

**import java.net.\*;**

**java.net.\*;**

- Classes supporting TCP/IP based client-server connections.
- (You will meet TCP/IP from C/C++ perspective in CSC1214; inner workings of TCP/IP covered in many SECTE subjects)
- In days of “*Java and the Internet*”, the “... *and the Internet*” component had introductory coverage of TCP/IP.

You must have had something on TCP/IP in CSC110; it cannot be that the only coverage of networking is in elective subjects.

## IP (Internet Protocol)

- Each machine has one (or more) IP addresses
- IP protocol defines how to get data packets across the Internet from one machine to another.
- But need
  - Client program communicates with server program
  - Not “*client machine communicates with server machine*”
- Other protocols layered over IP allow program to program communications
- TCP & UDP

## Program-to-program communications over IP

- How to identify end-point program?
- “Port”
  - Portal, a door or gateway
- Ports are OS structures
  - Data buffer for input/data buffer for output
  - Identified by a number
- Programs ask OS for use of a port
- Server program asks for specific port number, publishes its port number; client program uses this to identify server
- OS allocates port number for client

## UDP & TCP

- UDP
  - Client composes one data request packet
  - Client sends it to server (identified by combination of IP address and port number; includes its own IP address and port number in packet header)
  - Server responds with one packet (using client address from header)
  - Connection terminated
- TCP
  - TCP uses IP and ports in same way as UDP
  - TCP libraries hide “packet” nature of communications
    - Client and server see connection as continuously open input-output stream, both can read and write data
  - TCP library code deals with all the hard bits of arranging for data transmission in packets, guaranteed delivery, flow-control etc

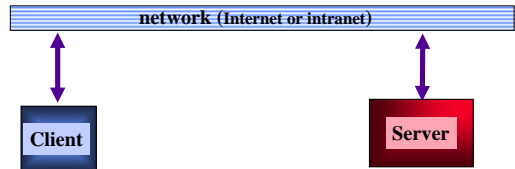
## Ports & sockets

- Ports belong to OS
- Programs need access to the I/O buffers on port
- “Sockets”
  - I/O streams working with buffers of given port

## Sockets

- Actually two kinds
  - Data stream sockets
    - Read and write application data
  - Server sockets
    - Used when client wants to make connection to server
      - Client details put in server-socket input buffer
      - Additional I/O buffers allocated for data transfers with this client
      - Server program given all these details

## clients and servers



Java Applet in Web page  
Java application  
C++ or other application

Java server  
C++ server

*No need for client and server to be implemented in same language – they can often communicate by text messages.*

## Clients and Servers

- Java Applets are restricted to servers running on same machine as Webserver program that provided HTML page with Applet
- Server program must be running (listening at “well known port”)
  - official servers (like ftpd, httpd, ...) are organised by OS on server machine
  - local standards are also handled automatically via “inetd” system (system administrator defines list of extra server programs run at this site and their ports, if OS receives request at port it starts corresponding server program; on Windows, programs are registered with OS as “services” and get started automatically)
  - private server programs must be launched by owner
- Client program launched by user
  - Familiar examples – browser client, ftp-client

## Clients

- Code for handling client side is completely standard, the code examples in books on TCP/IP programming in C/C++ can easily be adapted for Java.

```

client
  build address structure for server
  get socket
  connect to server
  while not finished
    get command from user
    write to socket
    read reply
    show reply to user
  
```

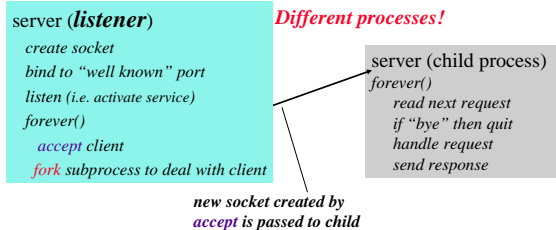
*Obviously, this could (& should) be handled by a dedicated thread if you also want to maintain GUI interaction*

## Servers

- Server may have to deal with many clients
- Concurrency strategies:
  - Serial:
    - deal with clients one at a time, handling all requests until client disconnects; then start handling next client.
    - OS helps by maintaining a queue of clients who wish to connect;
  - Forking
    - Start a new process for each client – just as was described for WebServer (this strategy rarely used with Java servers)
  - Threaded
    - Each client gets a thread to serve them

## Servers

- The normal server (as described in TCP/IP texts) is a **forking server**:

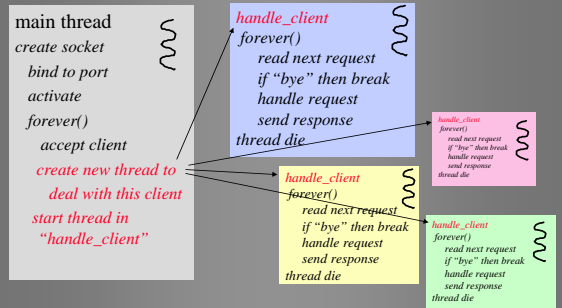


## Servers

- Forking server traditionally favoured on Unix systems
  - lots of system support
  - relatively easy to program
  - appropriate when clients are independent (don't desire to interact)
- Alternative**
  - a server that supports multiple concurrent clients, usually via a threads mechanism

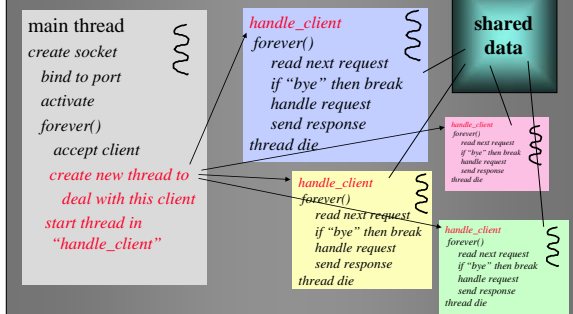
## Threaded server

### All the one process (independent clients)



## Threaded server

### All the one process (interacting clients)



## Servers

- Java servers are rarely (if ever) forking servers
  - though can launch a process via Runtime object
- Java's support for threads makes it simpler than usual to implement as threaded server
  - all objects have associated locks, simply need to start to use these (facilitating safe use of shared data)
  - thread package provides simple subset with most useful features from a full threads implementation
  - ...
- Java solution: have a "ClientHandler" class (optionally inherits Thread), create an instance for each client (passing new socket), let it run()

## Your programming of Client - server systems

- You can rely on the libraries to handle:
  - establishment of communications link
  - reliable data exchange (TCP/IP) (or the rarely useful simple packet transfer based on cheaper UDP/IP mechanisms)
- What you really see at both ends is a connection that supports read and write access.
- You must invent



## Server provides a service

- Typically, a server is a bit like an instance of a class that has a public interface advertising a few operations it can perform for a client.
  - Example 1: HTTP server
    - GET, POST, OPTIONS, ...
  - Example 2: FTP server
    - List directory, change directory, get file, put file, ...
- Sometimes will have a service with only one operation – echo, ping, ...

