Handling network timeouts in Java

By David Reilly

When writing network applications in a stable and controlled environment, it is easy to forget what the real world is like. Slow connections, traffic build ups, or power interruptions can cause network connections to stall or even die. Few programmers take the time to detect and handle network timeouts, but avoiding the problem can cause client applications to freeze, and for threads in servers to block indefinitely. There are, however, easy ways to handle network timeouts. In this article, I'll present two techniques for overcoming the problem of network timeouts - threads and setting a socket option for timeouts.

Overview

Handling timeouts is one of those tasks that people generally leave to the last moment. In an ideal world, we'd never need them. In an intranet environment, where most programmers do their development, it's almost always unnecessary. However, on the Internet, good timeout handling is critical. Badly behaved clients may go offline and leaving server threads locked, or overloaded servers may stall, causing a network client to block indefinitely for input. For these reasons, it is necessary to detect and handle network timeouts.

I've identified two relatively simple solutions to the problem of handling network timeouts. The first solution involves the use of second thread, which acts as a timer. This solution results in a slight increase in complexity, but is backwards compatible with JDK1.02. The second solution is by far, much simpler. It takes only a few lines of code, by setting a socket option, and catching a timeout exception. The catch is, that it requires a JDK1.1 or higher virtual machine.

Solution One : Using a timer thread

Many network software (particularly servers), are written as multi-threaded applications. However, a client can also use multiple threads. One solution to handling network timeouts is to launch a secondary thread, the 'timer', and have it cancel the application gracefully if it becomes stalled. This prevents end users from becoming confused when the application stalls - a good error message will at least let them know the cause of the problem.

Listing One shows a Timer, which can be used in networking applications to handle timeouts gracefully. Once the timer is started, it must be reset regularly, such as when data is returned by a server. However, if the thread that reads from a remote network host becomes stalled, the timer will exit with an error message. For those who require a custom handler, the timeout() method may be overridden to provide different functionality.

Listing One - Timer.java

```
/**
  * The Timer class allows a graceful exit when an application
  * is stalled due to a networking timeout. Once the timer is
  * set, it must be cleared via the reset() method, or the
  * timeout() method is called.
  * <p>
  * The timeout length is customizable, by changing the 'length'
  * property, or through the constructor. The length represents
  * the length of the timer in milliseconds.
  *
  * @author     David Reilly
  */
public class Timer extends Thread
{
        /** Rate at which timer is checked */
        protected int m_rate = 100;

        /** Length of timeout */
        private int m_length;

        /** Time elapsed */
        private int m_elapsed;

        /**
          * Creates a timer of a specified length
          * @param      length  Length of time before timeout occurs
          */
        public Timer ( int length )
        {
                // Assign to member variable
                m_length = length;

                // Set time elapsed
                m_elapsed = 0;
        }


        /** Resets the timer back to zero */
        public synchronized void reset()
        {
                m_elapsed = 0;
        }

        /** Performs timer specific code */
        public void run()
        {
                // Keep looping
                for (;;)
                {
                        // Put the timer to sleep
                        try
                        {
                                Thread.sleep(m_rate);
                        }
                        catch (InterruptedException ioe)
                        {
                                continue;
                        }

                        // Use 'synchronized' to prevent conflicts
                        synchronized ( this )
                        {
                                // Increment time remaining
                                m_elapsed += m_rate;

                                // Check to see if the time has been exceeded
                                if (m_elapsed > m_length)
                                {
                                        // Trigger a timeout
                                        timeout();
                                }
                        }

                }
        }

        // Override this to provide custom functionality
        public void timeout()
        {
                System.err.println ("Network timeout occurred.... terminating");
                System.exit(1);
        }
}
```

To illustrate the use of the Timer class, here is a simple TCP client (Listing Two) and server (Listing Three). The client sends a text string across the network, and then reads a response. While reading, it would become blocked if the server stalled, or if the server took too long to accept the connection. For this reason, a timer is started before connecting, and then reset after each major operation. Starting the timer is relatively simple, but it must be reset after each blocking operation or the client will terminate.

```
// Start timer
Timer timer = new Timer(3000);
timer.start();

// Perform some read operation
......

// Reset the timer
timer.reset();
```

The server is relatively simple - it's a single-threaded application, which simulates a stalled server. The server reads a response from the client, and echoes it back.To demonstrate timeouts, the server will always "stall" for a period of twenty seconds, on every second connection. Remember though, in real life, server timeouts are entirely unpredictable, and will not always correct themselves after several seconds of delay.

