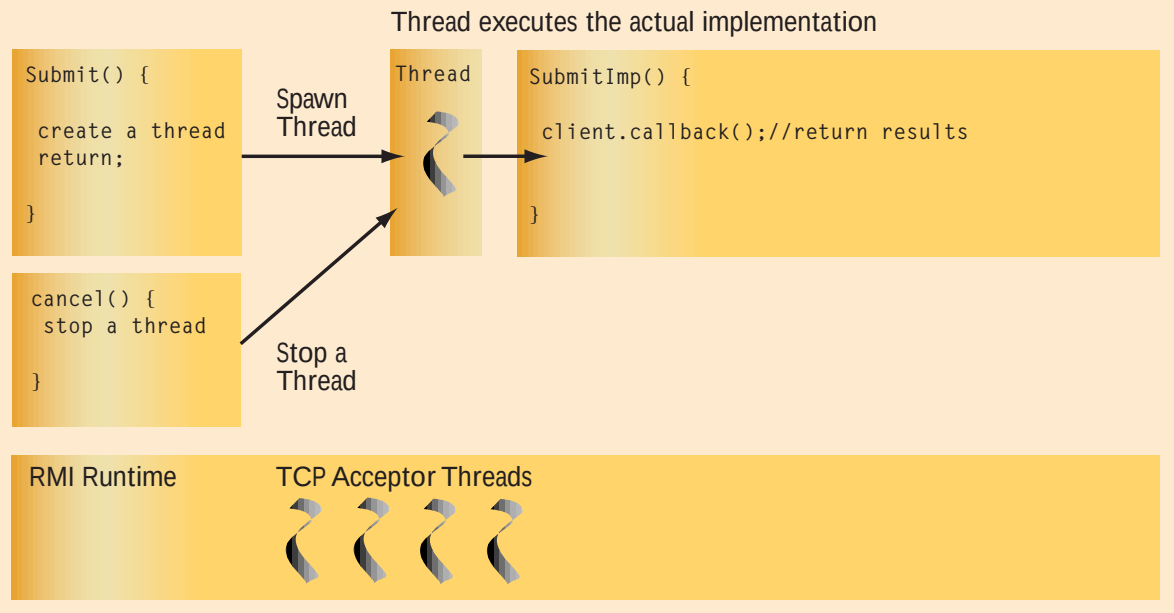


Figure 10. Design pattern: spawning a thread to obtain asynchronous behavior

Depending on the server needs, it is possible to have several queues. The worker threads can be dedicated to service a given queue, or shared by all queues from a general pool.

This pattern allows fewer worker threads to execute jobs coming from several clients. The server creates worker threads as needed. This pattern is advantageous when the worker thread has to invoke native methods, or use code that cannot be invoked from multiple threads concurrently. A single queue with a single worker thread allows serialized incoming requests.

Canceling Jobs

To cancel a job in process, the executing thread must be told to stop. Java provides two ways to do this: invoke the `interupt()` or `stop()` methods of the `java.lang.Thread` class.

Invoking `stop()` on a thread essentially terminates the thread. Therefore, within the queue model, a new thread must be started for other jobs in that particular queue to be executed.

Invoking `interrupt()` on a thread interrupts the thread if it is in a wait or sleep state and after the current job is deleted; the same thread can be used to execute the next job. However, if the thread is in a compute bound loop or blocked in an I/O, it may not be interrupted immediately. Hence, `interrupt()` may not be appropriate in some cases.

The Example section shows a sample application based on the queue design pattern. It illustrates the use of the `callback` method. In addition, it shows a simple way to persist (save) the client's job information so that a client can submit jobs to a server, exit the process, and later find the status of the jobs using the persisted file.

Synchronization Tips

Even if a server does not create any threads explicitly, the server runs in a multithreaded RMI environment. Typically, remote method invocations from different JVMs run concurrently on different threads in the server. When using synchronization in the server, be careful not to create deadlocks. For example, a deadlock arises when a thread T1 holds lock L1 and tries to acquire lock L2, while concurrently, thread T2 holding lock L2 tries to acquire lock L1. Both threads will become blocked forever because each thread needs a lock (resource) held by the other.