## Client – server application protocol

- Each operation defined by server has name, and arguments and returns particular kind of data (or “an exception”)
- Client must supply data identifying operation required, and supply and data needed.
- Protocol specifies how client to supply such information and what kinds of responses it may receive

## Your programming of Client - server systems

- Decide on the different commands and data that you want to exchange. List for each command (request) :
  - data that may be sent with the command
  - allowed responses (“success”, “failure”, other eg “defer”) and note appropriate action
  - data that may come with different responses
- Associate suitable key words with each command, and key word (or numeric codes) that could identify possible responses.

## Your programming of Client - server systems

- Client (*things that you should think about*)
  - how does user select the server?
  - connection mechanism set up; how to handle failures?
  - how does user select the commands to be sent, and how are necessary input data obtained?
  - sending data;
  - getting response
    - switch(response code) into ...
      - handle various permitted responses appropriate for last request
  - displaying response;
  - how does user terminate session?

## Your programming of Client - server systems

- Server
  - First, what strategy?
    - Depends on load that you expect on your server, if lightly used a serial server may suffice (and is a lot easier to implement)
  - If expect to deal with many clients, then need concurrent (threaded) server
    - Much of code is standardized, so can cut-copy-paste

## Your programming of Client - server systems

- Server (*things that you should think about*)
  - setting up listener mechanism, accepting new client and getting socket, creating thread for new client – *these aspects should all be standard*
  - “handle\_client”
    - reading request; *dealing with the unexpected, excessive data, ...*
    - switch(keyword) into ...
      - invoke separate functions for each of the commands that this server handles
    - generation of response
  - how does it terminate?

## Your programming of Client - server systems

- Handle-client
  - If serial server architecture,
    - call a function that will accept commands and generate responses until user disconnects
  - If threaded architecture
    - create a runnable ClientHandler object
    - create a thread that runs this ClientHandler
    - let multiple clients run with own threads



## Java support for networking

- java.net package includes a variety of classes that are essentially “wrappers” for the simple to use but rather messy Berkeley sockets networking facilities.
  - classes for the **sockets**
  - classes that represent **network addresses** for server machines (hosts) and clients
  - classes that handle URLs
  - class for talking http protocol to ‘httpd’ server on some host
  - ...

## network classes

- **Socket**
  - client side connection to network
    - connect(...)
    - provides InputStream and OutputStream (byte oriented; re-package inside a DataInputStream etc)
    - status info ...
- **ServerSocket**
  - server side connection to network
    - accept(...) (gives back new Socket for use in handle\_client())
    - status info, ...

## network classes

- **InetAddress**
  - “data structure” representing an address (a “final” class, basically it’s an opaque box holding system dependent information needed by Socket classes)
  - create instances using static member functions of class
    - static InetAddress.getByName(String host) ...
      - can ask for an Internet address for host (specified in dotted decimal or domain name form)
    - static InetAddress.getLocalHost() (own address)
  - *not really needed as can create Sockets without needing InetAddress structures, but can be useful as functions that create InetAddress objects can return more informative exceptions if something wrong (whereas exception with Socket simply tells you connection wasn’t possible)*



## Example

- **Program**
  - takes hostname and port number as command line arguments
  - attempts to build InetAddress for host
  - attempts to open Socket to host/port combination
  - creates DataStream adaptors for Socket
  - loop
    - read line of input
    - if “Quit” then stop
    - send line (and ‘\n’) to server
    - read bytes returned
    - convert to String and display

*A crude substitute for ‘telnet’ (port 23) – talk to ‘echo’, etc.*

## Example – telnet client

- Client reads in a line, making it a Java String with 16-bit characters
- telnet (or other standard service) expects ASCII text for communications
- Client cannot simply write the String to socket (which it could if talking Java-to-Java)
- Instead must write as 8-bit characters
- Can use “DataInputStream” and “DataOutputStream” to handle this

## Basic structure ...

- Essentially as shown before ...

```

client
  build address structure for server
  get socket - connect to server  (one step process with Java classes)
  while not finished
    get command from user
    write to socket
    read reply
    show reply to user
  
```

```

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

public class NetConnect {
    public static void main(String argv[])
    {
        check input arguments
        try to use first as a hostname when creating an InetAddress
        try to use second as a port number
        create socket for InetAddress, port combination
        create streams for socket

        promote System.in to BufferedReader
        loop
            prompt for input; read line, trim,
            if Quit then break
            write as bytes to socket
            loop reading bytes (until get '\n')
            print data read
    }
}

```

*“Telnet” client  
substitute*

```

public static void main(String argv[])
{
    if(argv.length != 2) {
        System.out.println("Invoke with hostname and port number arguments");
        System.exit(0);
    }

    InetAddress ina = null;
    try { ina = InetAddress.getByName(argv[0]); }
    catch (UnknownHostException uhne) {
        System.out.println("Couldn't interpret first argument as host name");
        System.exit(0);
    }

    int port = 0;
    try { String ps = argv[1].trim(); port = Integer.parseInt(ps); }
    catch (NumberFormatException nfe) {
        System.out.println("Couldn't interpret second argument as port number");
        System.exit(0);
    }
}

```

```

public static void main(String argv[])
{
    ...
    Socket s = null;
    try { s = new Socket(ina, port); }
    catch (IOException io) {
        System.out.println("No luck with that combination; try another host/port");
        System.exit(0);
    }

    DataOutputStream writeToSocket = null;
    DataInputStream readFromSocket = null;
    try {
        writeToSocket = new DataOutputStream(s.getOutputStream());
        readFromSocket = new DataInputStream(s.getInputStream());
    }
    catch (IOException io) {
        System.out.println("Got socket, but couldn't set up read/write communications");
        System.exit(0);
    }
}

```

```

public static void main(String argv[])
{
    ...
    BufferedReader d = new BufferedReader(new InputStreamReader(System.in));
    byte[] inputbuffer = new byte[2048];
    for(;;) {
        System.out.print(">");
        String str = null;
        try { str = d.readLine(); }
        catch (IOException io) {System.exit(0); }
        str = str.trim();
        if(str.equals("Quit")) break;
        try { writeToSocket.writeBytes(str + "\n"); }
        catch (IOException io) {
            System.out.println("Write to socket failed"); System.exit(0);
        }
    }
}

```

```

public static void main(String argv[])
{
    ...
    for(;;) {
        ...
        int numread = 0;
        for(numread=0; numread++) {
            byte b = 0;
            try { b = readFromSocket.readByte(); }
            catch (IOException io) {
                System.out.println("Read from socket failed"); System.exit(0);
            }
            if(b == '\n') break;
        }

        String response = new String(inputbuffer, 0, numread);
        System.out.println(response);
    }
}

```

*could have simply printed data from buffer  
but more typically would want String*

## Reading and writing @ sockets

- If talking Java to Java, it is your choice
  - promote socket connections to `BufferedReader` & `BufferedWriter` (send Unicode characters across network)
  - promote socket connections to `ObjectInputStream` and `ObjectOutputStream` (send objects across the network –this is often the best)
  - promote socket connections to `DataInputStream` and `DataOutputStream`, send Java doubles, UTF-8 data etc across web
- If talking Java to other
  - promote socket connections to `DataInputStream` and `DataOutputStream`, send data as text, writing and reading using byte transfers (don't want to confuse other programs with Unicode!)

