CS18000: Problem Solving and Object-Oriented Programming

Network I/O
Video 1
Internet Communication
External Communication

Network Communication
Some (Simplified) Definitions

• **Internet Protocol (IP):**
  Identifies hosts (servers, workstations, laptops, etc.) with a unique address (e.g., 128.10.2.21)

• **Domain Name System (DNS):**
  Maps domain names (e.g., galahad.cs.purdue.edu) to IP addresses (e.g., 128.10.9.143)

• **Transmission Control Protocol (TCP):**
  Identifies ports on hosts for a network connection

• **Socket:** IP address plus TCP port

• Two sockets makes a network connection
Client-Server

- A **Server** is a process that waits for a connection
- A **Client** is a process that connects to a server
- At different times, a process may be both a client and a server
- Need not be associated with a specific computer: Any computer can have both client and server processes running on it
- Once connected, the client and server can both read and write data to one another asynchronously ("a bi-directional byte pipe")
Use of Sockets

- Clients and Servers communicate via Sockets
- Socket: IP address plus TCP port
- Think: street name plus house number
- IP addresses
  - Identifies a computer on the Internet
  - Public addresses are globally unique
  - Represented using dotted-decimal (byte) notation: 128.10.9.143
  - Some firewalls translate addresses to internal ones (e.g., PAL)
- Port number
  - 0-65535 (16 bits)
  - Low-valued port numbers are reserved for privileged processes
Objects and Networking in Java

• You know that Java objects can be written to and read from files

• Java objects can also be exchanged over network connections

• Uses ObjectOutputStream and ObjectInputStream

• Tricky bits...
  – ObjectOutputStream generates a “header” of information that must be read
  – Requires “flush” to ensure ObjectInputStream reader is not blocked
ObjectStream Client-Server Timeline

Server Process

1. Server starts: opens ServerSocket and blocks (waiting for client)
2. Server receives connection
3. Server opens ObjectOutputStream, sends header with flush
4. Server receives object stream header
5. Client and Server exchange objects in agreed upon order

Client Process

1. Client starts: opens Socket to Server
2. Client opens ObjectOutputStream, sends header with flush
3. Client receives object stream header
4. Client and Server exchange objects in agreed upon order
Java Networking Class: Socket

• Models a TCP/IP socket
• Used by Client to identify Server
  – IP address (or DNS name)
  – Port number
  – new Socket("pc.cs.purdue.edu", 12190)
• Used by Server to identify connected Client
• Provides streams for communications:
  – getOutputStream()
  – getInputStream()
Java Networking Class: ServerSocket

- Used by Server to wait for a Client to connect
- Constructor specifies TCP port number to use:
  
  ```java
  ServerSocket ss = new ServerSocket(4242);
  ```

- Method `accept()` blocks waiting for connection

  ```java
  Socket socket = ss.accept();
  ```
Video 2
Clients and Servers
import java.io.*;
import java.net.*;

public class Server {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        // create socket on agreed-upon port...
        ServerSocket serverSocket = new ServerSocket(4242);

        // wait for client to connect, get socket connection...
        Socket socket = serverSocket.accept();

        // open output stream to client, flush send header, then input stream...
        ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
        oos.flush(); // ensure data is sent to the client
        ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());

        // send object(s) to client...
        String s1 = "hello there";
        oos.writeObject(s1);
        oos.flush(); // ensure data is sent to the client
        System.out.printf("sent to client: %s\n", s1);

        // read object(s) from client...
        String s2 = (String) ois.readObject();
        System.out.printf("received from client: %s\n", s2);

        // close streams...
        oos.close();
        ois.close();
    }
}
```java
import java.io.*;
import java.net.*;

public class Client {
    public static void main(String[] args) throws UnknownHostException, IOException, ClassNotFoundException {

        // create socket on agreed upon port (and local host for this example)...
        Socket socket = new Socket("data.cs.purdue.edu", 4242);

        // open input stream first, gets header from server...
        ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
        // open output stream second, send header to server...
        ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
        oos.flush();  // ensure data is sent to the server

        // read object(s) from server...
        String s1 = (String) ois.readObject();
        System.out.printf("received from server: %s\n", s1);

        // write object(s) to server...
        String s2 = s1.toUpperCase();
        oos.writeObject(s2);
        oos.flush(); // ensure data is sent to the server
        System.out.printf("sent to server: %s\n", s2);

        // close streams...
        oos.close();
        ois.close();
    }
}
```
Client-Server with Threads

• In many (most?) cases, a single server is connected to by multiple clients
• Server must be able to communicate with all clients simultaneously—no blocking
• Technique: server creates a separate thread to handle each client as it connects
• Client and server may also each create separate thread for reading and writing
Example: Echo Server

• Simple server that accepts connections from multiple clients
• Spawns a thread for each client
• Reads lines from the connection, logs information, echoes lines back
• Useful for debugging network and code
Echo Server Timeline

Server Process

Server starts: opens ServerSocket and blocks (waiting for client)

Server receives connection

Server creates thread

Server thread 1 reads/echoes line

Client Process

Client 1 starts: opens Socket to Server

Client 2 starts: opens Socket to Server

Client 3 starts: opens Socket to Server

Client 1 sends line

Client 1 reads line

Client 1 reads line
Example: Echo Server (1)

```java
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;

public class EchoServer implements Runnable {
    Socket socket;

    public EchoServer(Socket socket) {
        this.socket = socket;
    }

    // continued...

    // continued...
```
Example: Echo Server (2)

// run method for thread...

public void run() {
    System.out.printf("connection received from %s\n", socket);
    try {
        // socket open: make PrintWriter and Scanner from it...
        PrintWriter pw = new PrintWriter(socket.getOutputStream());
        Scanner in = new Scanner(socket.getInputStream());

        // read from input, and echo output...
        while (in.hasNextLine()) {
            String line = in.nextLine();
            System.out.printf("%s says: %s\n", socket, line);
            pw.printf("echo: %s\n", line);
            pw.flush();
        }

        // input done, close connections...
        pw.close();
        in.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
public static void main(String[] args) throws IOException {
    // allocate server socket at given port...
    ServerSocket serverSocket = new ServerSocket(4343);
    System.out.printf("socket open, waiting for connections on %s\n", serverSocket);

    // infinite server loop: accept connection, 
    // spawn thread to handle...
    while (true) {
        Socket socket = serverSocket.accept();
        EchoServer server = new EchoServer(socket);
        new Thread(server).start();
    }
}
Network Communication in Java

• Uses standard file I/O classes: low-level, high-level, object, and text

• Adds abstractions to deal with network connections
  – ServerSocket to wait for connections
  – Socket abstracts a TCP socket (IP address + port)

• Uses threads to improve responsiveness and avoid blocking