CS 580 Client-Server Programming
Spring Semester, 2009
Doc 4 Streams, Parsing, Sockets
Feb 5, 2009

References

SDSU Java Library, http://www.eli.sdsu.edu/java-SDSU/docs/

Java On-line API http://java.sun.com/j2se/1.5.0/docs/api/index.html

Unix Network Programming, Stevens, 1990, Berkeley Sockets chapter 6.

TCP/IP Illustrated Vol 1, Stevens, 1994, chapter 20.

Internetworking with TCP/IP, BSD Socket Version Vol. 3, Comer, Stevens, Prentice-Hall, 1993

Java Network Programming, Harold 3'rd Ed. Chapter 4

Java Streams

InputStream & Bytes

Read just bytes

```
int available()
void close()
abstract  int read()
int read(byte[] b)
int read(byte[] b, int off, int len)
long skip(long n)

void mark(int readlimit)
boolean markSupported()
void reset()
```

```
byte[] intput = new byte[10];
for (int k = 0; k < input.length; k++) {
    int b = in.read();
    if (b == -1) break;
    input[k] = (byte) b;
}
```

Issue - byte verses int

read returns an int

casts to signed byte

    -128 to 127

Works fine if value is between 0 and 127

```
int shifted = b >= 0 ? b : 256 + b;
```

```
byte[] intput = new byte[10];
for (int k = 0; k < input.length; k++) {
    int b = in.read();
    if (b == -1) break;
    input[k] = (byte) b;
}
```

Issue - Performance

Reading one byte at a time is slow

```
int bytesRead = 0;
int bytesToRead=1024;
byte[ ] input = new byte[bytesToRead];
while (bytesRead < bytesToRead) {
    int readSize = in.read(input, bytesRead, bytesToRead - bytesRead);
    if (readSize = -1 ) break;
    bytesRead += readSize;
}
```

Issue - How far to read?

Normally don't know the size of a message

Some protocols allow multiple requests to be sent as same time

Issue - Mark

void mark(int readlimit)
boolean markSupported()
void reset()

Most streams don't support mark

Be careful

Peek (look ahead) is Useful

messages;block:1;sender:whitney;;

messages;block:1;;

When we read a ";" are we

- done with the message
- Just done with one segment

Don't know until we read next character

Would Be Nice

But you need "peek"

```
while (!atEndOfMessage(in)) {
    messageText += readUpto(";", in);
}
```

atEndOfMessage(stream)

returns true if next character in stream is ";"
Does not remove characters from the stream

readUpto(char, stream)

reads up through the next occurrence of character

Some Smalltalk ReadStream Methods

peek
upTo: aCharacter
upToAll: aCollection
through: aCharacter
throughAll: aCollection
next
next: anInteger

Ruby Streams

```
def send(text)
    connection = TCPSocket.new(@server, @port)
    connection.print(text)
    connection.flush
    answer = connection.gets("\n")
    connection.close
    answer
  end
```

PrintStream

"PrintStream is evil and network programmers should avoid it like the plague!"

Elliotte Harold

Readers & Writers

Java's streams do not handle unicode.

If protocol uses unicode use readers and writers.

Java's Data Streams

Read/Write binary

Do not use if protocol is text based

If protocol is binary DataStreams format may not be correct

Parsing

Some low level Java Parsing

```
"cat;man;ran".split(";");
```

Returns an array of String [“cat”, “man”, “ran”];

StringTokenizer

```
parts = new java.util.StringTokenizer("cat,man;ran;,fan", ",;");
while (parts.hasMoreElements())
    {
    System.out.println( parts.nextToken());
    }
```

Output

```
cat
man
ran
fan
```

java.util.Scanner

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

Output

```
1
 2
red
blue
```

Java UpToReader?

```
Socket connection = new Socket(server, port);
InputStream rawIn = connection.getInputStream();
UpToReader in = new UpToReader(
            new InputStreamReader(rawIn));
String answer = in.upTo(';');
```

sdsu.io.ChunkReader

```
read = new sdsu.io.ChunkReader("catEOMmatEOM", "EOM")
while (read.hasMoreElements() )
    {
    System.out.println( read.readChunk());
    }
```

