Exploiting Java™ Technology with the SAS® Software

Barbara Walters, SAS Institute Inc., Cary, NC

Abstract

This paper describes how to use Java™ technology with SAS software. SAS Institute provides a beta SAS/SHARE*NET driver for JDBC™. In addition, we provide classes that allow Java applications and applets to create a remote SAS session, submit SAS statements, and retrieve results generated by the submitted statements. This paper describes how to use these class libraries and address client/server configuration and performance issues.

Introduction

Since its introduction in mid-1995, Java and applets have become an integral part of the World Wide Web. Java is a rich programming language that enables Web programmers to create sophisticated and responsive client/server applications. Because Java is portable and secure, users of Web applications can be confident that those applications will execute properly and not corrupt their computers.

SAS Institute provides a SAS/SHARE*NET driver for JDBC, a driver for Remote Computing Services, and several drivers to access both SAS data and SAS computing facilities. By using these drivers, SAS programmers can make SAS resources available to a much wider user community via the Web.

Overview of Java

Java is an object-oriented programming language developed by Sun Microsystems, Inc. Java source code is compiled to a byte stream, which is merely a long sequence of bytes. The byte stream is interpreted by the Java Virtual Machine (JVM). Since Java is interpreted, the code will run on any platform that has an implementation of the JVM. This enables Web authors to embed executable content, i.e., small programs called applets, within their HTML documents and be confident that the applets will execute properly on the receiving machine. Web browsers, such as Netscape Navigator and Microsoft's Internet Explorer, include a version of the JVM as part of the browser.

Java provides libraries of classes that offer a rich set of functionality. These include a library for graphical user interface (GUI) components called AWT (Abstract Window Toolkit), an I/O library, and a network access library. Java Developer's Kit (JDK) Release 1.1 provides a library for the JDBC driver manager. These libraries are standard and are included with the JVM.

Java programs are often delivered via the Internet. In order to protect the local machine from malicious programs, the Java language and the JVM provide a secure environment for applet execution. The secure environment ensures that the client machine (the machine where the browser is running) is not corrupted by the downloaded program and that no information is stolen from the client machine.

Java security is based upon the "sandbox" model. This means that the downloaded Java code is restricted from accessing resources outside of its sandbox. The sandbox is the set of resources the code is allowed to access. Examples of resources are threads, socket connections, and local files. Each application must define the limits of the sandbox.

Security is enforced by the Security Manager. The Security Manager is responsible for enforcing the limits of the sandbox. Each Java application has a Security Manager associated with it. The application can tailor the Security Manager, so each application may, and in fact does, allow different limits for the sandbox. For applets, the Web browser is the application that owns the Security Manager. Each browser may put different restrictions on applet behavior. The default limits imposed by the Security Manager are: classes may not access the local file system, read environment variables, execute other programs on the local machine, or make socket connections to machines other than the machine from which they were downloaded.

The JVM provided with Web browsers supports the Java "core" libraries. These libraries contain classes that are standard with all JVMs and may not be downloaded over the network. The JDBC driver manager is an example of a core library. It must be available on the local machine; it is ineligible for download.

All Java classes are included in a "package." The package is the hierarchical name associated with the class. Any class whose package name starts with `java` is considered a core class. The package name for the JDBC driver manager is `java.sql` and is therefore one of the core libraries.

As Java code starts executing, Java classes that are not available on the local machine and are not core classes, are dynamically downloaded from an HTTP server. Classes are loaded as they are needed, so the number of classes downloaded from the network can be kept to a minimum. Downloaded classes are subject to the restrictions imposed by the Security Manager. Classes available from the local machine are not usually subject to the same restrictions.

Since Java is portable, secure, and dynamically loaded, it is an ideal language for writing Web-enabled applications. Applications written in Java that are accessed via an HTML page are called "applets."

The following is an example of an applet tag in an HTML document that loads a Java class called `testapplet.class`.

```
<applet code="testapplet.class" width=600
height=425>
```

Figure 1: Sample applet tag

When the browser encounters the applet tag, it invokes the JVM and passes it the name of the applet class.

If the applet class itself was downloaded from the HTTP server, the Security Manager restricts the resources for all classes executed as part of the applet. If the applet class was available on the local machine, the restrictions may be relaxed.

