

Sockets

Sockets

- Sockets make network connections between machines, but you just read/write/block on them like there were plain file streams. The Internet is basically built on sockets.

Client Socket

- Make connection to host name "127.0.0.1" or "localhost" (the local machine itself) or "elaine26.stanford.edu" (machine on the internet) + a port number on that machine.
 - `Socket toServer = new Socket(host, port);` // make connection
 - `OutputStream out = toServer.getOutputStream();` // write to this
 - `InputStream in = toServer.getInputStream();` // read from this
- Reads will block if there is no data (do not do on swing thread!)
- Writes go through fast, so ok to do on swing thread (could fork off a thread to do it)
- Can wrap each stream in `ObjectInputStream/ObjectOutputStream` to send whole objects, e.g. `String`, -- a low budget way to do network i/o without a lot of parsing, although not the most efficient.

Server Sockets / accept()

- The server thread creates a sever socket and calls `accept()` to wait (block) for incoming client connections on a particular port number.
- On unix, ports under 1024 are "privileged" so regular users must use high port numbers, like 8000 or 3456.
- The `accept()` call blocks waiting for an incoming connection, and then returns a new socket, one for each incoming client. Typically you deal with the new connection, and then loop around and block in `accept` again.
- Get input and output streams, as above, for each client
- See the `ServerAcceptor` example below.

Blocking / Flushing

- Reading on a socket when there is no data will block -- so you can't do that on the swing thread
- Likewise, the server blocks in `accept()`, waiting for new client connections
- Writing on a socket may "buffer" the data to send it all in a big chunk. Use `flush()` on a stream to force the accumulated data to go out now. When you `close()` on a stream when you are done with it, that does an implicit `flush()` to send all the data.

XMLString Strategy -- Writing

- Create an xml String representation of a Message object using encoder. Use `WriteObject()` to send the string on the socket.

```
// Convert the message object into an xml string.
OutputStream memStream = new ByteArrayOutputStream();
XMLEncoder encoder = new XMLEncoder(memStream);
encoder.writeObject(message);
encoder.close();
String xmlString = memStream.toString();
```

r

XMLString Strategy -- Reading

- Use `readObject()` to get the string, then xml decoder to recreate the Message object.

```
// Get the xml string, decode to a Message object.
String xmlString = (String) in.readObject();
XMLDecoder decoder = new XMLDecoder(new ByteArrayInputStream(xmlString.getBytes()));
Message message = (Message) decoder.readObject();
```

Ticker GUI/Socket Example

- Message -- a little struct bean that contains a Date and a String
- Server button -> accepts client connections. Starts a ServerAcceptor thread.
- Server keeps a list of all the connections to clients -- sends messages to all of them.
- Client button -> connects to a server and listens for incoming messages, posts them to its GUI.
- Complete code available in hw directory

Ticker Example Code

```
//TickerExample.java
/*
Demonstrates using client and server sockets with a GUI.
One server ticker can support any number of client tickers --
sortof a primitive, one-way instant messenger.
Uses xml encoding to send a little data struct Message object.
*/
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.beans.XMLDecoder;
import java.beans.XMLEncoder;

import javax.swing.*;
import java.util.*;
import java.io.*;
import java.net.*;

public class TickerExample extends JFrame {
    private JTextArea textArea;
    private JTextField field;
    private JLabel status;

    // The are thread inner classes to handle
    // the networking.
    private ClientHandler clientHandler;
    private ServerAcceptor serverAcceptor;

    // List of object streams to which we send data
    private java.util.List<ObjectOutputStream> outputs =
        new ArrayList<ObjectOutputStream>();

    public static void main(String[] args) {
        // Prefer the "native" look and feel.
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception ignored) { }

        for (int i=0 ;i<3; i++ ) { // for testing, handy to make a few at a time
            new TickerExample();
        }
    }

    public TickerExample() {
        setTitle("Ticker");

        JComponent box = new JPanel();
        box.setLayout(new BoxLayout(box, BoxLayout.Y_AXIS));
        setContentPane(box);

        textArea = new JTextArea(20, 20);
        add(new JScrollPane(textArea), BorderLayout.CENTER);
```

```

        JButton button;
        button = new JButton("Start Server");
        box.add(button);
        button.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                doServer();
            }
        });

        button = new JButton("Start Client");
        box.add(button);
        button.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                doClient();
            }
        });

        field = new JTextField(15);
        JPanel panel = new JPanel();
        panel.setMinimumSize(new Dimension(200, 30));
        panel.add(field);
        box.add(panel);
        field.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                doSend();
            }
        });

        status = new JLabel();
        box.add(status);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }

    // Struct object just used for communication -- sent on the object stream.
    // Declared "static", so does not contain a pointer to the outer object.
    // Bean style, set up for xml encode/decode.
    public static class Message {
        public String text;
        public Date date;

        public Message() {
            text = null;
            date = null;
        }

        public String getText() {
            return text;
        }

        public void setText(String text) {
            this.text = text;
        }

        public Date getDate() {
            return date;
        }

        public void setDate(Date date) {
            this.date = date;
        }

        public String toString() {
            return "message: " + text;
        }
    }

    // Appends a message to the local GUI (must be on swing thread)
    public void sendLocal(Message message) {