The deadlock situation especially can arise when using `callback` methods. Suppose a thread in the server holds a lock,

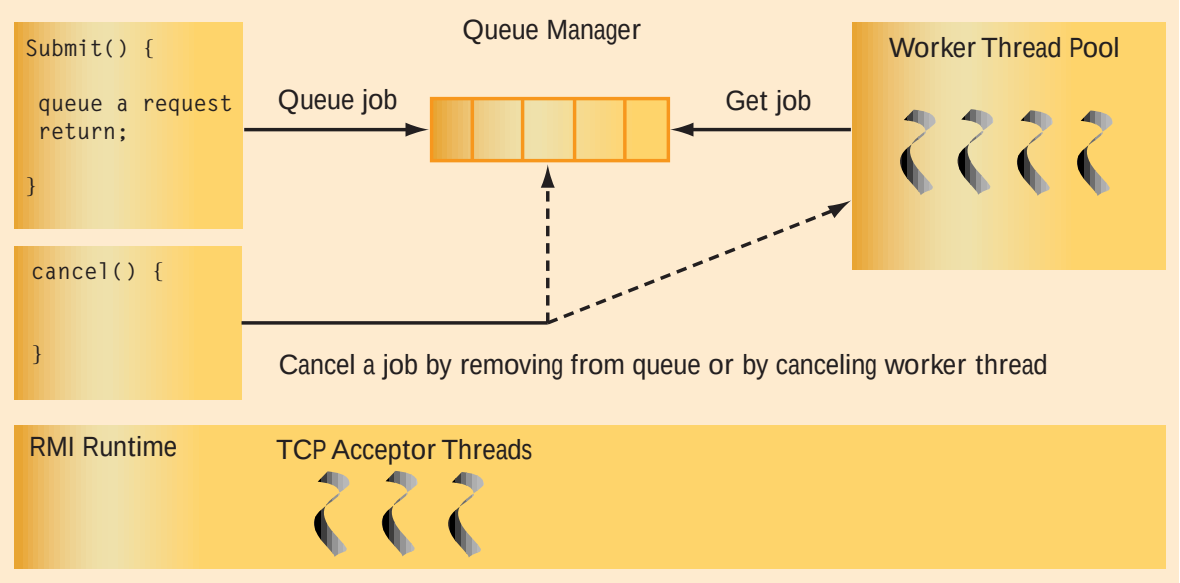


Figure 11. Design pattern: queue requests to worker threads

then invokes a client's `callback` method. If the `callback` method invokes a remote method on the server, the server can deadlock if contention exists for locks held by the calls. It is best not to hold locks while invoking `callback` methods.

Example

The following example was developed on AIX 4.3 using Java Development Kit 1.1.4 build a114-19980407.

The example illustrates the queue pattern to design an application. A `PrintServer` services requests for print jobs from clients. The `PrintClient` submits jobs to the server. The client does not wait for the job to complete, but persists its state (job information) and exits. At a later time, the client may cancel the jobs using the persisted information.

This example uses two interfaces: the Services and the `Callback` interfaces. The server implements the Services interface, which consists of three methods: `submit()`, `query()`, and `cancel()`. The `submit()` method executes the job asynchronously. It places a job request in a queue and returns to the client. `Query()` returns the current status of the job, and `cancel()` cancels the job either by removing it from the queue or by stopping the thread from executing the job.

This example uses one worker thread to take jobs off the queue and execute them. A pool of worker threads can be used instead of just one thread. The server creates the thread early in its execution. After a job is completed, the worker thread invokes a `callback` to the client with the execution results.

A `JobID` class saves the thread ID, the client object, and the status of the job. When a client invokes the `submit()` method, the server assigns a unique ID to the job (`java.rmi.server.UID()`). The server maintains a hashtable of these UIDs and the associated `JobIDs`, then places the job requests in a queue using the `put()` method of the `QueueManager` class. The `put()` method notifies the worker thread of work to do. The worker thread uses the `QueueManager's get()` method to get to the next executable job.

