Trail: Custom Networking

Lesson: All About Sockets

`URL`s and `URLConnection`s provide a relatively high-level mechanism for accessing resources on the Internet. Sometimes your programs require lower-level network communication, for example, when you want to write a client-server application.

In client-server applications, the server provides some service, such as processing database queries or sending out current stock prices. The client uses the service provided by the server, either displaying database query results to the user or making stock purchase recommendations to an investor. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it.

TCP provides a reliable, point-to-point communication channel that client-server applications on the Internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.

What Is a Socket?

A socket is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program. The java.net package provides two classes--Socket and ServerSocket--that implement the client side of the connection and the server side of the connection, respectively.

Reading from and Writing to a Socket

This page contains a small example that illustrates how a client program can read from and write to a socket.

Writing a Client/Server Pair

The previous page showed an example of how to write a client program that interacts with an existing server via a Socket object. This page shows you how to write a program that implements the other side of the connection--a server program.

Problems with the examples? Try Compiling and Running the Examples: FAQs.
Complaints? Compliments? Suggestions? Give us your feedback.

About Sun | About This Site | Terms of Use | Trademarks

Previous page: Previous Lesson
Next page: What Is a Socket?

Discuss

We welcome your participation in our community. Please keep your comments civil and on point. You may optionally provide your email address to be notified of replies — your information is not used for any other purpose. By submitting a comment, you agree to these Terms of Use.

Echo 1 Items Admin

Anil Patel
Excellent information here. This blog post made me smile. Maybe if you put in a couple of pics it will make the whole thing more interesting.
Building Contractors
Today, 2:54:55 AM – Flag – Like – Reply

Social Networking by

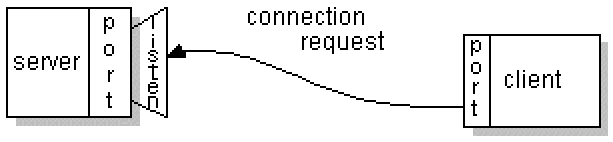
Leave a comment

Trail: Custom Networking
Lesson: All About Sockets

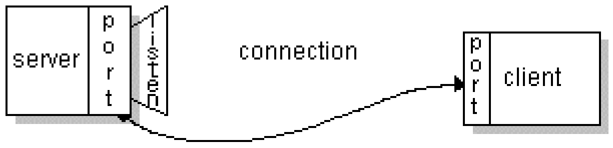
What Is a Socket?

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

Definition: A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent.

An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

The `java.net` package in the Java platform provides a class, `Socket`, that implements one side of a two-way connection between your Java program and another program on the network. The `Socket` class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the `java.net.Socket` class instead of relying on native code, your Java programs can communicate over the network in a platformindependent fashion.

Additionally, `java.net` includes the `ServerSocket` class, which implements a socket that servers can use to listen for and accept connections to clients. This lesson shows you how to use the `Socket` and `ServerSocket` classes.

If you are trying to connect to the Web, the `URL` class and related classes (`URLConnection`, `URLEncoder`) are probably more appropriate than the socket classes. In fact, URLs are a relatively high-level connection to the Web and use sockets as part of the underlying implementation. See Working with URLs for information about connecting to the Web via URLs.

Problems with the examples? Try Compiling and Running the Examples: FAQs.
Complaints? Compliments? Suggestions? Give us your feedback.

About Sun | About This Site | Terms of Use | Trademarks

A Sun Developer Network Site

Previous page: All About Sockets
Next page: Reading from and Writing to a Socket

Discuss
We welcome your participation in our community. Please keep your comments civil and on point. You may optionally provide your email address to be notified of replies — your information is not used for any other purpose. By submitting a comment, you agree to these Terms of Use.

Echo 0 Items Admin

Social Networking by

Leave a comment

Trail: Custom Networking
Lesson: All About Sockets

Reading from and Writing to a Socket

Let's look at a simple example that illustrates how a program can establish a connection to a server program using the `Socket` class and then, how the client can send data to and receive data from the server through the socket.