## Example's reads and writes

- This client meant for general use, not Java-to-Java, so use `DataInputStream`, `DataOutputStream`, and text written and read as bytes.
- Most “telnet” like services are “line-oriented”
  - read one line up to and including `\n`
  - respond with single line (if multi-line response, then some agreed terminator, eg line with just `.'`)
- Hence code to make sure `\n` sent and loop reading characters until get `\n` (could have used `DataInputStream.readLine()` but it is deprecated)

## Example – simple client

- This simple client works, eg connect to port 7 of typical Unix server and you are talking to “echo” program
  - each line you enter will be echoed by other machine

## Examples

## Examples

- Simple serial server
  - `getDate` etc
- Threaded version of same
- Internet Relay Chat program



## NetBeans and examples

- NetBeans environment helps develop and test single program,
- Now have “client” and “server”
- If developing in NetBeans
  - Use separate projects (possibly sharing files)
  - Build client, build server; get `.jar` files with applications
  - Run at `cmd.exe` level so can start server application then start client application from separate `cmd.exe` shell

## Remote service (simple serial server)

## Service operations

- getDate
  - returns date
- getFortune
  - returns a random fortune cookie string chosen from a small collection
- quit
  - client politely says goodbye

## Protocol

- Easy!
  - getDate
    - client sends string Date
    - client receives string with Date-time on server
  - getFortune
    - client sends string Fortune
    - client receives string chosen from servers collection
  - quit
    - client sends string Quit
    - server does not respond, it just closes socket

## Communications

- In this example, use ObjectStreams
  - overkill!
  - good practice for real examples

- ObjectStreams – preferred approach for Java to Java

## Serial server

- Set up server socket
- Loop forever
  - wait for client to connect (blocking accept call)
  - get socket connecting to client
  - open streams for socket
  - loop
    - read command
    - if command was getDate send back date-time
    - else if command was getFortune send back fortune
  - until command was quit
  - close client socket

## Client code

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

public class Client
{
    public static void main(String[] args)
    {
        // Procedural client, just one mainline for
        // something this simple!
        ...
    }
}
```

```
public static void main(String[] args)
{
    if(args.length != 2) {
        Basic setup – get host & port
        System.out.println(
            "Invoke with hostname and port number arguments");
        System.exit(1);
    }
    String hostName = args[0];
    int port = 0;
    try {
        String ps = args[1].trim();
        port = Integer.parseInt(ps);
    }
    catch (NumberFormatException nfe) {
        System.out.println(
            "Couldn't interpret second argument as port number");
        System.exit(1);
    }
}
```

```

public static void main(String[] args)
{
    ...
    // Create client socket
    Socket sock = null;
    try {
        sock = new Socket(hostName, port);
    }
    catch(Exception e) {
        // I/O, unknown host, ...
        System.out.println("Failed to connect because "
            + e.getMessage());
        System.exit(1);
    }
}

```

```

// Get I/O streams, make the ObjectStreams
// for serializable objects
ObjectInputStream responseStream = null;
ObjectOutputStream requestStream = null;
try {
    requestStream = new ObjectOutputStream(
        sock.getOutputStream());
    requestStream.flush();
    responseStream = new ObjectInputStream(
        sock.getInputStream());
}
catch(IOException ioel) {
    System.out.println("Failed to get socket streams");
    System.exit(1);
}
System.out.println("Connected!");

```

**Output first, flush it, then input – else system may stall; feature!**

```

// main loop, user commands
BufferedReader input = new BufferedReader(
    new InputStreamReader(System.in));
for(;;) {
    System.out.print(">");
    String line;
    try {
        ... // see next two slides
    }
    catch(IOException ioe2) {
        System.out.println("Problems!");
        System.out.println(ioe2.getMessage());
        System.exit(1);
    }
    catch(ClassNotFoundException cnfe) {
        System.out.println("Got something strange back from
server causing class not found exception!");
        System.exit(1);
    }
}
}

```

```

line = input.readLine();
if(line==null) { System.exit(0); }
if(line.equals("Fortune")) {
    requestStream.writeObject(line);
    requestStream.flush();
    requestStream.reset();
    String cookie = (String) responseStream.readObject();
    System.out.println(cookie);
}
else
if(line.equals("Date")) {
    requestStream.writeObject(line);
    requestStream.flush();
    requestStream.reset();
    String date = (String) responseStream.readObject();
    System.out.println(date);
}
else
...

```

```

...
else
if(line.equals("Quit")) {
// Disconnect, don't expect response to "Quit"
requestStream.writeObject(line);
requestStream.flush();
requestStream.reset();
requestStream.close();
responseStream.close();
break;
}
// ignore any other input commands -
// they are invalid

```

## Sending and receiving

- Write the object
- Flush
  - OS may not send small message!
    - It waits assuming that there will be more to send and better to send one big package than two small ones
  - So force sending
- Reset
  - Remember those dictionaries associated with objectstreams that remember every object written or read?
  - Clear by explicitly “resetting”
  - A reset by the sender causes a reset on receiver

## ObjectStream & Strings

- Strings are objects!
- Class String implements Serializable
- Can use Strings with ObjectInputStream and ObjectOutputStream

## Simple serial server

## Usual structure

- Create server socket bound to chosen port and prepare to accept client connections
- Loop
  - blocking wait for client to connect
  - get new datastream socket when client does connect
  - invoke “handle client” function to process requests
  - close datastream socket when client leaves

## Chosen port number?

- Don't use numbers < 1024 ever (*unless your program would be started by “root” on Unix – i.e. it is some major server like httpd, ftpd, ...*)
- Avoid numbers < 10,000
  - 1K...5K many claimed by things like Oracle
  - 5K...10K used for things like X-terminals on Unix
  - 10K..65000 ok usually
    - Some ports will be claimed, e.g. DB2 will grab 50,000
    - If request to get port at specified number fails, pick a different number!