The `java.rmi.server.UID()` obtains a unique ID to tag a client's request. This UID is guaranteed to be unique across all clients. It is used as the hash key in a hashtable containing `JobID` objects.

The client (`PrintClient`) implements the `Callback` interface, which consists of the `jobDone()` method. It uses the `UnicastRemoteObject'sexportObject()` method to export itself so that the server can issue `callback` methods on it.

When the client invokes `submit()`, it receives a UID, uses this UID as a key, and saves the jobs it submits in a hashtable. The client invokes the `query()` and `cancel()` methods to get status and to cancel jobs respectively.

Since a user can submit a set of jobs using the client and also terminate the client, we introduce Java's persistence to save the client's hashtable. This allows users to cancel or query the job status at a later time. We use the `SaveJob` class for this purpose.

Note the three important methods in this class:

- `save()`: Serializes the object given to it and saves it in a specified file
- `restore()`: Takes a filename and returns the deserialized object
- `getNextRJEID()`: All file names have two parts: a simple integer value followed by the suffix .RJE (for example, `1.RJE` is a filename). To get the next ID, `getNextRJEID()` gets a list of all file names in the `/tmp/RJE` directory. It steps through all of the prefixes to determine the maximum integer value of the prefixes. It increments this maximum integer value by one to get the prefix for the next filename. It creates files such as `1.RJE`, `2.RJE`, and so on, in the `/tmp/RJE` directory.

After submitting a set of remote jobs, the client process can exit. A user can find status and cancel jobs, as needed, at a later time by invoking the `CancelJobs` class with an argument indicating the persisted filename (such as `1.RJE`).

Files

The files in Figure 11 are used in the example.

Compiling Files Here is how the files are compiled:

```
javac *.java
rmic PrintServer
rmic PrintClient
```

Running the Example Here is how to run the example:

```
rmi registry & (background)
java PrintServer (in a dedicated
  window)
```

You may run the client commands in separate windows: `java PrintClient`

Note the filename to use in canceling jobs: java CancelJobs 1.RJE.

The files that follow illustrate the queue pattern for designing an application.

Filename	Type	Description
`CallBack.java`	Interface	Used by server to call back client
`Job.java`	Class	Used by client to create a job
`JobID.java`	Class	Created by server to store the client object, the thread ID, the job status
`PrintClient.java`	Class	The client application
`PrintServer.java`	Class	The server application
`QueueManager.java`	Class	Manages the queue
`Services.java`	Interface	Server interface
`SaveJob.java`	Class	Used by client to save its job information
`CancelJobs.java`	Class	Client-side application to cancel jobs

Figure 11. Files used in example

Java Code

CallBack.java

```
import java.rmi.*;
import java.rmi.server.*;

public interface Callback extends Remote {
      public void jobDone(UID uid, String result) throws RemoteException;
}
```

Job.java

```
import java.io.*;

public class Job implements Serializable {
      int id;
      Job(int i) {
            id = i;
      }
      int getID() {
            return id;
      }
}
```

JobID.java

```
import java.util.* ;
import java.io.*;
import java.rmi.server.*;
import java.rmi.*;

public class JobID implements java.io.Serializable {

      UID uid;
      Job job;
      String status;
      transient Thread threadID;
      Callback client;

      JobID() {
            uid = null;
            job = null;
            threadID = null;
            status = "Not Initialized";
      }

      UID getUID() {
            return this.uid;
      }
      void setUID(UID uid) {
            this.uid = uid;
      }
      synchronized void setJob(Callback client, Job job ) {
```