Output

```
cat
mat
```

Subclass FilterInputStream

```
public class UpToInputStream extends FilterInputStream {
    public UpToInputStream(InputStream stream)
        { super(stream); }

    public byte[] upto(char end) throws IOException {
        int EOF = -1;
        ByteBuffer buffer = new ByteBuffer();
        int c;
        while (( c = super.read()) != EOF )  {
            buffer.append( (byte)c);
            if (c == end )
                break;
        }
        if (c == EOF & (buffer.isEmpty()))
            return new byte[0];

        return buffer.getBytes();
        }
    }
```

Issue - What if User's text contains ";"

password = trou;;ble	login;screenName:whitney;password:trou\;\;ble;;
text = duh;now what	transmitMessage:duh\;now what;;

You need to escape/unescape the ";"

UpTo has to know about escaped characters

Relax

Clear your mind

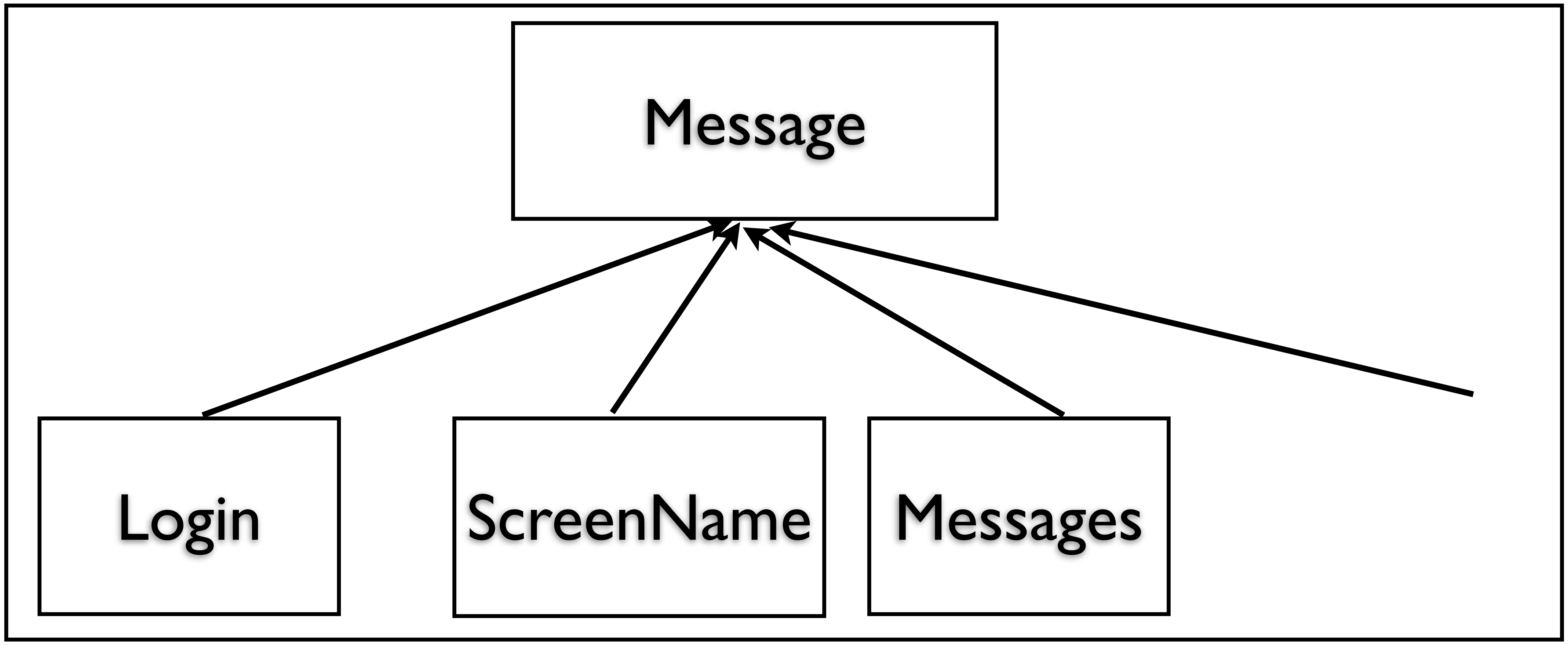
Get ready for big idea

Well maybe not so big, but it may take some getting used to

Why limit reading to characters?