With JDK 1.1, Java classes may contain digital signatures. Applet classes that are downloaded from the HTTP server and are not digitally signed are considered "untrusted." JDK 1.1 provides a Security API that allows special privileges for digitally signed applets, which are "trusted applets." Unfortunately, JDK 1.1 beta was released in mid-December 1996 and cannot be addressed in this paper.

SAS/SHARE*NET Driver for JDBC™

JDBC is the Java Database Connectivity API created by JavaSoft. JDBC is similar to ODBC, but for the Java environment. Users do not need JDBC drivers installed on their machines. Drivers are dynamically downloaded from an HTTP server as needed.

The JDBC API provides an SQL interface to databases. By using JDBC, applications and applets can establish a connection to the database server, submit SQL statements, and obtain result sets. Applications can obtain information about the database (database `metadata`) and a particular result set (result set `metadata`).

Database metadata contains information about the characteristics of the database and what data it contains. For example, through database metadata, a program can determine which numeric or string operations can be used in an SQL statement. The program can obtain a list of valid data types supported by the database. It may ask the names of tables available at a particular database server, the names of the columns within those tables, and the type of data contained in each column.

Although drivers themselves can be downloaded over the network, the JDBC driver manager must be installed on the local machine. The JDBC driver manager is included with JVMs based on JDK 1.1. JavaSoft maintains information about JDBC and has the driver manager available at its Web site. The SAS/SHARE*NET driver available for download from SAS Institute's Web site conforms to the JDBC 1.1 API specification from JavaSoft. (For the URLs, see the References at the end of this document.)

*Using the SAS/SHARE*NET Driver for JDBC*

Databases are accessed through a unique style of URL that is defined for JDBC. Each database protocol has a unique URL registered with JavaSoft for use with JDBC. The format of the URL for SAS/SHARE*NET driver is

`jdbc:sharenet://host:port.`

The name `host` indicates the TCP/IP address of the machine where the SAS/SHARE*NET server is running and `port` references the port number assigned to the server.

Code example:

```
import java.sql.Connection;
import java.sql.Driver;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

try {
 Driver driver = (Driver)Class.forName
  ("COM.sas.net.sharenet.ShareNetDriver")
  .newInstance();
 Connection connection = driver.connect
  ("jdbc:sharenet://sharenetserver:portnumber",
   properties);

 Statement statement =
  connection.createStatement();

 ResultSet resultset = statement.executeQuery
  ("select columnone from my.table");

 while (resultset.next() == true)
  String columnOne = resultset.getString(1);
  System.out.println("Results: " + columnOne);

} catch (SQLException e) {
 System.out.println("Exception thrown: "
  + e.getMessage();
}
```

Figure 2: Code to Create Objects

The code in Figure 2 creates all of the objects required to submit an SQL query and retrieve results. This code segment is simplified and will not compile properly as is.

In this code segment, the SAS/SHARE*NET server is running on the machine named `sharenetserver` and `portnumber` is the number of the port assigned to the SAS/SHARE*NET server. The SQL statement requests all of the rows for the column named `columnone` in the table named `my.table`.

All of the classes for the SAS/SHARE*NET driver are contained in the package named `COM.sas.net.sharenet`. The driver is written entirely in Java and may be dynamically downloaded from the HTTP server. Please keep in mind that the JDBC driver manager must be available on the client machine. It may not be dynamically downloaded.

The name of the database metadata class is `java.sql.DatabaseMetaData`. This class supports the following methods: `getCatalogs`, `getSchemas`, and `getTables`. "Schemas" are analogous to SAS libraries; "tables" are analogous to SAS data sets. There is no analogy for a JDBC catalog in SAS software. JDBC catalogs are not the same as SAS catalogs.

The SAS/SHARE*NET driver cannot provide metadata information about a foreign database such as Oracle. It can only provide this type of information about the SAS databases.

The `DatabaseMetaData` object can be created after a connection has been established to the database (i.e., a `java.sql.Connection` object has been created).

The result set metadata class `java.sql.ResultSetMetaData` provides methods for obtaining number of columns, column names, labels, and other characteristics of a particular result set. The `ResultSetMetaData` object can be created from any `ResultSet` object.

The diagram in Figure 3 depicts a typical configuration for an untrusted applet accessing a remote SAS/SHARE*NET server.