```
import java.io.*;
import java.net.*;
import java.util.*;
public class Server
{
    private static final int DEFAULT_PORT = 54321;
    private static Random rgen =
        new Random(System.currentTimeMillis());
    private static String getFortune() { ... }
    private static void handleClient(Socket sock) { ... }
    public static void main(String[] args)
    {
        int port = DEFAULT_PORT;
        if(args.length==1) {
            try { port = Integer.parseInt(args[0]); }
            catch (NumberFormatException ne) { }
        }
        ServerSocket reception_socket = null;
        try {
            reception_socket = new ServerSocket(port);
        }
        catch(IOException ioel) { ... }
        for(;;) { ... }
    }
}
```

```
public static void main(String[] args) {
    int port = DEFAULT_PORT;
    if(args.length==1) {
        try { port = Integer.parseInt(args[0]); }
        catch (NumberFormatException ne) { }
    }
    ServerSocket reception_socket = null;
    try {
        reception_socket = new ServerSocket(port);
    }
    catch(IOException ioel) { ... }
    for(;;) {
        Socket client_socket=null;
        try {
            client_socket =
                reception_socket.accept();
        }
        catch(IOException oops) { ... }
        handleClient(client_socket);
    }
}
```

Simple serial server!

```

private static void handleClient(Socket sock) {
    ObjectInputStream requests = null;
    ObjectOutputStream responses = null;
    try {
        responses = new ObjectOutputStream(
            sock.getOutputStream());
        responses.flush();
        requests = new ObjectInputStream(
            sock.getInputStream());
    }
    catch(IOException ioel) {
        System.out.println("Couldn't open streams");
        try { sock.close(); } catch(Exception e) {}
        return;
    }
    for(;;) {
        ...
    }
}

```

Setting up of communications streams for current client

```

for(;;) {
    try {
        String request = (String) requests.readObject();
        if(request.equals("Date")) {
            responses.writeObject(new Date().toString());
            responses.flush();
            responses.reset();
        }
        else
            if(request.equals("Fortune")) {
                responses.writeObject(getFortune());
                responses.flush();
                responses.reset();
            }
        else
            if(request.equals("Quit")) {
                try { sock.close(); } catch(Exception e) {}
                break;
            }
    }
    catch(Exception e) {
        try { sock.close(); } catch(Exception eclose) {}
        return;
    }
}

```

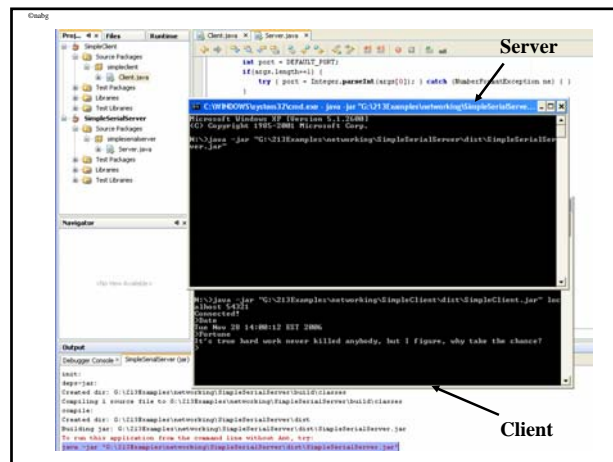
“Dispatching” remote requests to service functions

```

private static String getFortune() {
    String[] cookies = {
        "The trouble with some women is they get all excited about nothing, and then they marry him.",
        "It's true hard work never killed anybody, but I figure, why take the chance?",
        "A successful man is one who makes more money than his wife can spend.",
        "When a man brings his wife flowers for no reason, there's a reason.",
        "Genius is the ability to reduce the complicated to the simple."
    };

    int which = rgen.nextInt(cookies.length);
    return cookies[which];
}

```



## Multiple machines

- If practical, run client and server on different machines
  - Copy client .jar file to second machine
  - Start server on one machine (with known IP address or known DNS name)
  - Start client on second machine
    - Command line arguments are server DNS name (or IP address) and port

What about errors on server side?

## Errors

- First server was fault free – nothing can go wrong except for a break in communications.
- Usually client submits data as well as request identifier – and submitted data may cause problems on server.
- Modify our service to illustrate this
  - getFortune(String choice)
  - client specifies which fortune by supplying a string
    - string should represent integer
    - integer should be in range
- Bad data results in server side error that must be returned to client

## ... and the communications

- Client now sending a composite message
  - control (keyword string identifying command)
  - other data (some form of serializable data, or maybe null if command doesn't take data)
- Worth introducing a Message structure for client to send data

```
public class Message implements Serializable {
    public String control;
    public Serializable associateddata;
    public Message(String cntrl, Serializable data)
    {
        control = cntrl;
        associateddata = data;
    }
}
```

## Server needs mechanism to return errors to client

- At least two approaches:
  - Use the same message structure
    - control word indicates success or failure
    - associated data is the response or supplementary information about the failure
  - Create an Exception Object on the server
    - do not "throw" it on the server!
    - send it back (Exceptions are serializable and go through object streams)
    - client receives object, determines whether or not is exception

## Using messages first

- Client – modify code in command loop
- ```
line = input.readLine();
if(line==null) { System.exit(0); }
if(line.equals("Fortune")) {
    System.out.print("Which one : ");
    String choice = input.readLine();
    requestStream.writeObject(new Message(line, choice));
    requestStream.flush();
    requestStream.reset();
    Message response = (Message)
        responseStream.readObject();
    if(response.control.equals("OK"))
        System.out.println(response.associateddata);
    else
        System.out.println(response.associateddata);
}
```

## Server

- Fortune function

```
private static String getFortune(int choice)
{
    String[] cookies = {
        ...
        "It's true hard work never killed anybody, but I ...",
        ...
    };
    return cookies[choice];
}
```

```
Message request = (Message) requests.readObject();
if(request.control.equals("Fortune")) {
    int choice;
    try {
        choice = Integer.parseInt(
            (String) request.associateddata);
        String fortune = getFortune(choice);
        responses.writeObject(new Message("OK", fortune));
        responses.flush(); responses.reset();
    }
    catch(NumberFormatException nfe) {
        responses.writeObject(new Message("Error",
            "- non numeric data for choice"));
        responses.flush(); responses.reset();
    }
    catch(IndexOutOfBoundsException ndxe) {
        responses.writeObject(new Message("Error",
            "IndexOutOfBounds!"));
        responses.flush(); responses.reset();
    }
}
```

## Other version – with exceptions

- Client still sends its requests using those Message structs
- Response object is what?
  - Know it will be a Serializable, but what?
  - String
    - print it
  - Exception
    - throw it
- Use “instanceof” to find if response object is Exception

## Client

```
if(line.equals("Fortune")) {
    System.out.print("Which one : ");
    String choice = input.readLine();
    requestStream.writeObject(new Message(line, choice));
    requestStream.flush();
    requestStream.reset();
    Serializable response =
        (Serializable) responseStream.readObject();
    if(response instanceof Exception)
        throw ((Exception) response);
    System.out.println(response); // know it is a string
}
```

## Client – exception handling

```
catch(IOException ioe2) {
    System.out.println("Problems!");
    System.out.println(ioe2.getMessage());
    System.exit(1);
}
catch(ClassNotFoundException cnfe) {
    System.out.println("Got something strange back from server
    causing class not found exception!");
    System.exit(1);
}
catch(Exception other) {
    System.out.println(
        "Server probably returned this exception!");
    System.out.println(other);
}
```

## Server