Why not read Message Objects?

```
InputStream rawIn = connection.getInputStream();
SDwitterReader in = new SDwitterReader(rawIn);
Message answer = in.next();
```


Message
Login
ScreenName
Messages

Message Responsibilities

Hide all message syntax

Read message and convert to object

```
 TransmitMessage message =
TransmitMessage.from(“transmitMessage:duh\;now what;;”);
```

Create message from values

```
 TransmitMessage message = new TransmitMessage("duh;now what);
```

Convert object to required protocol string

```
message.toString()  // returns “transmitMessage:duh\;now what;;”
```

Access information about message

```
 message.isLogin();
 message.name();
```

Client Side

```
Socket connection = new Socket(server, port);
OutputStream rawOut = connection.getOutputStream();
PrintWriter out = new PrintWriter(new BufferedOutputStream(rawOut));
InputStream rawIn = connection.getInputStream();

SDwitterReader in = new SDwitterReader(rawIn);
SDwitterMessage login = new LoginMessage(“whitney”, "foo");
out.print(login.toString());
out.flush();

SDwitterMessage result = in.next();
if (result.isError() ) then
    deal with error
else
    blah
```

Server Side

```
SDwitterMessage request = in.next();
if (request.isLogin() ) {
            etc
}
else if (request.isTransmit() ) {
      etc
}
 blah
```

Consequences

Main code operates at higher level

Isolates protocol syntax

Testing becomes easier

More Classes

Logic is spread across multiple classes

Testing

Can test more parts without using network

```
public void testAdd() {
    AddMessage add = new AddMessage("cat");
    assertTrue( add.toString() == "add cat;";
    AddMessage fromString = new AddMessage.from(add.toString());
    assertTrue( fromString.name() == "cat");
}
```

Testing Servers

```
public class DateServer {

    public void run(int port) throws IOException {
        ServerSocket input = new ServerSocket( port );

        while (true) {
            Socket client = input.accept();
            BufferedReader parsedInput =
                new BufferedReader(new InputStreamReader(client.getInputStream()));

            boolean autoflushOn = true;
            PrintWriter parsedOutput = new PrintWriter(client.getOutputStream());

            String inputLine = parsedInput.readLine();

            if (inputLine.startsWith("date")) {
                Date now = new Date();
                parsedOutput.println(now.toString());
            client.close();
        }
    }
```

Testing DateServer

Must use network to test server

OK for date server, but not for more complex servers

Idea 1 - Keep Network Layer Thin

```
public class DateServer {
    private static Logger log = Logger.getLogger("dateLogger");

    public void run(int port) throws IOException {
        ServerSocket input = new ServerSocket( port );

        while (true) {
            Socket client = input.accept();
            log.info("Request from " + client.getInetAddress());
            processRequest(
                client.getInputStream(),
                client.getOutputStream());
            client.close();
        }
    }
```

```
void processRequest(InputStream in,OutputStream out)
        throws IOException {

        BufferedReader parsedInput =
                new BufferedReader(new InputStreamReader(in));

        boolean autoflushOn = true;
        PrintWriter parsedOutput = new PrintWriter(out,autoflushOn);
        etc.
        }
    }
}
```

Idea 1 - Keep Network Layer Thin

```
public class TestDateServer {
    public void testDate() {
        InputStream in = new ByteArrayInputStream("date;".getBytes()))));
        ByteArrayOutputStream fakeOut = new ByteArrayOutputStream();
        DateServer counter = new DateServer();
        counter.processRequestOn(in, fakeOut);
        assertTrue(fakeOut.toString() == "2006 02 14;")
    }
}
```

Idea 2 - Separate IO from Action

```
class SDwitterServer {
    boolean login(String name, String password) {
        code here
    }

    boolean transmit(String message) {
        code here
    }

    etc.
```

Now can test action without

going through protocol strings

Scale Changes Everything

As a Server grows in complexity testing through sockets/streams is too hard

Idea 3 Fake it

Create a fake Socket class that

- returns fixed output
- records input

Build class from scratch or use Mock Objects

Ruby FlexMock
http://onestepback.org/software/flexmock/

Mock Object Home
http://www.mockobjects.com/

Example of Mock Object

```
require 'flexmock'
require 'test/unit'

class TestExample < Test::Unit::TestCase
  def testShowMockObject()
    a = FlexMock.new
    a.should_receive(:foo).with(4).returns{|x| x + 1}
    a.should_receive(:foo).with(10).returns{'cat'}
    a.should_receive(:bar).returns{'dog'}
    assert( a.bar == 'dog')
    assert( a.foo(4) == 5)
    assert( a.foo(10) == 'cat')
    assert( a.foo(4) == 5)
    assert( a.bar == 'dog')
  end
end
```

