Socket programming

Complement for the programming assignment
INFO-0010
Outline

- Prerequisites
- Socket definition
- Briefing on the Socket API
- A simple example in Java
- Multi-threading and Synchronization
- Debugging tools
- Project overview
Prerequisites

Java Programming

• Compiling and executing Java programs
• Comments and Indentation
• Including libraries
• Classes, objects, methods, constructors
• Inheritance and Implementation
• Native types: boolean, byte, int, char, float, and basic operations on them.
• Simple arrays: [ ], ArrayList
• Alternatives and loops: if, while, for
• String manipulation: creation, concatenation, substring, comparison
• Thread creation
• Exceptions handling: try, catch, throws, throw.
• Input/OutputStreams usage: read, write, flush
What are sockets?

- Interface to network protocol stack
  - Typically to transport layer

Connectionless (Datagram) Socket

Connection-Oriented Socket

Transport (UDP and TCP)

Network

Data Link (MAC / LLC)

Physical (PHY)
What are sockets? (2)

• A socket is an end-point of communication which identifies a local “process” at one end of a communication association
  – A socket is a half association
  – \{ protocol, local-address, local-port \}

• A communication association is identified by two half associations

  \{ protocol,
  local-address, local-port,
  remote-address, remote-port
  \}
Communication models

• Datagrams (UDP)
  – Message-oriented
  – Connectionless
  – Unreliable
  – No congestion control

• Connections (TCP)
  – Stream-oriented
  – Requires a connection
  – Reliable (no packet loss, no reordering)
  – Congestion control

• Raw
  – For traffic generation/capture
UDP vs TCP

- Conceptually:
  - Sent datagram = received datagram
  - No congestion control
  - No reordering
  - Packet loss
UDP vs TCP

• Conceptually:
  - Socket
  - Application
  - Waiting for more
    → setTcpNoDelay(true)
    (Nagle’s algo)
  - Buffering
    → flush()

Stream oriented:
- May require multiple reads
- Order preserved
- No loss

1000 bytes
512 bytes
488 bytes

TCP
Connections

Implemented by TCP
- Reliable stream transfer
- Guarantees delivery and ordering provided connection not broken
- Does congestion control
  - What you have sent to the socket may not have left the box yet!
    - You can use `out.flush()` to force the writing to the socket
    - You can use `socket.setTcpNoDelay(true)` to disable Nagle’s algorithm
- Chunks read may be different from chunks sent, but streams are identical
  - Programmer must check how many bytes should be read
  - Convention or application protocol header
Sockets' life cycle (syscalls)

1. `socket` (client side)
2. `bind`
3. `connect`
4. `data transmission`
5. `close`

1. `socket` (server side)
2. `bind`
3. `listen`
4. `accept`
5. `connection accepted`
6. `close`

Connection request from client to server.
Socket's life cycle (Java)

- `new Socket()`
  - `bind`
  - `connect`
- `new ServerSocket(...)`
  - `bind`
  - `listen`
  - `accept`

Connection request
- `new Socket("host", port)`
- Connection accepted

Data transmission through `InputStream` and `OutputStream`

Client side
- `close`

Server side
- `close`
Simple example: client/server chat

- Objective: The server duplicates each incoming message to all the clients.

- For starters, the server is only an echo server.
The Client Side – a bot

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

class Bot {
    public static void main(String argv[]) throws Exception {
        Socket s = new Socket("localhost", 8086);
        OutputStream out = s.getOutputStream();
        InputStream in = s.getInputStream();
        byte[] msg = new byte[64];
        out.write("ANnA joined the channel").getBytes();
        while (true) {
            if (in.read(msg) <= 0) break;
            if (new String(msg).startsWith("ANnA"))
                out.write("ANnA feels fine, thanks.
".getBytes());
        }
        s.close();
    }
}
```

main() should always catch exceptions. Would only make the code harder to read in this example.

Use explicit variable names. We mainly focus on the class names in this example.
The Server Side – Incoming Connection

class Server {
    public static void main ( String argv [] ) throws Exception {
        ServerSocket ss = new ServerSocket (8086) ;
        while ( true ) {
            Socket ts = ss.accept () ;
            OutputStream out = ts.getOutputStream () ;
            InputStream in = ts.getInputStream () ;
            out.write ("Hello, this is the echo server". getBytes () ) ;
            byte msg [ ] = new byte [64] ;

            while ( true ) {
                int len= in.read (msg); // get bytes (max 64)
                if (len <=0) break ; // connection closed by peer ?
                out.write (msg,0,len); // send them away .
                out.flush(); // don’t wait for more .
            }
            ts .close ();
        }
    }
}

What if multiple clients connect simultaneously?
The Server Side – multithreading

```java
class Server {
    public static void main ( String argv [ ] ) throws Exception {
        ServerSocket ss = new ServerSocket (8086) ;
        while ( true ) {
            Socket ts = ss.accept () ;
            Worker w = new Worker(ts);
            w.start(); //Worker extends Thread
        }
    }
}
```

• We spawn a thread every time a connection arrives
• That fresh-new thread will deal with the new client.
• And the main thread can return to the welcoming of incoming clients.
The Server Side – defining a thread

