Socket programming

Complement for the programming assignment

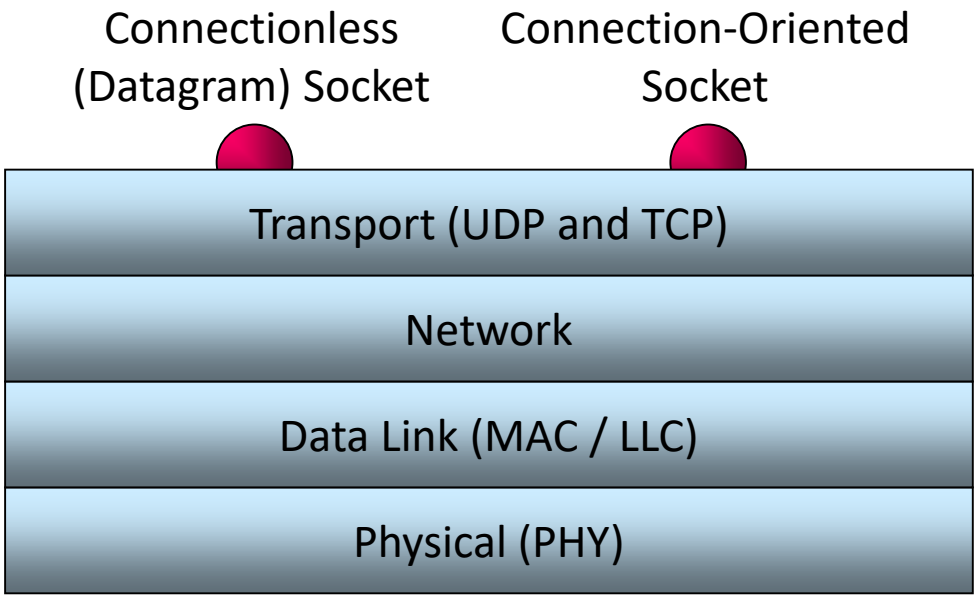
INFO-0010

Outline

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

What are sockets?

- Interface to network protocol stack
  - Typically to transport layer



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

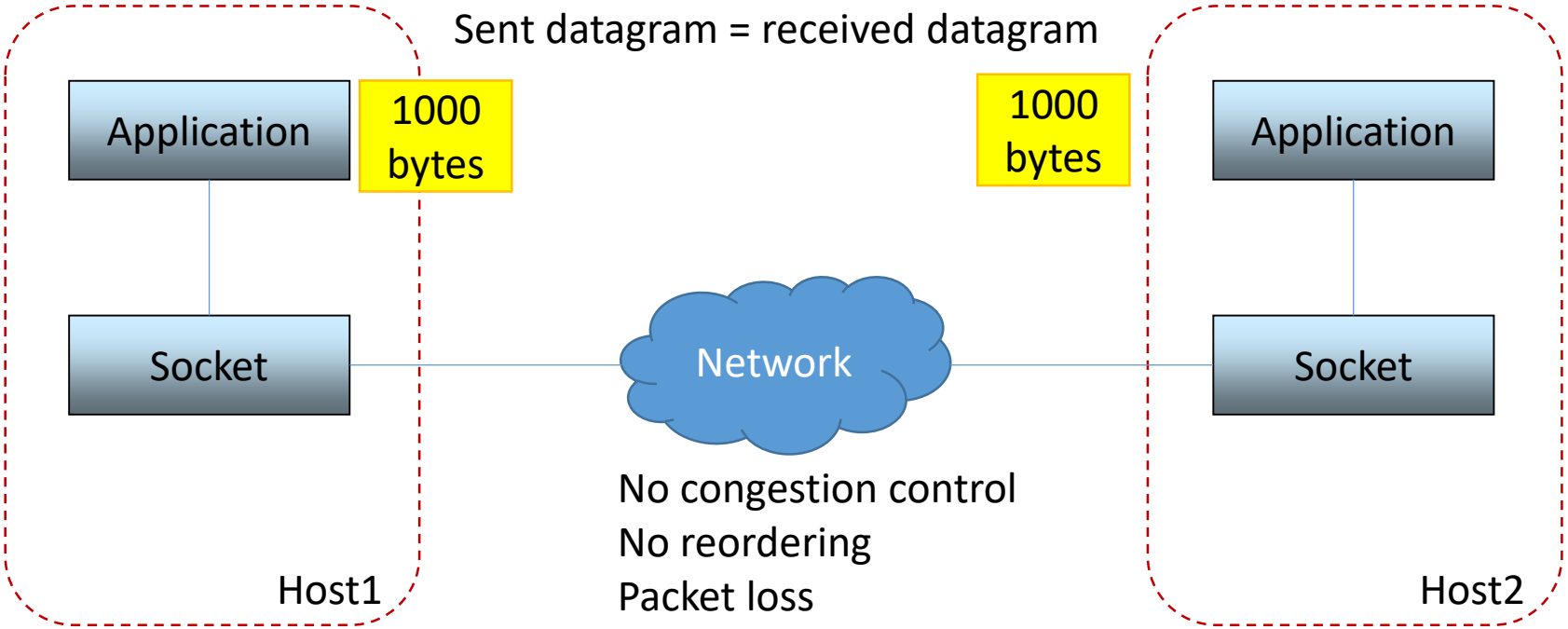
```
{
        protocol,
        local-address, local-port,
        remote-address, remote-port
}
```