Idea 4 - Run Client & Server in test case

```
require 'flexmock'
require 'test/unit'
require 'server'
require 'client'

class TestExample < Test::Unit::TestCase
  def setup()
    @server = Server.new(4444)
    @serverThread = Thread.new { @server.run }
  end

  def teardown()
    @serverThread.terminate
  end

  def testServer()
    client = Client.new("localhost", 4444)
    result = client.count("/foo")
    blah
  end
end
```

Look out for deadlock

Worry about scaling

Socket Options

Timeouts

Buffer Size

Multi-Homing

No Delay for small data

Linger on close

Keep-Alive

Urgent-Data

Timeouts

Socket will time out after specified time of inactivity

Java

Both Socket and ServerSocket class support:

```
void setSoTimeout(int timeoutInMilliseconds) throws SocketException
void getSoTimeout() throws SocketException
```

Must be sent before performing a read

Read throws SocketTimeoutException when socket times out

Not normally used on ServerSockets

Buffer Size

Each TCP socket has

- Receive buffer
- Send Buffer

Buffers are in the TCP stack space (not the VM)

Buffer size should:

- Be at least 16KB on Ethernet
- Applications that send lots of data use 48KB or 64KB

TCP does not allow the sender to overflow the receiver's buffer

So the receiver's receive buffer as large as the sender's send buffer

Buffers larger than 64KB require special set up

Java Example

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

public class ServerWithTimeout extends Thread {
    static final int CLIENT_TIMEOUT = 3 * 1000; // in milliseconds
    static final int BUFFER_SIZE = 16 * 1024;
    ServerSocket acceptor;

    public static void main(String[] args) throws IOException {
        int port = Integer.parseInt( args[1]);

        ServerWithTimeout server = new ServerWithTimeout( port );
        server.start();
    }

    public ServerWithTimeout(int port ) throws IOException {
        acceptor = new ServerSocket(port);
        acceptor.setReceiveBufferSize( BUFFER_SIZE );
    }
```

Java Example

```
public void run() {
    while (true)  {
        try {
            Socket client = acceptor.accept();
            processRequest( client );
        }
        catch (IOException acceptError){
            // for a later lecture
        }
    }
}
void processRequest( Socket  client) throws IOException {
    try {
        client.setReceiveBufferSize( BUFFER_SIZE);
        client.setSoTimeout( CLIENT_TIMEOUT);
        processRequest(
            client.getInputStream(),
            client.getOutputStream());
    }
    finally {
        client.close();
    }
```

Java Example

```
void processRequest(InputStream in,OutputStream out) throws IOException  {
    BufferedReader parsedInput = null;
    PrintWriter parsedOutput = null;
    try {
        parsedInput = new BufferedReader(new InputStreamReader(in));
        parsedOutput = new PrintWriter(out,true);

        String inputLine = parsedInput.readLine();

        if (inputLine.startsWith("date"))  {
            Date now = new Date();
            parsedOutput.println(now.toString());
        }
    }
    catch (SocketTimeoutException clientTooSlow) {
            parsedOutput.println("Connection timed out");
        }
    }
}
```

Nagle's Algorithm

Delays transmission of new TCP packets while any data remains unacknowledged

Allows TCP to merge data into larger packets before sending

Introduced to avoid lots of small packets across a WAN

Delay is on by default

```
class Socket
    {
    void setTcpNoDelay(Boolean noDelay) throws SocketException
    void getTcpNoDelay() throws SocketException
    }
```

Linger on Close

Determines what happens when a socket is closed

How long does the socket remain after being closed

- Acknowledge packets
- Retransmit lost packets

Default is to

- Allow the application to continue
- TCP handles sending unsent data & rejecting new requests

Keep Alive

Send packet on inactive connection to prevent timeouts

At least 2 hour delay between sending keep alive packets

Long delay limits it usefulness

Urgent (Out of Band) Data

Urgent data can be read out of order

Read before data that was sent before it

Java

Supports sending of urgent data

Does not promote urgent data in the input stream