Listing Two

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

/**
  * SimpleClient connects to TCP port 2000, writes a line
  * of text, and then reads the echoed text back from the server.
  * This client will detect network timeouts, and exit gracefully,
  * rather than stalling.
  * Start server, and the run by typing
  *
  *  java SimpleClient
  */
public class SimpleClient
{
        /** Connects to a server on port 2000,
            and handles network timeouts gracefully
          */
        public static void main (String args[]) throws Exception
        {
                System.out.println ("Starting timer.");
                // Start timer
                Timer timer = new Timer(3000);
                timer.start();

                // Connect to remote host
                Socket socket = new Socket ("localhost", 2000);
                System.out.println ("Connected to localhost:2000");

                // Reset timer - timeout can occur on connect
                timer.reset();

                // Create a print stream for writing
                PrintStream pout = new PrintStream (
                        socket.getOutputStream() );

                // Create a data input stream for reading
                DataInputStream din = new DataInputStream(
                        socket.getInputStream() );

                // Print hello msg
                pout.println ("Hello world!");

                // Reset timer - timeout is likely to occur during the read
                timer.reset();

                // Print msg from server
                System.out.println (din.readLine());

                // Shutdown timer
                timer.stop();

                // Close connection
                socket.close();
        }
}
```

Listing Three

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

/**
  * SimpleServer binds to TCP port 2000, reads a line
  * of text, and then echoes it back to the user. To
  * demonstrate a network timeout, every second connection
  * will stall for twenty seconds.
  *
  * Start server by typing
  *
  *  java SimpleServer
  */
public class SimpleServer extends Thread
{
        /** Shall we stall? flag */
        protected static boolean shallWeStall = false;

        /** Socket connection */
        private Socket m_connection;

        /**
```

```
      * Constructor, accepting a socket connection
      *     @param  connection     Connection to process
      */
    public SimpleServer (Socket connection)
    {
            // Assign to member variable
            m_connection = connection;
    }

    /** Starts a simple server on port 2000 */
    public static void main (String args[]) throws Exception
    {
            ServerSocket server = new ServerSocket (2000);

            for (;;)
            {
                    // Accept an incoming connection
                    Socket connection = server.accept();

                    // Process in another thread
                    new SimpleServer(connection).start();
            }
    }

    /** Performs connection handling */
    public void run()
    {
            try
            {
                    DataInputStream din = new DataInputStream (
                            m_connection.getInputStream() );

                    PrintStream pout = new PrintStream (
                            m_connection.getOutputStream() );

                    // Read line from client
                    String data = din.readLine();

                    // Check to see if we should simulate a stalled server
                    if (shallWeStall)
                    {
                            // Yes .. so reset flag and stall
                            shallWeStall = false;

                            try
                            {
                                    Thread.sleep (20000);
                            } catch (InterruptedException ie ) {}
                    }
                    else
                    {
                            // No.... but we will next time
                            shallWeStall = true;
                    }

                    // Echo data back to clinet
                    pout.println (data);

                    // Close connection
                    m_connection.close();
            }
            catch (IOException ioe)
            {
```

```
                        System.err.println ("I/O error");
                }
        }
}
```

Solution Two : Simplified timeout handling with socket options

A significantly easier solution to handling network timeouts is to set a socket option. Socket options allow programmers greater control over socket communication, and are supported by Java as of JDK1.1. One socket option in particular, SO_TIMEOUT, is extremely useful, because it allows programmers to specify an amount of time that a read operation will block for, before generating an *java.io.InterruptedIOException*, which can be easily caught to provide a timeout handler.

