Java 109 : Networking with Java

Previous | Next

In this tutorial, we'll cover networking with Java. Don't worry if you're not familiar with networking - this will be only a brief introduction. We'll examine some of the classes in the java.net package, and show you how to write a simple network client in Java. First, however, we need to cover some background theory.

How do computers talk to each other via the Internet?

The Internet is composed of millions of computers, located all across the globe, communicating and transmitting information over a variety of computing systems, platforms, and networking equipment. Each of these computers (unless they are connecting via an intranet) will have a unique IP address.

IP addresses are 32-bit numbers, containing four octets (8 bit numbers) separated by a full stop. Each computer with a direct internet connection will have a unique IP address, (e.g. 207.68.156.61). Some computers have temporary addresses, such as when you connect to your ISP through a modem. Others have permanent addresses, and some even have their own unique domain names (e.g. www.microsoft.com).

An IP address allows us to uniquely identify a device or system connected to the Internet. If I wanted to connect to a specific IP address, and send a message, I could do so. Without an IP address, my message would have no way of reaching its destination - a bit like leaving the address off a letter or parcel.

Often, computers connected to the Internet provide services. This page is provided by a web server, for example. Because computers are capable of providing more than one type of service, we need a way to uniquely identify each service. Like an IP address, we use a number. We call this number a port. Common services (such as HTTP, FTP, Telnet, SMTP) have well known port numbers. For example, most web servers use port 80. Of course, you can use any port you like - there's no rule that says you *must* use 80.

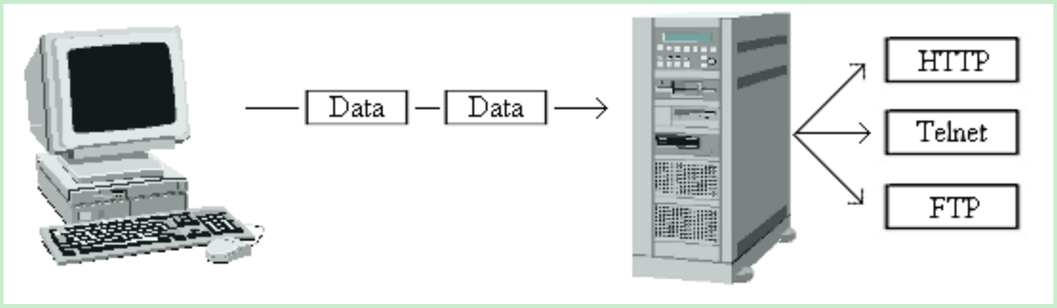

Figure 1.0 - Ports help computers identify which service data is for.

There are several communications mechanisms that we can use to provide network services. We could use UDP (unreliable datagram protocol), or TCP (transfer-control protocol). For the purposes of this tutorial, we'll choose TCP, because it makes life much easier. TCP guarantees that messages will arrive at their destination. UDP is unreliable, and your application isn't notified if the message is lost in transit. Also, many protocols (such as HTTP, SMTP, POP & FTP) use TCP, so it's important that you are familiar with it for networking in Java.

Internet Addressing with Java

Handling internet addresses (domain names, and IP addresses) is made easy with Java. Internet addresses are represented in Java by the InetAddress class. InetAddress provides simple methods to convert between domain names, and numbered addresses.

We start by importing the java.net package, which contains a set of pre-written networking routines (including InetAddress).

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

Next, we declare a new variable of type InetAddress, which we assign the value of the local host machine (for machines not connected to a network, this should represent 127.0.0.1). Due to the fact that InetAddresses can generate exceptions, we must place this code between a try .. catch UnknownHostException block.

```
// Obtain the InetAddress of the computer on which this program is running
InetAddress localaddr = InetAddress.getLocalHost();
```

The InetAddress class has methods that return the IP address as an array of bytes (which can be easily converted into a string), as well as a string representation of its domain name (e.g. mydomain.org). We can print out the InternetAddress, as well as the domain name of the local address.

```
System.out.println ("Local IP Address : " + localaddr );
System.out.println ("Local hostname : " + localaddr.getHostName());
```

Download the source code

```
public class MyFirstInternetAddress
{
	public static void main(String args[])
	{
		try
		{
			InetAddress localaddr = InetAddress.getLocalHost();

			System.out.println ("Local IP Address : " + localaddr );
			System.out.println ("Local hostname   : " + localaddr.getHostName());
```

```
                }
                catch (UnknownHostException e)
                {
                        System.err.println ("Can't detect localhost : " + e);
                }

        }

        /** Converts a byte_array of octets into a string */
        public static String byteToStr( byte[] byte_arr )
        {
                StringBuffer internal_buffer = new StringBuffer();

                // Keep looping, and adding octets to the IP Address
                for (int index = 0; index < byte_arr.length -1; index++)
                {
                        internal_buffer.append ( String.valueOf(byte_arr[index]) + ".");
                }

                // Add the final octet, but no trailing '.'
                internal_buffer.append ( String.valueOf (byte_arr.length) );

                return internal_buffer.toString();
        }
}
```

Compile and run this application, and you should be told your local IP address, and hostname. Don't worry if your computer isn't connected to the Internet, though. Providing your system has a TCP stack, it should give you back an IP address even if you aren't currently connected. On most systems, you can refer to your local machine (which often has the hostname "localhost") as IP address 127.0.0.1

Why would every machine that's not connected to the Internet have the same address? This address is known as a loopback address. Every time you connect to this address, you're actually connected to your local machine. So, if you were running a local webserver, and you pointed your browser to http://127.0.0.1, you should see your web-site. But if I were to go to the same address, I'd connect to a different site - that of my own machine.