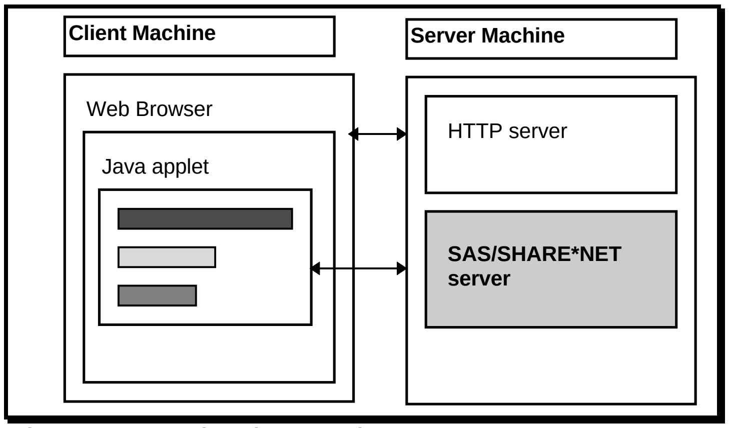

Figure 3: Untrusted Applet Accessing Remote Server

The Web browser on the client machine requests an HTML document from the HTTP server. The document arrives at the client machine and the browser detects the applet tag in the HTML document. The browser invokes the JVM and passes the name of the applet class to the JVM. The applet class is downloaded from the HTTP server and starts executing.

The Java class loader requests the SAS/SHARE*NET driver class file from the HTTP server. The driver class creates a `java.sql.Connection` object that establishes a socket connection with the SAS/SHARE*NET server. It is not a requirement that JDBC drivers use a socket interface, but that is the default for our driver.

The Connection object sends the SQL queries to the SAS/SHARE*NET server and the server sends its response via the socket interface. The Connection object communicates directly with the SAS/SHARE*NET server.

It is very important to note that HTTP server and SAS/SHARE*NET server are running on the same physical machine. If the applet is untrusted and the SAS/SHARE*NET server is not located on the same machine as the HTTP server, the socket connection is disallowed and a

Security Exception is thrown. This is a restriction imposed by the default Java Security Manager, not a restriction imposed by our driver.

The SAS/SHARE*NET driver for JDBC also supports access to foreign databases such as Oracle. By setting properties for the Connection object, a programmer can specify another database. The properties named `dbms` and `dbms_options` allow the programmer to specify the database and the options needed for connecting to that database.

Java Classes for Remote SAS Compute Services

SAS Institute plans to provide Java classes that allow an applet to create a remote SAS session, submit SAS statements, and retrieve results. The classes also provide support for accessing SAS data sets and downloading files. These classes provide a subset of the functionality of a SAS/CONNECT™ client. The plans do not include support for a remote graphics driver. Sample applets using the classes that provide access to a remote SAS session will be available from the SAS Institute Web site. (See the References section for the site URL.)

Several different mechanisms are available to start a remote SAS session. We provide a CGI program to start SAS on a remote system and we provide a Java class that starts a SAS session using telnet. The Java class that starts the SAS session is named `TelnetConnectClient`. The code segment in Figure 4 shows how to create a SAS session using the `TelnetConnectClient` class. This code is still under development, so the class names and methods are subject to change. This example starts the remote SAS session on the machine named `myhost` and retrieves the log lines generated by SAS software initialization.

Code example:

```
import COM.sas.net.connect.TelnetConnectClient;

String host = new String(“myhost”);
String loginprompt = new String(“logon:”);
String passwordprompt = new String(“Password:”);
String portprompt = new String(“PORT=”);
String cmdprompt = new String(“myhost>”);
String sascmd = new String(“sas -dmr”);
String userid = new String(“myid”);
String password = new String(“password”);

try {
 TelnetConnectClient tconnection =
  new TelnetConnectClient();
  tconnection.connect(
   host,                /* host            */
   loginprompt,         /* login prompt    */
   userid,              /* login id        */
   passwordprompt,      /* password prompt */
   password,            /* password        */
   cmdprompt,           /* command prompt  */
   sascmd = sascmd      /* command to enter */
   portprompt,          /* port prompt     */
);

 String lines = tconnection.getLogLines();
 System.out.println(lines);

} catch (Exception e) {
 System.out.println(e.getMessage());
}
```

Figure 4: Code to Start SAS Session with TelnetConnectClient

The `TelnetConnectClient` communicates with either the Telnet daemon or the SAS/CONNECT spawner program on the remote host. The `TelnetConnectClient` uses the variables passed to the connect method to log in a user identified by `userid` and start a SAS session using the `sascmd` variable. The documentation provided with the package will describe how each of the parameters is used.