JobID.java continued on following page

JobID.java

```
            uid = new java.rmi.server.UID();
            this.uid = uid;
            this.job = job;
            this.client = client;
      }

      synchronized void setStatus(String status) {
            this.status = status;
      }
      synchronized String getStatus() {
            return this.status ;
}
      synchronized void setThread( Thread thread) {
            this.threadID = thread;
      }
      synchronized Thread getThread() {
            return this.threadID ;
      }
      void callback() {
            String curStatus;
            synchronized(this) {
                 curStatus = new String(status);
            }

            try {
                  client.jobDone(uid, curStatus);
            }
            catch(RemoteException re) {
                  System.out.println("JobID: callback except=" + re);

            }
      }


}
```

PrintClient.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
import java.util.*;


public class PrintClient implements java.io.Serializable, Callback {


      Hashtable jobTable= new Hashtable();
      String s;
      String filename;

      public static void main(String[] args) {
            new PrintClient();

      }
```

PrintClient.java continued on following page

PrintClient.java

```
      void showStatus(UID uid, String s) {
            Job job;
            job = (Job)jobTable.get(uid);
            System.out.println("UID= " + uid.hashCode() + "    " +
"JobID="+job.getID()+ ": status= " + s);
      }

      PrintClient() {
            String url = "rmi://:8010/PRINTERS";
            Services server ;
            Job job ;
            UID jobID1, jobID2, jobID3;


            try {

            /* Connect to a Print Server */
            server = (Services)Naming.lookup(url);
            /* Export the client's reference */
            UnicastRemoteObject.exportObject(this);

            SaveJob saveJob = new SaveJob("/tmp/RJE");

            /* Create jobs and submit them */
            job = new Job(1);
            jobID1 = server.submit(this, job);
            jobTable.put(jobID1, job);
            filename =  saveJob.getNextRJEID();
            job = new Job(2);
            jobID2 = server.submit(this, job);
            jobTable.put(jobID2, job);
            job = new Job(3);
            jobID3 = server.submit(this, job);
            jobTable.put(jobID3, job);

            s = server.query(jobID1) ;
            showStatus(jobID1, s);
            s = server.query(jobID2) ;
            showStatus(jobID2, s);
            s = server.query(jobID3) ;
            showStatus(jobID3, s);

            saveJob.save(jobTable, filename);
            System.out.println("PrintClient saved file=" + filename);

            }
            catch(Exception  r) {
              System.out.println("PrintClient client got exception = " + r);
            }
      }

      public void jobDone(UID uid, String result){
            showStatus(uid, result);
      }
}
```

PrintServer.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.util.*;


public class PrintServer extends java.rmi.server.UnicastRemoteObject implements
java.io.Serializable, Services {

      public static void main(String[] args) {

            try {
            System.out.println("Welcome to Print Server ");
            PrintServer prs =new PrintServer();

            System.out.println("Locate registry");
            LocateRegistry.createRegistry(8010);
            String url = "rmi://:8010/PRINTERS";
            System.out.println("Bind to 8010");
            Naming.rebind(url, prs);
            }catch(Exception e) {
            System.out.println("exception got..");
            }
            System.out.println("Server in business..");
      }

      QueueManager qmanager;
      JobExecuter jobExecuter;
      Hashtable map = new Hashtable();


      PrintServer() throws RemoteException {
            qmanager = new QueueManager();
            jobExecuter= new JobExecuter (qmanager);
      }

      public UID submit(Callback client, Job job) throws RemoteException {
            JobID jobID = new JobID();
            jobID.setJob(client, job);
            jobID.setStatus("Waiting");
            map.put(jobID.getUID(), jobID);
            qmanager.put(jobID);
            /* Restart the executer thread if not alive*/
            if(!jobExecuter.isAlive())
                  jobExecuter = new JobExecuter(qmanager);

            return jobID.getUID();
      }
      public String query(UID uid) throws RemoteException {
            JobID jobID= (JobID)map.get(uid);
            if(jobID == null)
                  return "Job Not Found";
            else
            return jobID.getStatus();
      }

      public String cancel(UID uid) throws RemoteException {
```