The example program implements a client, `EchoClient`, that connects to the Echo server. The Echo server simply receives data from its client and echoes it back. The Echo server is a well-known service that clients can rendezvous with on port 7.

`EchoClient` creates a socket thereby getting a connection to the Echo server. It reads input from the user on the standard input stream, and then forwards that text to the Echo server by writing the text to the socket. The server echoes the input back through the socket to the client. The client program reads and displays the data passed back to it from the server:

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

public class EchoClient {
    public static void main(String[] args) throws IOException {

        Socket echoSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {
            echoSocket = new Socket("taranis", 7);
            out = new PrintWriter(echoSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(
                                        echoSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: taranis.");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for "
                               + "the connection to: taranis.");
            System.exit(1);
        }

        BufferedReader stdIn = new BufferedReader(
                                   new InputStreamReader(System.in));
        String userInput;

        while ((userInput = stdIn.readLine()) != null) {
            out.println(userInput);
            System.out.println("echo: " + in.readLine());
        }

        out.close();
        in.close();
        stdIn.close();
        echoSocket.close();
        }
    }
```

Note that `EchoClient` both writes to and reads from its socket, thereby sending data to and receiving data from the Echo server.

Let's walk through the program and investigate the interesting parts. The three statements in the `try` block of the `main` method are critical. These lines establish the socket connection between the client and the server and open a `PrintWriter` and a `BufferedReader` on the socket:

```
echoSocket = new Socket("taranis", 7);
out = new PrintWriter(echoSocket.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader(
                             echoSocket.getInputStream()));
```

The first statement in this sequence creates a new `Socket` object and names it `echoSocket`. The `Socket` constructor used here requires the name of the machine and the port number to which you want to connect. The example program uses the host name `taranis`. This is the name of a hypothetical machine on our local network. When you type in and run this program on your machine, change the host name to the name of a machine on your network. Make sure that the name you use is the fully qualified IP name of the machine to which you want to connect. The second argument is the port number. Port number 7 is the port on which the Echo server listens.

The second statement gets the socket's output stream and opens a `PrintWriter` on it. Similarly, the third statement gets the socket's input stream and opens a `BufferedReader` on it. The example uses readers and writers so that it can write Unicode characters over the socket.

To send data through the socket to the server, `EchoClient` simply needs to write to the `PrintWriter`. To get the server's response, `EchoClient` reads from the `BufferedReader`. The rest of the program achieves this. If you are not yet familiar with the Java platform's I/O classes, you may wish to read Basic I/O.

The next interesting part of the program is the `while` loop. The loop reads a line at a time from the standard input stream and immediately sends it to the server by writing it to the `PrintWriter` connected to the socket:

```
String userInput;

while ((userInput = stdIn.readLine()) != null) {
    out.println(userInput);
    System.out.println("echo: " + in.readLine());
}
```

The last statement in the `while` loop reads a line of information from the `BufferedReader` connected to the socket. The `readLine` method waits until the server echoes the information back to `EchoClient`. When `readline` returns, `EchoClient` prints the information to the standard output.

The `while` loop continues until the user types an end-of-input character. That is, `EchoClient` reads input from the user, sends it to the Echo server, gets a response from the server, and displays it, until it reaches the end-of-input. The `while` loop then terminates and the program continues, executing the next four lines of code:

```
out.close();
in.close();
stdIn.close();
```

```
        echoSocket.close();
```

These lines of code fall into the category of housekeeping. A well-behaved program always cleans up after itself, and this program is well-behaved. These statements close the readers and writers connected to the socket and to the standard input stream, and close the socket connection to the server. The order here is important. You should close any streams connected to a socket before you close the socket itself.

This client program is straightforward and simple because the Echo server implements a simple protocol. The client sends text to the server, and the server echoes it back. When your client programs are talking to a more complicated server such as an HTTP server, your client program will also be more complicated. However, the basics are much the same as they are in this program:

1. Open a socket.
2. Open an input stream and output stream to the socket.
3. Read from and write to the stream according to the server's protocol.
4. Close the streams.
5. Close the socket.

Only step 3 differs from client to client, depending on the server. The other steps remain largely the same.

Problems with the examples? Try Compiling and Running the Examples: FAQs.
Complaints? Compliments? Suggestions? Give us your feedback.

About Sun | About This Site | Terms of Use | Trademarks

Previous page: What Is a Socket?
Next page: Writing the Server Side of a Socket

Discuss
We welcome your participation in our community. Please keep your comments civil and on point. You may optionally provide your email address to be notified of replies — your information is not used for any other purpose. By submitting a comment, you agree to these Terms of Use.

Echo 25 Items Admin

Gość
This example does not work on Windows XP. At least on my machine. You always get "UnknownHostException". Any ideas why?
Sunday, August 09, 2009, 12:23:18 PM – Flag – Like – Reply

Umer

Trail: Custom Networking
Lesson: All About Sockets

Writing the Server Side of a Socket

This section shows you how to write a server and the client that goes with it. The server in the client/server pair serves up Knock Knock jokes. Knock Knock jokes are favored by children and are usually vehicles for bad puns. They go like this:

Server: "Knock knock!"
Client: "Who's there?"
Server: "Dexter."
Client: "Dexter who?"
Server: "Dexter halls with boughs of holly."
Client: "Groan."

The example consists of two independently running Java programs: the client program and the server program. The client program is implemented by a single class, `KnockKnockClient`, and is very similar to the `EchoClient` example from the previous section. The server program is implemented by two classes: `KnockKnockServer` and `KnockKnockProtocol`, `KnockKnockServer` contains the `main` method for the server program and performs the work of listening to the port, establishing connections, and reading from and writing to the socket. `KnockKnockProtocol` serves up the jokes. It keeps track of the current joke, the current state (sent knock knock, sent clue, and so on), and returns the various text pieces of the joke depending on the current state. This object implements the protocol-the language that the client and server have agreed to use to communicate.

The following section looks in detail at each class in both the client and the server and then shows you how to run them.

The Knock Knock Server

This section walks through the code that implements the Knock Knock server program. Here is the complete source for the `KnockKnockServer` class.

The server program begins by creating a new `ServerSocket` object to listen on a specific port (see the statement in bold in the following code segment). When writing a server, choose a port that is not already dedicated to some other service. `KnockKnockServer` listens on port 4444 because 4 happens to be my favorite number and port 4444 is not being used for anything else in my environment:

```
try {
    serverSocket = new ServerSocket(4444);
} catch (IOException e) {
    System.out.println("Could not listen on port: 4444");
    System.exit(-1);
}
```

`ServerSocket` is a `java.net` class that provides a system-independent implementation of the server side of a client/server socket connection. The constructor for `ServerSocket` throws an exception if it can't listen on the specified port (for example, the port is already being used). In this case, the `KnockKnockServer` has no choice but to exit.

If the server successfully binds to its port, then the `ServerSocket` object is successfully created and the server continues to the next step--accepting a connection from a client (shown in bold):

```
Socket clientSocket = null;
try {
    clientSocket = serverSocket.accept();
} catch (IOException e) {
    System.out.println("Accept failed: 4444");
    System.exit(-1);
}
```

The `accept` method waits until a client starts up and requests a connection on the host and port of this server (in this example, the server is running on the hypothetical machine taranis on port 4444). When a connection is requested and successfully established, the accept method returns a new `Socket` object which is bound to the same local port and has it's remote address and remote port set to that of the client. The server can communicate with the client over this new `Socket` and continue to listen for client connection requests on the original `ServerSocket` This particular version of the program doesn't listen for more client connection requests. However, a modified version of the program is provided in Supporting Multiple Clients.

After the server successfully establishes a connection with a client, it communicates with the client using this code:

```
PrintWriter out = new PrintWriter(
                      clientSocket.getOutputStream(), true);
BufferedReader in = new BufferedReader(
                        new InputStreamReader(
                            clientSocket.getInputStream()));
String inputLine, outputLine;