The code segment in Figure 5 shows how to submit statements that compute the mean value from column `abc` in data set `xyz` and create a new data set called `results`, obtain the list output, and print the list output to standard out.

Code Example:

```
String lines = new String
 (“proc means data=xyz mean;
   var abc;
   output out=results;
   run;”);

tconnection.rsubmit(lines);

String logLines = tconnection.getLogLines();
String listLines = tconnection.getListLines();
System.out.println(listLines);
```

Figure 5: Code to Create Data Set and Print Results

Access to SAS data is provided through a Single User server. A method `getSingleUserServer` starts the server in the remote session. The Single User server is made available as a JDBC Connection object. The code segment in Figure 6 shows how to start the Single User server and retrieve the contents of the data set `WORK.A`. This example uses the `TelnetConnectClient` object named `tconnection` to start the Single User server and return the `java.sql.Connection` object. The `java.sql.Connection` object is used to retrieve the contents of the data set named `WORK.RESULTS` generated in the previous example.

Code Example:

```
java.sql.Connection suServer =
    tconnection.getSingleUserServer();
Statement statement =
    suServer.createStatement();
ResultSet resultset =
    statement.executeQuery
       (“select x from work.results”);
    while (resultset.next() == true)
       String x = resultset.getString(1);
```

Figure 6: Code to Start Server Using TelnetConnectClient

The classes that provide remote computing services communicate with a remote SAS session in a very similar manner to the SAS/SHARE*NET driver for JDBC. By starting the remote SAS session with the -DMR option, the remote SAS session is listening on a particular port. The Java classes establish a socket connection to that port and exchange messages with the remote SAS session.

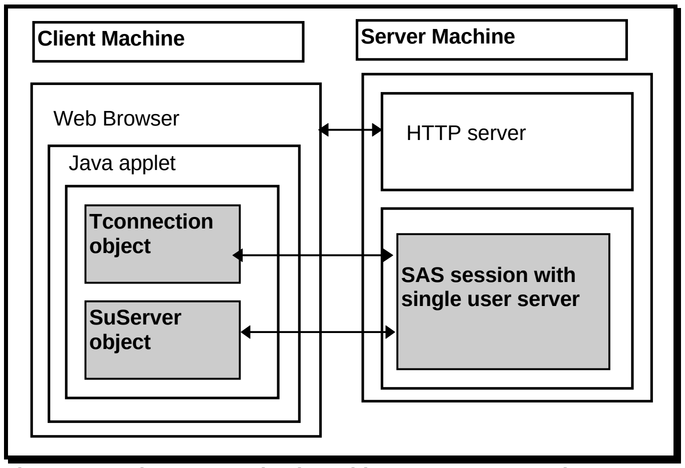

Figure 7: Applet Communicating with Remote SAS Session

Figure 7 depicts an applet communicating with a remote SAS session. Since applet classes are assumed to be downloaded from the HTTP server, the security restrictions apply and the remote SAS session must run on the same machine as the HTTP server.

HTTP Tunneling

We find the requirement of having an HTTP server available on each machine that has a SAS/SHARE*NET server or a remote SAS session to be quite restrictive. In addition, many Internet users are behind firewalls that will not allow socket connections to machines outside of the firewall. We provide a solution to both of these problems through “HTTP tunneling.”

“Tunneling” refers to the practice of encapsulating a communication protocol within another protocol. By using a combination of CGI programs and Java classes that provide HTTP protocol support, we are able to encapsulate the proprietary protocol used by the SAS/SHARE*NET driver and remote SAS session in HTTP requests and HTTP responses. In other words, the proprietary SAS protocol is wrapped in HTTP.

Many firewalls allow HTTP protocol to pass through. Since the SAS protocol can be wrapped in HTTP, many users that previously could not access remote SAS services can if the applet makes use of HTTP tunneling.