PrintServer.java continued on following page

PrintServer.java

```
            JobID jobID = (JobID)map.get(uid);
            if(qmanager.cancel(jobID)) {
                  System.out.println("cancel waiting job: uid=" + uid);
                  jobID.setStatus("Canceled");
                  map.remove(uid);
                  return jobID.getStatus();
            }else if(jobID.getStatus() == "Executing"){
                  System.out.println("Cancel Executing Job: uid=" + uid);
                  jobID.getThread().stop();
                  if(!jobExecuter.isAlive()) {
                        System.out.println("Restart Executer thread");
                        jobExecuter = new JobExecuter(qmanager);
                  }
                  jobID.setStatus("Canceled");
                  map.remove(uid);
                  return jobID.getStatus();
            }else
                  return "Job Not Found!";
      }


      class JobExecuter extends Thread {
        QueueManager qmanager;
            JobExecuter(QueueManager qmgr) {
                  super();
                  this.qmanager = qmgr;
                  start();
            }

            public void run() {
                  System.out.println("running jobs..");
                  JobID jid = null;


                  for(;;)
                    {
                       synchronized(qmanager) {
                        try {
                          while( (jid = qmanager.get())== null)
                                qmanager.wait();

                        }
                        catch(InterruptedException e) {
                          System.out.println("Job Executer:interrupted=" + e);
                        }

                       }

                        jid.setThread(this);
                        jid.setStatus("Executing");

                       try{
                          // This must be replaced by the actual job */
                          sleep(2*60 * 1000); // Sleep for 2 minutes
                          }
                       catch(InterruptedException e) {
                          System.out.println("Job interrupted=" + e);
```

PrintServer.java continued on following page

PrintServer.java

```
                        jid.setStatus("Interrupted");
                        jid.callback();
                        continue;
                    }

                    // Now the job is done. So set status
                    jid.setStatus("JobDone");
                    // Let the client know
                    jid.callback();
                }
            }
        }

}
```

QueueManager.java

```
import java.util.*;
import java.io.*;

public class QueueManager implements java.io.Serializable {

    Vector jobs = new Vector();


    public synchronized void put(JobID id)  {
        jobs.addElement(id);
        notify();
    }

    public synchronized JobID get() {
        JobID id ;
        if(jobs.isEmpty()) return null;
        id = (JobID)jobs.firstElement();
        jobs.removeElement(id);
        return id;
    }
    private int findJobIDIndex(JobID id) {
        JobID jid;
        for(int i=0;i<jobs.size(); i++) {
            jid = (JobID)jobs.elementAt(i);
            if(jid.uid.equals(id.uid)) {
                return i;
            }
        }
        return -1;
    }
    public synchronized boolean cancel(JobID id) {
        JobID jid;
        int index = findJobIDIndex(id);
        if(index == -1) return false;
        jid = (JobID)jobs.elementAt(index);
        jobs.removeElement(jid);
        return true;

    }

}
```

Services.java

```
import java.rmi.*;
import java.rmi.server.*;

public interface Services extends Remote {
      public UID submit(Callback client, Job job) throws RemoteException;
      public String query(UID jobID) throws RemoteException;
      public String cancel(UID jobID ) throws RemoteException;
}
```

SaveJob.java

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.awt.*;

/**
Purpose: SaveJob allows persisting of jobs
*/

public class SaveJob
{


      public static String rootDir;
      int lastRJEID=0;    // last assigned persistent ID
      FileOutputStream f_out ;
      ObjectOutputStream out ;
      FileInputStream f_in ;
      ObjectInputStream in ;
      public  final String ioExc = " IO Exception:";
        public  final String misMatchExc = " Mismatch Exception:"
;
        public  final String classNotFoundExc = " Class Not Found Exception:";


      public SaveJob(String root)
      {
            this.rootDir = root;

      }
      public  void  save(Object obj, String fileName)
      {
            try
            {
                  f_out = new FileOutputStream(fileName);
            }
            catch(IOException e)
            {
                  System.out.println(ioExc + "FileName=" + fileName );
            }

            try
            {
                  out = new ObjectOutputStream(f_out);
```

Savejob.java continued on following page

SaveJob.java