This is great when developing Java applications. You don't need a permanent connection to the Internet - you can run client and server applications on your own machine. This is handy, because writing and testing client/server applications can take some time, and unless you have a permanant connection, you wouldn't want to be billed on an hourly rate by your ISP!

Writing a TCP client in Java

Writing network client in Java is very simple. If you've ever written a network client in C, you'll know how complicated it can be. You have to be concerned with structures, and pointers. Java cuts out this complexity, through its java.net.Socket class. To demonstrate just how easy Java makes it, I'm going to show you how to write a finger client.

For those who are unfamiliar with the finger protocol, I'll briefly explain how it works. Finger allows a remote user to query a host machine for information, either about the host machine in general or a specific user. Most unix systems support finger, and many non-Unix systems also support the protocol. Most finger applications take as a paramater 'username@hostmachine'.

Finger clients connect to a host server at port 79 and establish a TCP stream. The client sends the username (or a blank, for a general query), followed by a newline character. The server then sends back information about the user, in the form of a text stream. This should be displayed to the user, and then the connection should be terminated.

As with any networking application in Java, we need to first import the network and input/output packages.

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

Our application should have a single method, main, which is responsible for issuing a finger query. The first step is to validate and parse the command line paramaters, looking for a username and hostname.

```
public static void main ( String args[] )
{
        // Check command line paramaters
        if (args.length != 1)
        {
                System.err.println ("Invalid paramaters");
                System.exit(1);
        }
        else
        // Check for existence of @ in paramater
        if (args[0].indexOf("@") == -1)
        {
                System.err.println ("Invalid paramater : syntax user@host");
                System.exit(1);
        }

        // Split command line paramater at the @ character
        String username = args[0].substring(0, args[0].indexOf("@") );
        String hostname = args[0].substring(args[0].indexOf("@") +1, args[0].length());

        ........
}
```

In the code above, we check that only a single paramater has been entered, and also that there exists an '@' character in the paramater. The next step is to split the command line paramater into a username and a hostname. To do this, we rely on the substring method which can be applied to any string. Username becomes the string from offset 0, to the first index of character

'@'. Hostname becomes the string from the first index of character '@', to the length of the original paramater.

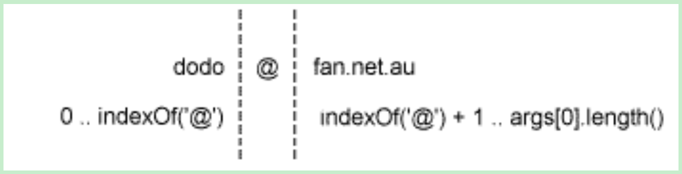

Figure 2.0 - Extracting username & hostname from command-line parameter

The next step is to connect to the finger server, which opperates on port 79. As with the previous example, we must enclose our network code inside of a try { ... } catch block. This allows us to trap any network errors that may occur (such as invalid hostnames, or an inability to connect with the server). You'll notice that the code to create a TCP collection is actually only a single line - networking in Java is very easy.

```
try
{
        // Create a connection to server
        Socket s = new Socket(hostname, 79);

        // Remainder of finger client code goes here
        .........

}
catch (SocketException e )
{
        System.err.println ("Socket error : " + e);
}
catch (UnknownHostException e )
{
        System.err.println ("Invalid host!");
}
catch (IOException e )
{
        System.err.println ("I/O error : " + e);
}
```

After connecting to port 79 of the finger server, we now have to obtain input and output streams for the socket. We can treat these streams then just as we would file or text input and output. For ease of use we'll covert the input stream into a DataInputStream, and the output stream into a PrintStream. This will allow us to use the readLine and println methods with our socket streams.

```
// Create input and output streams to socket
PrintStream out = new PrintStream( s.getOutputStream()) ;
DataInputStream in = new DataInputStream(s.getInputStream());
```

Using our PrintStream out, we write the name of the user we wish to find out information about to the finger server. The server will process the query, and output a result, which we will print to the user's screen.

```
// Write username to socket output
out.println( username );

// Read response from socket
System.out.println ( in.readLine() );

// Read remaining finger response
String line = in.readLine();

while (line != null)
{
        System.out.println ( line );

        // Read next line
        line = in.readLine();
}
```

The first line is read from the input stream, and then printed to screen. We then enter a loop, checking to see if there are any more lines to display. The while loop terminates when there are no more bytes available.

Finally, we must close the connection to our finger server. With this last statement, our finger client is complete!

```
// Terminate connection
s.close();
```

While the example program functions as a finger client, it can easily be used as a TCP client skeleton, substituting the connection to port 79 with another port matching the application protocol you are trying to use, and modifying the read/write routines to send different data.

Running the example client

Download the source code

To run the client, you'll need to be connected to the Internet (this example requires a direct connection, and won't run behind a firewall). To execute, you need to know the name of a user, and the site on which the user resides. For example, to finger dodo@fan.net.au, you would do the following

```
java TCP_Finger_Client dodo@fan.net.au
```

Tip - Not all ISP's run finger servers, so you won't be able to finger every user. The example above should work, however (providing my ISP doesn't change its policy).

Summary

Writing network applications in Java is extremely easy. Java takes away much of the implementation details which are operating system specific. There's no need to worry about hostname lookups, size of IP address structures, or pointers. The java.net package provides all this functionality for you, and provides a simple way to write networking clients. However, when writing Java applets (as opposed to applications), remember that the browser may impose security restrictions - many applets can only communicate with the host from which they were downloaded from.

Additional Resources

Merlin Hughes, et al. Java Network Programming, Manning Publications, 1997.

Reilly, D. Java Network Programming FAQ (online) available at http://www.davidreilly.com/java/java_network_programming/

Previous | Next