We can specify a value for SO_TIMEOUT, by using the setSoTimeout() method. This method is supported by *java.net.Socket*, *java.net.DatagramSocket*, and *java.net.ServerSocket*. This is important, as it means we can handle timeouts in both the client and the server. The setSoTimeout accepts as its sole parameter an *int*, which represents the number of milliseconds an operation may block for. Settings this value to one thousand will result in timeouts after one second of inactivity, whereas setting SO_TIMEOUT to zero will allow the thread to block indefinitely.

```
// Set SO_TIMEOUT for five seconds
MyServerSocket.setSoTimeout(5000);
```

Once the length for the timeout is set, any blocking operation will cause an InterruptedIOException to be thrown. For example, a DatagramSocket that failed to receive packets when the receive() method is called, would throw an exception. This makes it easy to separate our timeout handling code from the task of communicating via the network.

The next example, Listing Four, shows another multi-threaded echo server with timeout support. It limits the number of connections it can support (currently set to two), and rejects further connections. However, if a client fails to send data, a server thread will become blocked, and the number of available connections will be reduced. With larger servers, and hundreds or thousands of connections per hour, blocked threads become a significant problem, and can lead to denial of service. This example shows how to detect a timeout, and to disconnect gracefully. By using socket_options, and catching exceptions, it's easier to shut down an individual thread.

To test the echo server, you can use the telnet command (available on both Unix/Wintel systems), and connect to port 2000 of your local machine. Type a few characters of text, and then watch what happens after thirty seconds of inactivity. The server will automatically disconnect the telnet client, freeing up the connection for another user.

Listing Four

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

/**
  * EchoServer offers an echo service to multiple clients.
  * The echo service is limited to a set number of connections,
  * to prevent server over-load. It also includes timeout handling
  * code to prevent server threads from blocking if a client is
  * stalled.
  */
public class EchoServer extends Thread
{
        /** Connection to client */
        private Socket m_connection;

        /** Number of connections */
        private static int number_of_connections = 0;

        /** Maximum number of connections */
        private static final int max_connections = 2;

        /** Port to bind to */
        private static final int service_port = 2000;

        /** Timeout length */
        private static final int timeout_length = 30000;

        /**
          * Creates a new instance of EchoServer thread, to
          * service the specified client connection.
          *
          * @param connection  Connection to service
          */
        public EchoServer (Socket connection)
        {
                // Assign to member variable
                m_connection = connection;

                // Set a timeout of 'timeout_length' milliseconds
                try
                {
                        m_connection.setSoTimeout (timeout_length);
                }
                catch (SocketException se)
                {
                        System.err.println ("Unable to set socket option SO_TIMEOUT");
                }

                // Increment number of connections
                number_of_connections++;
        }

        public void run()
        {
                try
                {
                        // Get I/O streams
                        InputStream  in = m_connection.getInputStream();
                        OutputStream out= m_connection.getOutputStream();

                        try
                        {
                                for (;;)
                                        // Echo data straight back to client
                                        out.write(in.read());
                        }
                        catch (InterruptedIOException iioe)
                        {
                                System.out.println ("Timeout occurred - killing
connection");
                                m_connection.close();
                }
            }
            catch (IOException ioe)
            {
                // No code required - thread will terminate at end
                // of the run method
            }

            // Decrement the number of connections
            number_of_connections--;
        }

        public static void main(String args[]) throws Exception
        {
            // Bind to a local port
            ServerSocket server = new ServerSocket (service_port);

            for (;;)
            {
                // Accept the next connection
                Socket connection = server.accept();

                // Check to see if maximum reached
                if (number_of_connections > max_connections-1)
                {
                    // Kill the connection
                    PrintStream pout = new PrintStream
(connection.getOutputStream());
                    pout.println ("Too many users");
                    connection.close();
                    continue;
                }
                else
                {
                    // Launch a new thread
                    new EchoServer(connection).start();
                }
            }
        }
}
```

Summary

All good network applications will include timeout detection and handling. Whether you're writing a client, and need to detect a wayward server, or writing a server and need to prevent stalled connections, timeout handling is a critical part of error handling. For those who require backwards compatibility with JDK1.02, timers may be used to detect stalled connections. The most preferable solution though, is to use socket options, and to provide an exception handler for *java.io.InterruptedIOException*. This reduces the amount of code required to handle timeouts, and makes for a cleaner design.