// initiate conversation with client
KnockKnockProtocol kkp = new KnockKnockProtocol();
outputLine = kkp.processInput(null);
out.println(outputLine);

while ((inputLine = in.readLine()) != null) {
    outputLine = kkp.processInput(inputLine);
    out.println(outputLine);
    if (outputLine.equals("Bye."))
        break;
}
```

This code:

1. Gets the socket's input and output stream and opens readers and writers on them.
2. Initiates communication with the client by writing to the socket (shown in bold).
3. Communicates with the client by reading from and writing to the socket (the `while` loop).

Step 1 is already familiar. Step 2 is shown in bold and is worth a few comments. The bold statements in the code segment above initiate the conversation with the client. The code creates a `KnockKnockProtocol` object-the object that keeps track of the current joke, the current state within the joke, and so on.

After the `KnockKnockProtocol` is created, the code calls `KnockKnockProtocol`'s `processInput` method to get the first message that the server sends to the client. For this example, the first thing that the server says is "Knock! Knock!" Next, the server writes the information to the `PrintWriter` connected to the client socket, thereby sending the message to the client.

Step 3 is encoded in the `while` loop. As long as the client and server still have something to say to each other, the server reads from and writes to the socket, sending messages back and forth between the client and the server.

The server initiated the conversation with a "Knock! Knock!" so afterwards the server must wait for the client to say "Who's there?" As a result, the while loop iterates on a read from the input stream. The readLine method waits until the client responds by writing something to its output stream (the server's input stream). When the client responds, the server passes the client's response to the `KnockKnockProtocol` object and asks the `KnockKnockProtocol` object for a suitable reply. The server immediately sends the reply to the client via the output stream connected to the socket, using a call to println. If the server's response generated from the `KnockKnockServer` object is "Bye." this indicates that the client doesn't want any more jokes and the loop quits.

The `KnockKnockServer` class is a well-behaved server, so the last several lines of this section of `KnockKnockServer` clean up by closing all of the input and output streams, the client socket, and the server socket:

```
out.close();
in.close();
clientSocket.close();
serverSocket.close();
```

The Knock Knock Protocol

The `KnockKnockProtocol` class implements the protocol that the client and server use to communicate. This class keeps track of where the client and the server are in their conversation and serves up the server's response to the client's statements. The `KnockKnockServer` object contains the text of all the jokes and makes sure that the client gives the proper response to the server's statements. It wouldn't do to have the client say "Dexter who?" when the server says "Knock! Knock!"

All client/server pairs must have some protocol by which they speak to each other; otherwise, the data that passes back and forth would be meaningless. The protocol that your own clients and servers use depends entirely on the communication required by them to accomplish the task.

The Knock Knock Client

The `KnockKnockClient` class implements the client program that speaks to the `KnockKnockServer`. `KnockKnockClient` is based on the `EchoClient` program in the previous section, Reading from and Writing to a Socket and should be somewhat familiar to you. But we'll go over the program anyway and look at what's happening in the client in the context of what's going on in the server.

When you start the client program, the server should already be running and listening to the port, waiting for a client to request a connection. So, the first thing the client program does is to open a socket that is connected to the server running on the hostname and port specified:

```
kkSocket = new Socket("taranis", 4444);
out = new PrintWriter(kkSocket.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader(
                            kkSocket.getInputStream()));
```

When creating its socket, `KnockKnockClient` uses the host name `taranis`, the name of a hypothetical machine on our network. When you type in and run this program, change the host name to the name of a machine on your network. This is the machine on which you will run the `KnockKnockServer`.

The `KnockKnockClient` program also specifies the port number 4444 when creating its socket. This is a *remote port number*--the number of a port on the server machine--and is the port to which `KnockKnockServer` is listening. The client's socket is bound to any available *local port*--a port on the client machine. Remember that the server gets a new socket as well. That socket is bound to local port number 4444 on its machine. The server's socket and the client's socket are connected.

Next comes the while loop that implements the communication between the client and the server. The server speaks first, so the client must listen first. The client does this by reading from the input stream attached to the socket. If the server does speak, it says "Bye." and the client exits the loop. Otherwise, the client displays the text to the standard output and then reads the response from the user, who types into the standard input. After the user types a carriage return, the client sends the text to the server through the output stream attached to the socket.