```java
class Worker extends Thread {
    Socket s;
    Worker (Socket _s) { s = _s; }
    @Override
    public void run () {
        try {
            OutputStream out = s.getOutputStream();
            InputStream in = s.getInputStream();
            out.write("Hello, this is the echo server").getBytes());
            byte msg [] = new byte [64];

            while (true) {
                int len = in.read(msg); // get bytes (max 64)
                if (len <=0) break; // connection closed by peer?
                out.write(msg,0,len); // send them away .
                out.flush(); // don’t wait for more .
            }
            s.close(); //acknowledge end of connection
        } catch (Exception any) {
            System.err.println("worker died " + any);
        }
    }
}
```
Shared objects

• What if some objects need to be manipulated by different threads?

• For instance, we could keep a list (e.g. `ArrayList`) of all the `OutputSteam` and turn the "echo" server into a real chat server.

• Multiple threads will use elements of the list simultaneously
  • Execution is `concurrent` and `non-atomic`
  • Consistency is thus not ensured

• Solution: **only one thread at a time** can use the elements in the list
  • Deem the sending phase a **critical section**
  • Implement **mutual exclusion** over critical section – i.e. prevent multiple threads from entering at once
  • `synchronized` keyword
The Server Side – shared objects

// all is just an ArrayList where each OutputStream is .add() ed
// out is the OutputStream corresponding to the Socket from which we are receiving

void send ( byte msg [ ] , int len ) throws Exception {
    synchronized ( all ) {
        for ( Enumeration e = all.elements ( ) ; e.hasMoreElements () ; ) {
            OutputStream o = (OutputStream) e.nextElement ();
            if ( o != out ) {
                o.write (msg, 0, len ); // send them away .
                o.flush (); // don’t wait for more .
            }
        }
    }
}
Reading from TCP streams

- **Recall**: Chunks read may be different from chunks sent
- Sending a message on one side and calling `read` on the other side will not guarantee that the message has been fully read.
  - By the way, our chat bot is thus not very well coded.
- **You must find the message boundaries**
  - Message always have the same size $X$
    - While I haven’t received $X$ bytes, I keep calling `read` (possibly adjusting the number of bytes to read next)
  - Messages are separated by a delimiter
    - I keep reading until I see that delimiter
    - I might read more than one message, so the extra bytes must be considered as the beginning of a new message
    - If the delimiter is `\r\n`, some classes might help (e.g. BufferedReader)
  - Messages are preceded with a header
    - The header has a fixed size
    - In the header, I find the size of the message.
A few debugging tools

Each bug has its proper catcher. So, use `catch` and `printStackTrace(...)` wisely!

telnet (for direct connection)
strace –enetwork –f java Server

```
Pid 4199

[pid 4199] setsockopt(5, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
[pid 4199] bind(5, {sa_family=AF_INET6, sin6_port=htons(8086), inet_pton(AF_INET6, "::", &sin6_addr), sin6_flowinfo=0, sin6_scope_id=0}, 28) = 0
[pid 4199] listen(5, 50) = 0
[pid 4199] accept(5, {sa_family=AF_INET6, sin6_port=htons(3764), inet_pton(AF_INET6, "::ffff:127.0.0.1", &sin6_addr), sin6_flowinfo=0, sin6_scope_id=0}, [28]) = 6
Process 4211 attached
[pid 4211] sendto(-1251980320, umovestr: Input/output error
0xc, 3086176244, 0, ptrace: umovestr: Input/output error
{...}, 3042985144) = 0
[pid 4199] accept(5, <unfinished ...>
[pid 4211] send(6, "Hello, this is the echo server", 30, 0) = 30
[pid 4211] recv(6, <unfinished ...>
[pid 4199] (... accept resumed> {sa_family=AF_INET6, sin6_port=htons(3765), inet_pton(AF_INET6, "::ffff:127.0.0.1", &sin6_addr), sin6_flowinfo=0, sin6_scope_id=0}, [28]) = 7
Process 4215 attached
[pid 4215] sendto(-1252312096, umovestr: Input/output error
0xc, 3086176244, 0, ptrace: umovestr: Input/output error
{...}, 30429853368) = 0
[pid 4199] accept(5, <unfinished ...>
[pid 4215] send(7, "Hello, this is the echo server", 30, 0) = 30
[pid 4215] recv(7, System calls
```

This screenshot shows a strace output for a Java server process, demonstrating system calls like setsockopt, bind, listen, and accept, which are crucial for network setup and handling client requests.
strace -enetwork -f java Server

Socket N°\rightarrow 5 = new ServerSocket(8086)

# bytes to send

# bytes handled by TCP
Some command lines

- (examples are better commented on the web).
- `javac Prog.java` to compile
- `java Prog` to launch
- `telnet localhost 8086` to test
- `strace -e trace=network -f java Prog` to track system calls issued by your program
- `netstat -t lp` to list server sockets
- `netstat -tcp` to list running connections.