```
if(request.control.equals("Fortune")) {
    int choice;
    try {
        choice = Integer.parseInt(
            (String) request.associateddata);
        String fortune = getFortune(choice);
        responses.writeObject(fortune);
        responses.flush(); responses.reset();
    }
    catch(NumberFormatException nfe) {
        responses.writeObject(nfe);
        responses.flush(); responses.reset();
    }
    catch(IndexOutOfBoundsException ndxe) {
        responses.writeObject(ndxe);
        responses.flush(); responses.reset();
    }
}
```

## Threaded version?

!

## Concurrent server

- No change to client
- Server:
  - Use application specific client handler class that implements Runnable
    - its public void run() method has the code from the handle\_client function (plus any auxiliary functions)
  - After accepting client
    - Create new ClientHandler with reference to Socket
    - Create thread
    - Thread runs handler

Add another operation to demonstrate that each client has separate handler!

- Ping
- Client sends Ping as message
- Server (ClientHandler) responds with
  - “Pong” plus a count
  - count value is incremented each time
  - count is instance variable of ClientHandler so each client has distinct count value

## Client Ping code

```
if(line.equals("Ping")) {
    requestStream.writeObject(
        new Message(line, null));
    requestStream.flush();
    requestStream.reset();
    String response =
        (String) responseStream.readObject();
    System.out.println(response);
}
```

## Server

```
public class Server
{
    private static final int DEFAULT_PORT = 54321;
    public static void main(String[] args)
    {
        ...
        for(;;) {
            try {
                Socket client_socket =
                    reception_socket.accept();
                ClientHandler ch =
                    new ClientHandler(client_socket);
                Thread t = new Thread(ch);
                t.start();
            }
            ...
        }
    }
}
```

## Java 1.5 addition

- Example code creates a new thread for each client and destroys the thread when finished.
- Costly.
- Well threads aren't as costly as separate processes but they aren't cheap either.
- The Java 1.5 concurrency utilities package has things like `Executor.newFixedThreadPool` (which gives you a pool of threads); then give `Runnable` to this pool, a thread will be allocated.

## ClientHandler

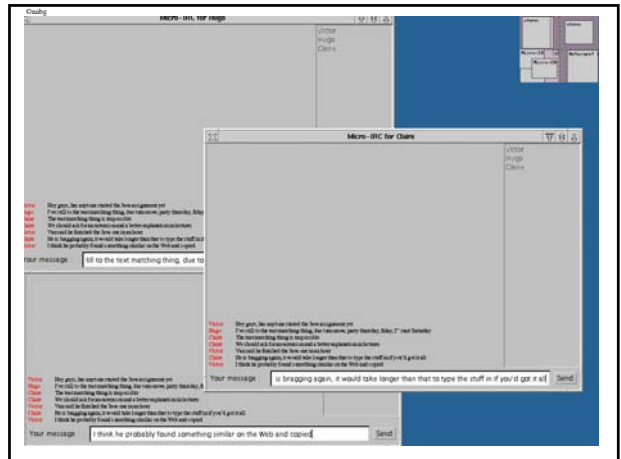
```
public class ClientHandler implements Runnable
{
    private int count;
    private ObjectInputStream requests;
    private ObjectOutputStream responses;
    private String getFortune(int choice) { ... }
    public ClientHandler(Socket sock) {
        // initialize, open streams etc
        ...
    }
    public void run()
    {
        for(;;) { ... }
    }
}
```

## ClientHandler.run()

```
Message request = (Message) requests.readObject();
if(request.control.equals("Date")) { ... }
else
if(request.control.equals("Fortune")) { ... }
else
if(request.control.equals("Ping")) {
    responses.writeObject("Pong " + Integer.toString(++count));
    responses.flush();
    responses.reset();
}
else
if(request.control.equals("Quit")) { ... }
```

# Micro IRC

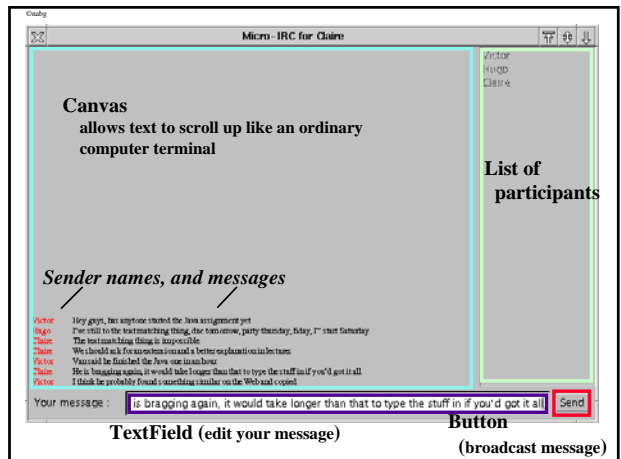
Internet relay chat



- ## IRC?
- Internet relay chat:
    - host machine runs server program
      - program supports multiple concurrent clients (via threading or polling mechanisms) accessing shared data
    - server program provides
      - “virtual channels”
        - each channel related to a discussion topic
        - when first connect to server, can review choices, and pick channel
      - discussion groups on channels
        - associated with each channel have
          - group of clients (identified by nicknames)
          - buffer containing last few messages exchanged

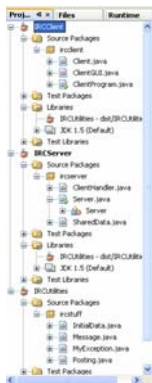
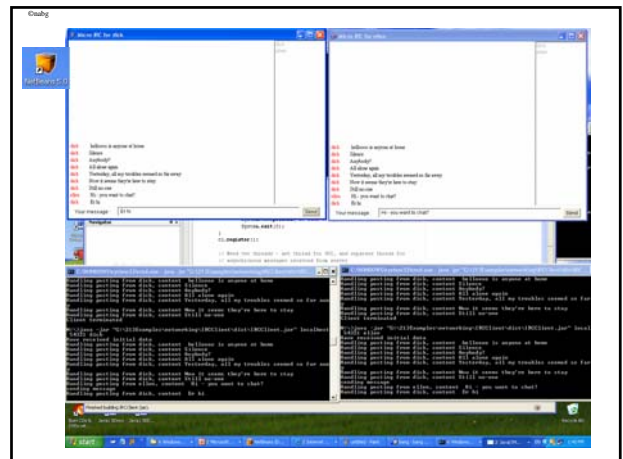
- ## IRC
- Client program
    - connect to server
      - specify hostname and port and nickname
    - view “channels”, submit selection
    - view list of participants, get display of recent messages
    - own messages broadcast to other participants
    - see any messages entered by others

- ## Micro-IRC
- Misses out the channels layer.
  - MicroIRC server
    - multithreaded Java application
    - buffer for 10 recent messages
  - MicroIRC client
    - also multithreaded (listening for messages from others, responding to local users keys/mouse actions)
    - List display of names of participants
    - Canvas (fake scrolling) of recent text (no local buffer)
    - TextField and action Button for message entry



## MicoIRC NetBeans style

- Three projects
  - Server application
  - Client application
  - Utilities library
- *(In /share/cs-pub/csci213 as compressed collection of NetBeans projects)*

## Structure

- Server
  - Threaded (using thread pool)
  - ClientHandlers created for each client
  - SharedData – keeps track of connected clients and last few messages posted
- Client
  - Simple GUI with text field to post data
  - Separate client thread to handle asynchronous messages from server

## Overall operation

- Server started
- Client connects
- Client attempts to “register”
  - Provides nickname; registration attempt will fail if nickname already in use
  - If registration succeeds, client receives in return a set of strings with nicknames for all connected clients and a copy of the last few messages posted.
- Client can now post messages
  - Messages posted to the server are distributed to all connected clients

## Distribution of postings by server

- Two possible strategies
  - Thread in ClientHandler that receives message is used to run code that forwards it to each client
  - Separate thread on server
    - Forever
      - Sleep a bit
      - Check if any new messages in buffer
        - Post each one to each of clients
- This implementation uses first strategy

## Communications

- TCP/IP socket streams wrapped in ObjectOutputStream and ObjectOutputStream
- Serializable objects written and read

## Classes - utilities

- “Utilities” library
  - Just a collection of little serializable structs
    - A “posting” has an author and a message
    - The initial data received by a client should contain a String[] and an array of postings
    - A message has an “operation” and some other data
    - A MyException is a specialization of Exception



## Classes - server

- Server, the usual
  - Server – procedural main-line, cut-&-paste code from any other threaded server
  - ClientHandler
    - A Runnable
  - SharedData
    - Owns collections of client handlers and recent postings



## Classes - client

- ClientProgram – procedural, deal with command line arguments, create client object (which creates its GUI)
- Client
  - Runnable
- ClientGUI



## Client

- AWT thread
  - Handle input fields
  - Used for actionPerformed that handles send button
  - Writes to ObjectOutputStream
- Created thread in Client.run()
  - Forever
    - Blocking read on ObjectInputStream
    - Read a Message object
      - Add\_Client
      - Remove\_Client
      - Post

## Library project

- Classes like Posting used in both Client and Server applications
- Create a “Java library” project (“IRCUtilities”)
- Define the classes
- Build the project
- Create Client and Server Application projects
  - In Library tab of project
    - Add library/Project/IRCUtilities