Figure 8 depicts the components needed for HTTP tunneling.

Because the SAS Protocol Interpreter is maintaining the socket connections, the restriction of having the remote SAS session created on the same machine as the HTTP server is removed. The applet is sending requests to the SAS Message Router, not directly to the remote SAS session. The remote SAS session can be created on any machine with which the SAS Protocol Interpreter can communicate.

To some extent HTTP tunneling is circumventing the Java Security Manager. We are allowing applets to communicate with a machine other than the machine from which the applet originated. The intent of the Security Manager was to disallow socket connections to machines within a firewall, if the applet was downloaded from outside a firewall. The SAS Protocol Interpreter allows the administrator of the HTTP server to restrict which machines are eligible for connections on behalf of an applet.

Because the applets are using HTTP protocol to communicate rather than direct socket connections, SAS services are available to users behind firewalls that allow HTTP communication.

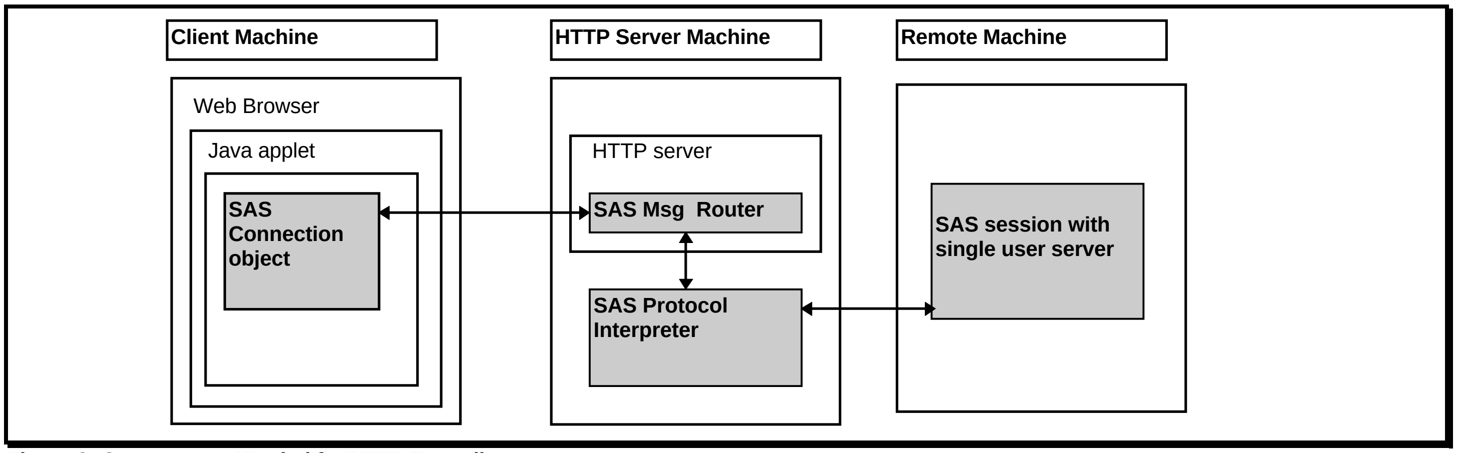

Figure 8: Components Needed for HTTP Tunneling

The SAS connection object, either a TelnetConnectClient or JDBC Connection object, does not create a socket connection to the remote SAS session. Instead, it sends HTTP requests to a CGI program installed on the HTTP server machine. We refer to this program as the SAS Message Router. The SAS Message Router passes the HTTP request to the SAS Protocol Interpreter. If the SAS Protocol Interpreter is not currently running, the SAS Message Router starts it. The SAS Message Router is a CGI program that is run each time a request is sent from the SAS connection object. It is not a persistent process. The SAS Protocol Interpreter is a persistent process that maintains connections to the remote SAS sessions.

The SAS Protocol interpreter opens and maintains a socket connection to the remote SAS session. It sends the messages received in the HTTP requests to the appropriate SAS session. The response from the remote SAS session is received by the SAS Protocol Interpreter from the socket connection. The SAS Protocol Interpreter wraps the SAS response in an HTTP response that is sent back to the SAS connection object.