```

```

        textArea.setText(textArea.getText() + message.getText() + "\n" + message.getDate() + "\n\n");
    }

    // Initiate message send -- send both local and remote (must be on swing thread)
    // Wired to text field.
    public void doSend() {
        Message message = new Message();
        message.setText(field.getText());
        message.setDate(new Date());
        sendLocal(message);
        sendRemote(message);
        field.setText("");
    }

    // Client runs this to handle incoming messages
    // (our client only uses the inputStream of the connection)
    private class ClientHandler extends Thread {
        private String name;
        private int port;

        ClientHandler(String name, int port) {
            this.name = name;
            this.port = port;
        }

        // Connect to the server, loop getting messages
        public void run() {
            try {
                // make connection to the server name/port
                Socket toServer = new Socket(name, port);

                // get input stream to read from server and wrap in object input stream
                ObjectInputStream in = new ObjectInputStream(toServer.getInputStream());
                System.out.println("client: connected!");

                // we could do this if we wanted to write to server in addition
                // to reading
                // out = new ObjectOutputStream(toServer.getOutputStream());

                while (true) {
                    // Get the xml string, decode to a Message object.
                    // Blocks in readObject(), waiting for server to send something.
                    String xmlString = (String) in.readObject();
                    XMLDecoder decoder = new XMLDecoder(new ByteArrayInputStream(xmlString.getBytes()));
                    Message message = (Message) decoder.readObject();

                    System.out.println("client: read " + message);

                    invokeToGUI(message);
                }
            } catch (Exception ex) { // IOException and ClassNotFoundException
                ex.printStackTrace();
            }
            // Could null out client ptr.
            // Note that exception breaks out of the while loop,
            // thus ending the thread.
        }
    }

    // Given a message, puts that message in the local GUI.
    // Can be called by any thread.
    public void invokeToGUI(Message message) {
        final Message temp = message;
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                status.setText("Client receive");
                sendLocal(temp);
            }
        });
    }

```

```

    }

    // Sends a message to all of the outgoing streams.
    // Writing rarely blocks, so doing this on the swing thread is ok,
    // although could fork off a worker to do it.
    public synchronized void sendRemote(Message message) {
        status.setText("Server send");
        System.out.println("server: send " + message);

        // Convert the message object into an xml string.
        OutputStream memStream = new ByteArrayOutputStream();
        XMLEncoder encoder = new XMLEncoder(memStream);
        encoder.writeObject(message);
        encoder.close();
        String xmlString = memStream.toString();

        // Now write that xml string to all the clients.
        Iterator<ObjectOutputStream> it = outputs.iterator();
        while (it.hasNext()) {
            ObjectOutputStream out = it.next();
            try {
                out.writeObject(xmlString);
                out.flush();
            }
            catch (Exception ex) {
                ex.printStackTrace();
                it.remove();
                // Cute use of iterator and exceptions --
                // drop that socket from list if have probs with it
            }
        }
    }

    // Adds an object stream to the list of outputs
    // (this and sendToOutputs() are synchronzied to avoid conflicts)
    public synchronized void addOutput(ObjectOutputStream out) {
        outputs.add(out);
    }

    // Server thread accepts incoming client connections
    class ServerAcceptor extends Thread {
        private int port;
        ServerAcceptor(int port) {
            this.port = port;
        }

        public void run() {
            try {
                ServerSocket serverSocket = new ServerSocket(port);
                while (true) {
                    Socket toClient = null;
                    // this blocks, waiting for a Socket to the client
                    toClient = serverSocket.accept();
                    System.out.println("server: got client");

                    // Get an output stream to the client, and add it to
                    // the list of outputs
                    // (our server only uses the output stream of the connection)
                    addOutput(new ObjectOutputStream(toClient.getOutputStream()));
                }
            }
            catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }

    // Starts the sever acceptor to catch incoming client connections.
    // Wired to Server button.
    public void doServer() {

```

```

        status.setText("Start server");
        String result = JOptionPane.showInputDialog("Run server on port", "8001");
        if (result!=null) {
            System.out.println("server: start");
            serverAcceptor = new ServerAcceptor(Integer.parseInt(result.trim()));
            serverAcceptor.start();
        }
    }

    // Runs a client handler to connect to a server.
    // Wired to Client button.
    public void doClient() {
        status.setText("Start client");
        String result = JOptionPane.showInputDialog("Connect to host:port", "127.0.0.1:8001");
        if (result!=null) {
            String[] parts = result.split(":");
            System.out.println("client: start");
            clientHandler = new ClientHandler(parts[0].trim(), Integer.parseInt(parts[1].trim()));
            clientHandler.start();
        }
    }
}

```