```

public class Message implements Serializable {
    public String operation;
    public Serializable otherdata;
    public Message(String op, Serializable data) {
        operation = op;
        otherdata = data;
    }
}

public class Posting implements Serializable {
    public String author;
    public String comment;
    public Posting(String person, String txt) {
        author = person;
        comment = txt;
    }
}
    
```

“otherdata” in Message will be a Posting or an InitialData object

```

public class InitialData implements Serializable {
    public String[] playerNames;
    public Posting[] postings;
    public InitialData(String[] names, Posting[] posts) {
        playerNames = names;
        postings = posts;
    }
}

```

```

public class MyException extends Exception {
    public MyException(String reason) {
        super(reason);
    }
}

```

## Micro-IRC Application protocol

- Client initiated exchanges
  - **Register**
    - client sends Message with “Register” and a nickname
    - server sends either
      - Message with “OK” and an InitialData object with details of all current players
      - MyException
  - if client gets OK, then continue; else terminate

*determine (main aspects of) application protocol*

## Micro-IRC Application protocol

- Client initiated exchanges
  - **Post**
    - client sends Message with “Post” and contents of text input buffer
    - not acknowledged
  - **Quit**
    - client sends “Quit:” (not acknowledged)

*determine (main aspects of) application protocol*

## Micro-IRC Application protocol

- Server initiated
  - **Post**
    - server sends Message “Post” together with a Posting object containing the nickname of participant who posted message and contents of message
    - no acknowledgement from client
  - **Add\_Client:**
    - another client has connected
  - **Remove\_Client:**
    - some other client has disconnected

*determine (main aspects of) application protocol*

## MicroIRC Client

- *Selection of server and establishment of connection*
  - use command line arguments
    - host\_name, port number, nickname to use when connect
  - construct InetAddress with given host\_name, if fail print error report and quit
  - socket connection opened to server, again failures result in program exiting.

*decide on general mechanisms for client, (but not their implementation)*

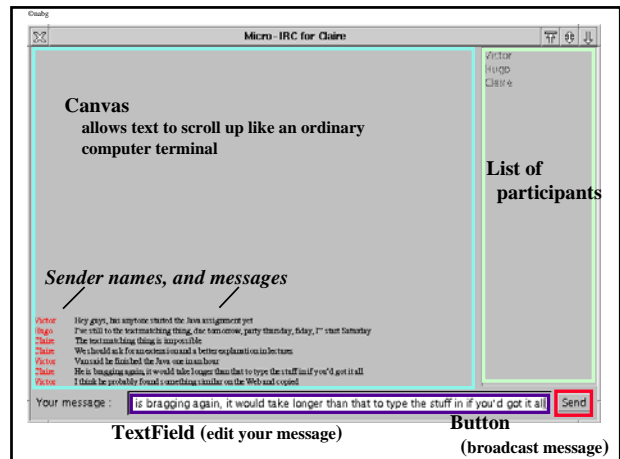
## MicroIRC Client

- Most messages from server are **asynchronous!**
  - server sends client a Post whenever any of users on system has entered data, and sends “add/remove client” as others connect and disconnect
  - so not synchronized with actions of local user
  - implication => *Need a separate thread ready at all times to handle asynchronous message from server*

## MicroIRC Client

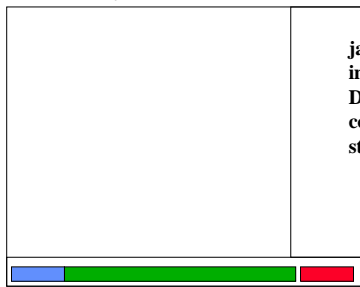
- *displaying response*
  - several possible ways of handling messages
    - TextArea (disabled) in scrolling window
      - allows user to scroll back, refresh is possible after display temporarily obscured
      - potentially large storage demands as add Strings to TextArea
    - array of Strings displayed in window
      - discard oldest if array full
      - limits storage, still allows some scrollbar and refresh
    - no storage of data
      - messages drawn directly on Canvas
      - “copyarea” used to produce scrolling effect
      - can't view messages “scrolled off screen”, can't refresh if display temporarily obscured

*This option  
picked for  
demo*



## Client GUI

### Frame with BorderLayout



**java.awt.List  
in East area;  
Displays its  
collection of  
strings**

**Panel in South, flowlayout, holds label, textfield, actionbutton**