```
            }
            catch(IOException e)
            {
                  System.out.println(ioExc + "FileName=" + fileName);
            }

            try {
                  out.writeObject(obj);
                  out.flush();
                  out.close();
            }
            catch(IOException e)
            {
                  System.out.println(ioExc + "FileName=" + fileName);
            }


      }

      public  Object restore(String fileName )
      {
Object obj=null;

            try
            {
                     String fullName= rootDir +  File.separatorChar+ fileName;
               f_in = new FileInputStream(fullName);
               in = new ObjectInputStream(f_in);
            }
            catch(IOException e)
            {
               System.out.println(ioExc + "FileName=" + fileName);
               return null;
            }
            try {
                obj = in.readObject();
                in.close();
            }
            catch(ClassNotFoundException el)
            {
               System.out.println(classNotFoundExc + "FileName=" + fileName);
            }

            catch(IOException e)
            {
                  System.out.println(ioExc + "FileName=" + fileName);
            }

            return obj;

      }
      public String getNextRJEID()
      {
                  if ( lastRJEID == 0 ) {
                  File file      = new File( rootDir );
                  String files[] = file.list();
                  if ( files != null ) {
```

Savejob.java continued on following page

SaveJob.java

```
                                for ( int i = 0; i < files.length; i++ ) {
                                String curFile = files[i];
                                int len     = curFile.length();
                                if(((curFile.substring( len - 4, len )).
toUpperCase()). compareTo( ".RJE" ) == 0 )
                                                lastRJEID = Math.max( lastRJEID,
                                                Integer.parseInt(
curFile.substring( 0, len - 4) ) );
                                        }
                        }
                        }
                        return rootDir +  File.separatorChar+(++lastRJEID) + ".RJE";
        }

}
```

CancelJobs.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
import java.util.*;


public class CancelJobs implements java.io.Serializable {


        Hashtable jobTable;
        String s;


        public static void main(String[] args) {
                new CancelJobs(args);

        }

        void showStatus(UID uid, String s) {
                Job job;
                job = (Job)jobTable.get(uid);
                System.out.println("UID= " + uid.hashCode() + "    " +
"JobID="+job.getID()+ ": status= " + s);
        }

        CancelJobs(String[] args) {
                String url = "rmi://:8010/PRINTERS";
                Services server ;
                Job job ;

                try {

                /* Connect to a Print Server */
                server = (Services)Naming.lookup(url);
                /* Export the client's reference */

                SaveJob saveJob = new SaveJob("/tmp/RJE");
```

CancelJobs.java continued on following page

CancelJobs.java

```
            System.out.println("Restore file=" + args[0] );
            jobTable = (Hashtable)saveJob.restore(args[0]);

            int size = jobTable.size();
            System.out.println("size=" + size);

            Enumeration keysEnum = jobTable.keys();
            UID uid;
            do{
                  uid = (UID)keysEnum.nextElement();
                  s = server.query(uid) ;
                  showStatus(uid, s);
                  s = server.cancel(uid);
                  showStatus(uid, s);
                  s = server.query(uid) ;
                  showStatus(uid, s);

            }while(keysEnum.hasMoreElements());


            }
            catch(Exception  r) {
              System.out.println("PrintClient client got exception = " + r);
            }
      }

}
```

Conclusion

The description of the RMI runtime, RMI's execution environment, synchronous and asynchronous method calls, and synchronization tips should provide enough knowledge to develop applications that require asynchronous behavior.

Reference

Downing, Troy Bryan. *Java RMI: Remote Method Invocation*. IDG Books Worldwide Inc. (*http://www.idgbooks.com*).

Chary Tamirisa, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Internet: chary@austin.ibm.com.

Mr. Tamirisa is currently the manager of the Performance and System Test group in the Global Business Intelligence Solutions (Data Mining) area. He has extensive experience in developing Java applications and the JavaBean component model. Mr. Tamirisa worked on the DCE threads package on AIX and OS/2®. 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 in Montreal, Quebec, Canada, and a BTech in Electrical Engineering from the Indian Institute of Technology in Madras, India.