The SAS connection object supports a property that specifies the path and name of the SAS Message Router. If this property is set, the connection object will emit HTTP requests rather than use a socket connection.

The configuration program associated with the SAS Protocol Interpreter allows an administrator to tailor its capabilities. The configuration program may include a list of machines that are allowed to provide SAS services to Java applets or a list of machines that are not allowed to provide services to applets. The administrator can configure the interpreter to only allow certain commands to be used to start a SAS session and disallow all others. The configuration program can restrict the users that are allowed to use the SAS Protocol Interpreter. The Interpreter can also be configured to close a connection after a certain amount of time has passed.

Tips for Constructing Well-behaved Applets

Using Multiple Threads Within an Applet

The Java class `java.applet.Applet` is the base class for Java applets. It provides the following methods:

```
init(), start(), stop() and destroy()
```

These methods are called by the browser. The `init()` method is called when the applet is first loaded; `destroy()` is called when the applet is no longer needed. The `start()` method is called when the applet is visible to the user; `stop()` is called when the applet is no longer visible. `Start()` and `stop()` may be called multiple times, if the user is switching between HTML pages.

The applet will not be painted (i.e., the GUI components will not be visible to the user) until the `init()` method has exited. All of our sample applets establish communication with a remote machine. Establishing the connection may take a substantial amount of time. We feel it is unacceptable to write applets that do not provide status information about the state of a connection. Since the applet will not provide visual feedback until the `init()` method has completed, we do not establish connections in the `init()` method.

In all of our sample applets, we create a TextField called `status` in the `init()` method that displays status information, lay out the GUI (graphical user interface), and exit the method. The status area is now visible to the user.

Next, the applet’s `start()` method is called. Our sample applets create a separate thread to establish the connection to the remote SAS session. If the applet’s main thread is used to create the connection and the `stop()` method is called, the connection to the remote SAS session may be left in some unknown state. Many conditions exist under which the applet’s `stop()` method may be called. If the main applet thread is performing some operation that changes the state of the applet and the `stop()` method is called, the applet may be left in some unrecoverable state. If the applet is left in an unknown state, later the `start()` method may not be able to successfully restart the applet. Therefore, any work that is performed that changes the state of the applet should be performed in some thread other than the main applet thread. The applet thread can communicate with the other threads to inform them the `stop()` method has been called. Once informed that the applet will stop shortly, the other threads can complete whatever work they are doing and leave the applet in a state so it can be successfully restarted.

The thread that is establishing the connection to the remote SAS session can update the status field, so the user receives continuous feedback concerning the applet's activities.

Well-behaved applets will establish connections during the `start()` method and close the connections during the `stop()` method. Applets should release their resources when the `stop()` method has been called.

Using Archive Files

Java classes may be dynamically downloaded from the HTTP server or they may be bundled together in a Java Archive file (JAR file). If the applet requires a number of classes, it may take a significant amount of time to download each class file individually. Instead the classes may be bundled together to reduce download time. JAR files are a standard part of JDK 1.1. At the time this paper was written, neither Netscape nor Microsoft had a browser that supported JDK 1.1. However, both of these browsers supported some type of archive file. For Netscape Navigator, uncompressed zip files could be used to contain multiple Java class files; Microsoft Internet Explorer used CAB files. The sample applets provided by SAS Institute on its Web site make use of archive files. They significantly reduce the time it takes to start the applet and have it perform useful work.

Summary

This paper described the Java classes provided by SAS Institute. We provide a package for data access, the SAS/SHARE*NET driver for JDBC, and a package for remote SAS computing services. We provide Java classes and programs to support HTTP tunneling to alleviate some configuration restrictions and provide access to users behind a firewall. In addition, we provided some tips concerning applet behavior to help the reader construct well-behaved applets.

NOTE: *At the time of this paper, the code is still under development. We expect that results from further testing will change some of the functionality described here. For more up-to-date information, please visit the SAS Web site listed below and follow the product links.*

References

http://splash.javasoft.com/jdbc - Use this URL to review JDBC information.

http://www.sas.com/rnd/web - Use this URL to download a beta of the SAS/SHARE*NET driver for JDBC. SAS Institute provides sample code with the SAS/SHARE*NET driver for JDBC.

Author: Barbara Walters
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
(919) 677-8000 x6668
sasbbw@sas.com