```
public class ClientProgram {
    public static void main(String[] args) {
        if(args.length != 3) { ... }
        InetAddress ina = null;
        try { ina = InetAddress.getByName(args[0]); }
        catch(UnknownHostException uhne) { ... }
        int port = 0;
        try { String ps = args[1].trim(); port = Integer.parseInt(ps);
            } catch (NumberFormatException nfe) { ... }

        Client cl = new Client(args[2]);
        try { Thread.sleep(100); } catch (InterruptedException ie) {}

        if(!cl.connect(ina, port)) { ... }
        cl.register();
        Thread t = new Thread(cl);
        t.start();
    }
}
```

**Create client & GUI** (pause to let AWT thread start)  
**Connect**  
**Send registration request**  
**Start thread for asynchronous messages**

```
public class ClientGUI extends Frame {
    public ClientGUI(Client cl) {
        // Next slide
    }
    public void addToList(String s) { fMembers.add(s); }
    public void removeFromList(String s) { fMembers.remove(s); }

    public void addPosting(Posting p) {
        // later slide
    }

    String getInput() { return fInput.getText(); }
    private List fMembers;
    private Canvas fCanvas;
    private TextField fInput;
    private Button fAction;
    private Font fFont;
}
```

ClientGUI

```
public ClientGUI(Client cl) {
    super("Micro-IRC for " + cl.getName());
    setLayout(new BorderLayout());
    fMembers=new List(5); fMembers.setEnabled(false);
    add(fMembers, "East");
    fCanvas = new Canvas(); fCanvas.setSize(500,400);
    fFont = new Font("Serif", Font.PLAIN, 12);
    fCanvas.setFont(fFont); add(fCanvas, "Center");
    Panel p = new Panel();
    p.add(new Label("Your message :"));
    fInput = new TextField(" ", 60); p.add(fInput);
    Button b = new Button("Send"); p.add(b);
    b.addActionListener(cl);
    add(p, "South");
    pack();
    addWindowListener(cl);
}
```

ClientGUI

## GUI building

- The usual
- Frame (container)
  - Layout manager defined
  - Components added
    - List
    - Canvas
    - Panel
      - Label
      - TextField
      - Button
- ActionListener, WindowListener

```
public void addPosting(Posting p) {
    Graphics g = fCanvas.getGraphics();
    FontMetrics f = g.getFontMetrics(fFont);           Scroll up
    int delta = f.getHeight();
    Dimension d = fCanvas.getSize();
    g.copyArea(0, delta, d.width, d.height-delta, 0, -delta);
    g.clearRect(0, d.height-delta, d.width,delta);

    g.setColor(Color.red);                             Add new
    g.drawString(p.author, 4, d.height - f.getDescent());
    int pos = 4 + f.stringWidth(p.author) + 10;
    if(pos<50) pos = 50;
    g.setColor(Color.black);
    g.drawString(p.comment, pos, d.height - f.getDescent());
}
```

(As data only written to screen, information is lost if window is hidden then re-exposed)  
*ClientGUI*

## Client

- Client – Runnable, ActionListener, WindowListener
  - Runnable – for separate thread that gets asynchronous messages from server
  - ActionListener – when GUI's button clicked, grab input from textfield and post it to server
  - WindowListener – on window closing, hide GUI, stop run thread, send quit message to server

```
public class Client implements Runnable, ActionListener, WindowListener {
    public Client(String nickname) { ... }
    public void register() { ... }
    public String getName() { return fName; }
    boolean connect(InetAddress ina, int port) { ... }
    public void run() { ... }
    public void actionPerformed(ActionEvent e) { ... }
    public void windowOpened(WindowEvent e) { ... }
    public void windowClosing(WindowEvent e) { ... }
    public void windowClosed(WindowEvent e) { ... }
    public void windowIconified(WindowEvent e) { ... }
    public void windowDeiconified(WindowEvent e) { ... }
    public void windowActivated(WindowEvent e) { ... }
    public void windowDeactivated(WindowEvent e) { ... }
    private Socket mySocket;
    private ObjectInputStream in;
    private ObjectOutputStream out;
    private ClientGUI myGUI;
    private String fName;
    private Thread runThread;
}
```

*Client*

```
public Client(String nickname) {
    fName = nickname;
    myGUI = new ClientGUI(this);
    myGUI.setVisible(true);
}
public String getName() { return fName; }
boolean connect(InetAddress ina, int port) {
    try {
        mySocket = new Socket(ina, port);
        out = new ObjectOutputStream(mySocket.getOutputStream());
        out.flush();
        in = new ObjectInputStream(mySocket.getInputStream());
    } catch (IOException io) { return false; }
    return true;
}
```

*Wrap byte level socket streams  
in object streams to allow transfer  
of objects*

*Client*

```
public void register() {
    try {
        Message m = new Message("Register", fName);
        out.writeObject(m); out.flush();
        Object o = in.readObject();
        if(o instanceof MyException) throw (MyException) o;
        m = (Message) o;
        InitialData id = (InitialData) m.otherdata;
        String[] players = id.playerNames;
        for(String player : players)
            myGUI.addToList(player);
        Posting[] posts = id.postings;
        for(Posting p : posts) {
            myGUI.addPosting(p);
        }
    } catch(ClassNotFoundException cnfe) { ... }
    catch(ClassCastException cce) { ... }
    catch(MyException me) { ... }
    catch(IOException io) { ... }
}
```

*Client*

```

public void run() {
    runThread = Thread.currentThread();
    try {
        for(;;) {
            Message m = (Message) in.readObject();
            String s = m.operation;
            if(s.equals("Add_Client")) {
                String cname = (String) m.otherdata;
                myGUI.addToList(cname);
            } else if(s.equals("Remove_Client")) {
                String cname = (String) m.otherdata;
                myGUI.removeFromList(cname);
            } else if(s.equals("Post")) {
                Posting p = (Posting) m.otherdata;
                myGUI.addPosting(p);
            }
        }
    } catch (Exception e) { ... }
}

```

Client

```

public void actionPerformed(ActionEvent e) {
    String msg = myGUI.getInput();
    if(msg == null) return;
    try {
        Message m = new Message("Post", msg);
        out.writeObject(m);
        out.flush();
        out.reset();
    } catch (IOException io) { ... }
}

```

Client