```
while ((fromServer = in.readLine()) != null) {
    System.out.println("Server: " + fromServer);
    if (fromServer.equals("Bye."))
        break;
    fromUser = stdIn.readLine();
    if (fromUser != null) {
        System.out.println("Client: " + fromUser);
        out.println(fromUser);
    }
}
```

The communication ends when the server asks if the client wishes to hear another joke, the client says no, and the server says "Bye."

In the interest of good housekeeping, the client closes its input and output streams and the socket:

```
out.close();
in.close();
stdIn.close();
kkSocket.close();
```

Running the Programs

You must start the server program first. To do this, run the server program using the Java interpreter, just as you would any other Java application. Remember to run the server on the machine that the client program specifies when it creates the socket.

Next, run the client program. Note that you can run the client on any machine on your network; it does not have to run on the same machine as the server.

If you are too quick, you might start the client before the server has a chance to initialize itself and begin listening on the port. If this happens, you will see a stack trace from the client. If this happens, just restart the client.

If you forget to change the host name in the source code for the `KnockKnockClient` program, you will see the following error message:

```
Don't know about host: taranis
```

To fix this, modify the `KnockKnockClient` program and provide a valid host name for your network. Recompile the client program and try again.

If you try to start a second client while the first client is connected to the server, the second client just hangs. The next section, Supporting Multiple Clients, talks about supporting multiple clients.

When you successfully get a connection between the client and server, you will see the following text displayed on your screen:

```
Server: Knock! Knock!
```

Now, you must respond with:

```
Who's there?
```

The client echoes what you type and sends the text to the server. The server responds with the first line of one of the many Knock Knock jokes in its repertoire. Now your screen should contain this (the text you typed is in bold):

```
Server: Knock! Knock!
Who's there?
Client: Who's there?
Server: Turnip
```

Now, you respond with:

```
Turnip who?"
```

Again, the client echoes what you type and sends the text to the server. The server responds with the punch line. Now your screen should contain this:

```
Server: Knock! Knock!
Who's there?
Client: Who's there?
Server: Turnip
Turnip who?
Client: Turnip who?
Server: Turnip the heat, it's cold in here! Want another? (y/n)
```

If you want to hear another joke, type **y**; if not, type **n**. If you type **y**, the server begins again with "Knock! Knock!" If you type **n**, the server says "Bye." thus causing both the client and the server to exit.

If at any point you make a typing mistake, the `KnockKnockServer` object catches it and the server responds with a message similar to this:

```
Server: You're supposed to say "Who's there?"!
```

The server then starts the joke over again:

```
Server: Try again. Knock! Knock!
```

Note that the `KnockKnockProtocol` object is particular about spelling and punctuation but not about capitalization.

Supporting Multiple Clients

To keep the `KnockKnockServer` example simple, we designed it to listen for and handle a single connection request. However, multiple client requests can come into the same port and, consequently, into the same `ServerSocket`. Client connection requests are queued at the port, so the server must accept the connections sequentially. However, the server can service them simultaneously through the use of threads - one thread per each client connection.

The basic flow of logic in such a server is this:

```
while (true) {
    accept a connection ;
    create a thread to deal with the client ;
end while
```

The thread reads from and writes to the client connection as necessary.

Try This: Modify the `KnockKnockServer` so that it can service multiple clients at the same time. Two classes compose our solution: `KKMultiServer` and `KKMultiServerThread`. `KKMultiServer` loops forever, listening for client connection requests on a `ServerSocket`. When a request comes in, `KKMultiServer` accepts the connection, creates a new `KKMultiServerThread` object to process it, hands it the socket returned from accept, and starts the thread. Then the server goes back to listening for connection requests. The `KKMultiServerThread` object communicates to the client by reading from and writing to the socket. Run the new Knock Knock server and then run several clients in succession.

Problems with the examples? Try Compiling and Running the Examples: FAQs.
Complaints? Compliments? Suggestions? Give us your feedback.