```

public void windowClosing(WindowEvent e) {
    myGUI.setVisible(false);
    runThread.stop();
    try {
        Message m = new Message("Quit", null);
        out.writeObject(m);
        out.flush();
        try { Thread.sleep(1000); } catch (InterruptedException ie) {}
        in.close();
        out.close();
    } catch (IOException io) {}
    System.out.println("Client terminated");
    System.exit(0);
}
public void windowClosed(WindowEvent e) {}
...

```

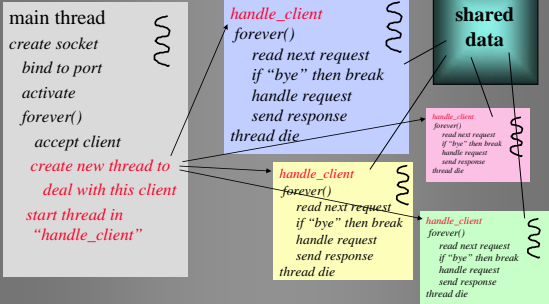
Client

## Your programming of Client - server systems

- Server (*things that you should think about*)
  - setting up listener mechanism, accepting new client and getting socket, creating thread for new client –
  - “handle\_client”
    - reading request;
    - switch(keyword) into ...
      - invoke separate functions for each of the commands that this server handles
    - generation of response
  - how does it terminate?

## Threaded server

All the one process (interacting clients)



## Micro-IRC Server

- Server
  - normal mechanism
    - “receptionist” listening at published well know port
    - accepts connections (getting new port-socket combination)
    - creates ClientHandler objects (and associated threads) for clients
  - shared data
    - vector with ten most recent messages
    - vector with ClientHandlers

```

Server
ClientHandler.java
Server.java
Server
SharedData.java

```

## Micro-IRC Server

### – ClientHandlers

- hard code
  - wait for “Register”, check nickname (if name in use, return exception, destroy this ClientHandler)
  - register client with shared data
  - send backlog info
- run main loop
  - read command
    - if Quit, terminate loop
    - if Post, add message to shared data
    - ?
- on exit from main loop, or any i/o error, deregister this ClientHandler from shared data and terminate

```
public class Server {
    private static final int DEFAULT_PORT = 54321;
    private static final int THREADPOOL_SIZE = 8;
    public static void main(String[] args) {
        int port = DEFAULT_PORT;
        if(args.length==1) {
            try { port = Integer.parseInt(args[0]);
            } catch (NumberFormatException ne) { }
        }
        ServerSocket reception_socket = null;
        try {
            reception_socket = new ServerSocket(port);
        }
        catch(IOException ioe1) { ... }
        SharedData sd = new SharedData();
        ExecutorService pool = Executors.newFixedThreadPool(
            THREADPOOL_SIZE);
        for(;;) {
            try {
                Socket client_socket = reception_socket.accept();
                ClientHandler ch = new ClientHandler(client_socket, sd);
                pool.execute(ch);
            }
            catch(Exception e) { ... }
        }
    }
}
```

**Threaded-server, thread-pool style**

*Server*

## Threads & Locking

- Multiple clients
- Multiple threads in server
- Possibly more than one thread trying to modify shared data
- Possibly more than one thread trying to use a particular ClientHandler!
  - ClientHandler’s own thread receiving a message
  - Some other thread trying to broadcast another user’s posting
- Need locks – synchronized methods

```
public class SharedData {
    private Vector<Posting> data;
    private Vector<ClientHandler> subscribers;

    public SharedData() {
        data = new Vector<Posting>();
        subscribers = new Vector<ClientHandler>();
    }

    public synchronized void addClient(ClientHandler ch, String
        nickname) throws MyException
    { ... }

    public synchronized void removeClient(ClientHandler ch)
    { ... }

    public synchronized void postMessage(String poster, String msg)
    { ... }
}
```

*SharedData*

```
public synchronized void addClient(ClientHandler ch, String nickname) throws
    MyException {
    for(ClientHandler c : subscribers)
        if(c.getName().equals(nickname))
            throw new MyException("Duplicate name");
    // Send notification to each existing subscriber that there is a new participant
    Message m = new Message("Add_Client", nickname);
    for(ClientHandler c: subscribers)
        c.sendMessage(m);
    // Add new client to collection
    subscribers.add(ch);
    // Compose a response for new client
    String[] participants = new String[subscribers.size()];
    int i=0;
    for(ClientHandler c: subscribers)
        participants[i++] = c.getName();
    Posting[] posts = data.toArray(new Posting[0]);
    InitialData id = new InitialData(participants, posts);
    Message reply = new Message("OK", id);
    ch.sendMessage(reply);
}
```

*SharedData*

```
public synchronized void removeClient(ClientHandler ch) {
    subscribers.remove(ch);
    Message m = new Message("Remove_Client", ch.getName());
    for(ClientHandler c : subscribers)
        c.sendMessage(m);
}

public synchronized void postMessage(String poster, String msg) {
    Posting p = new Posting(poster, msg);
    data.add(p);
    if(data.size()>10)
        data.removeElementAt(0);
    Message m = new Message("Post", p);
    for(ClientHandler c: subscribers)
        c.sendMessage(m);
}
```

*SharedData*

## Locks on SharedData

- Synchronization locks on all methods
- Guarantee only one thread will be updating shared data

```
public class ClientHandler implements Runnable {
    private ObjectInputStream input;
    private ObjectOutputStream output;
    private SharedData shared;
    private Socket mySocket;
    private String name;
    public ClientHandler(Socket aSocket,
        SharedData sd) throws IOException { ... }
    public synchronized void sendMessage(
        Message m) { ... }
    public String getName() { return name; }
    private boolean handleRegister() { ... }
    public void run() { ... }
}
```

ClientHandler

```
public ClientHandler(Socket aSocket, SharedData sd) throws IOException {
    mySocket = aSocket;
    output = new ObjectOutputStream(aSocket.getOutputStream());
    output.flush();
    input = new ObjectInputStream(aSocket.getInputStream());
    shared = sd;
}

Wrap byte level socket streams
in object streams to allow transfer
of objects

public synchronized void sendMessage(Message m) {
    try {
        output.writeObject(m);
        output.flush();
        output.reset();
    }
    catch(IOException ioe) { ... }
}
```

ClientHandler

```
private boolean handleRegister() {
    try {
        Message m = (Message) input.readObject();
        String operation = m.operation;
        if(!operation.equals("Register"))
            return false;
        name = (String) m.otherdata;
        shared.addClient(this, name);
        return true;
    }
    catch(ClassNotFoundException cnfe) { ... }
    catch(IOException ioe) { ... }
    catch(MyException me) {
        try {
            output.writeObject(me);
            output.flush();
        }
        catch(IOException io) {}
    }
    return false;
}
```

ClientHandler

```
public void run() {
    if(!handleRegister()) {
        try { mySocket.close(); } catch(IOException ioe) {}
        return;
    }

    for(;;) {
        try {
            Message m = (Message) input.readObject();
            String operation = m.operation;
            if(operation.equals("Quit"))
                break;
            else
                if(operation.equals("Post"))
                    shared.postMessage(name, (String) m.otherdata);
        }
        catch(Exception e) { break; }
    }

    shared.removeClient(this);
    try { mySocket.close(); } catch(IOException ioe) {}
}
```

ClientHandler

## MicroIRC

- As networked application, quite likely to get a break in client-server link without formal logout
  - client program died
  - server terminated
  - ...
- No particular event sent to other process, but will get fail at next read/write operation on socket.
- Code tries to trap these and :
  - (if break spotted by client) terminate client process
  - (if break spotted by server) destroy client handler object