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 trafic generation/capture

UDP vs TCP

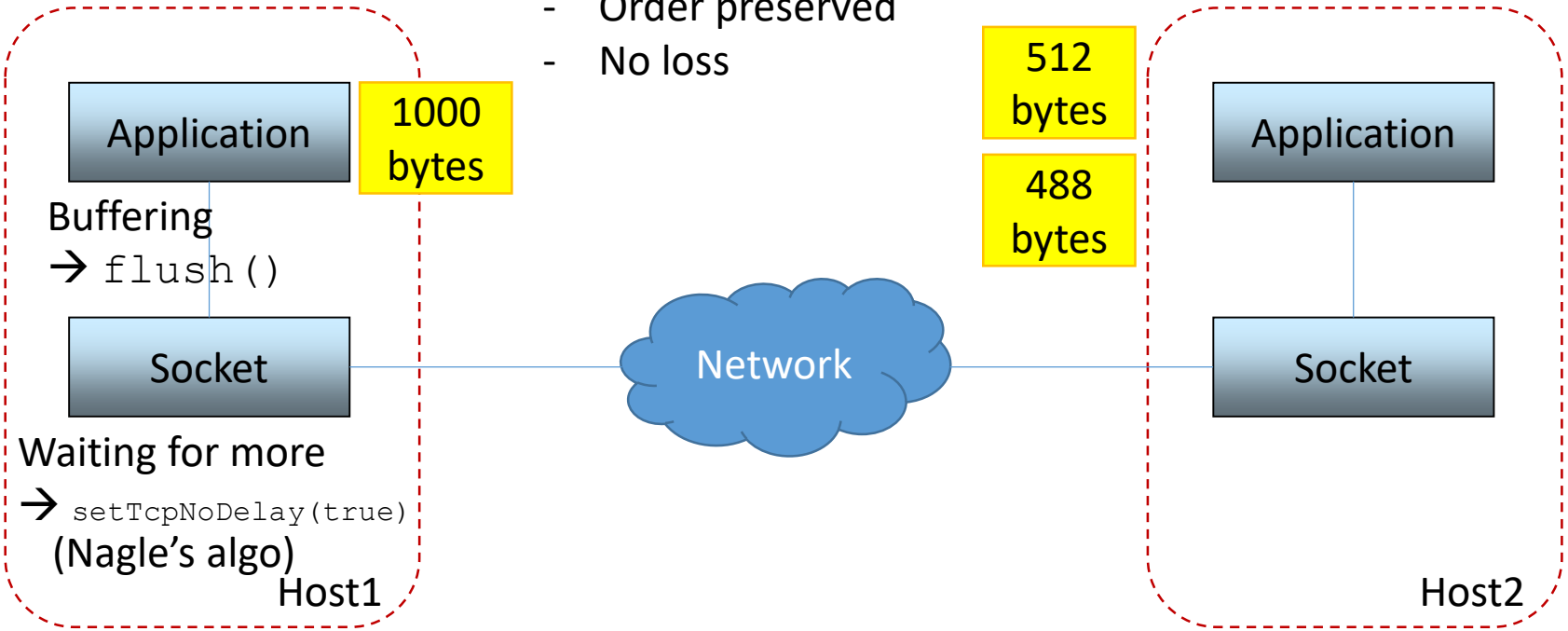
- Conceptually:



UDP vs TCP

- Conceptually:

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

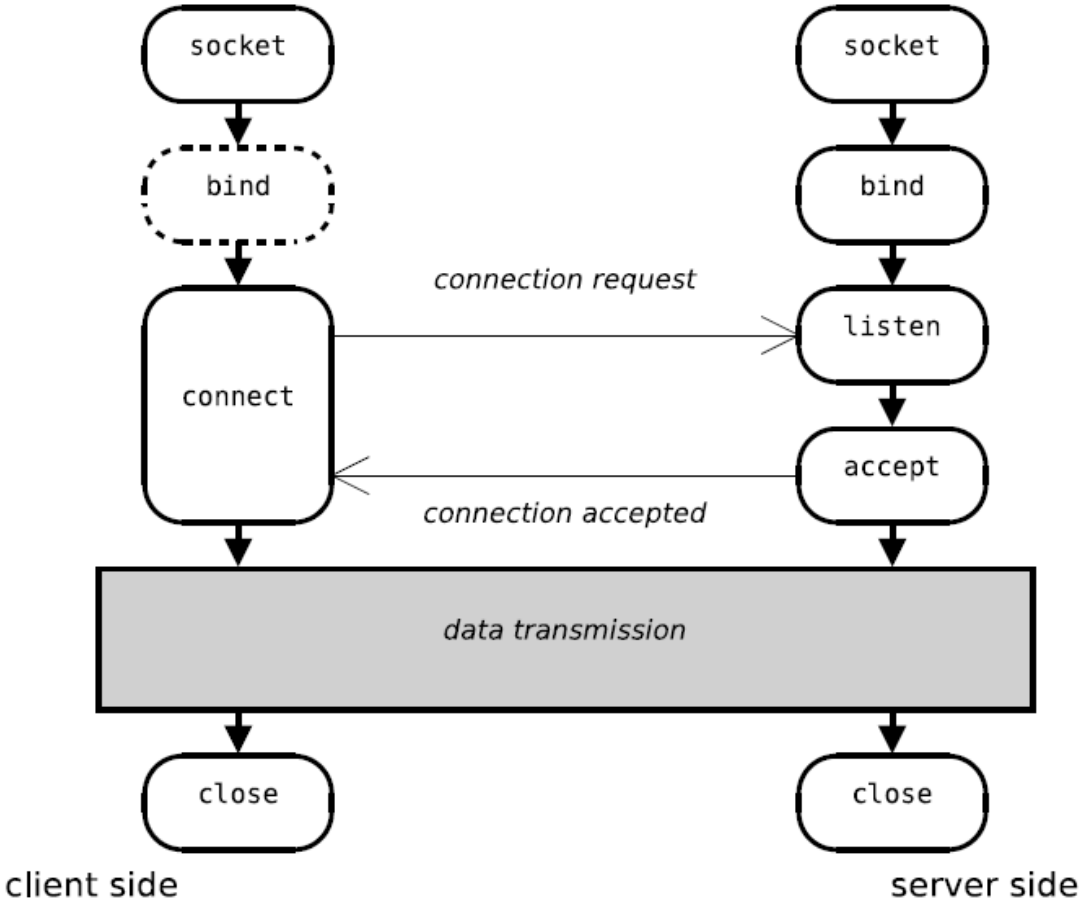


Connections

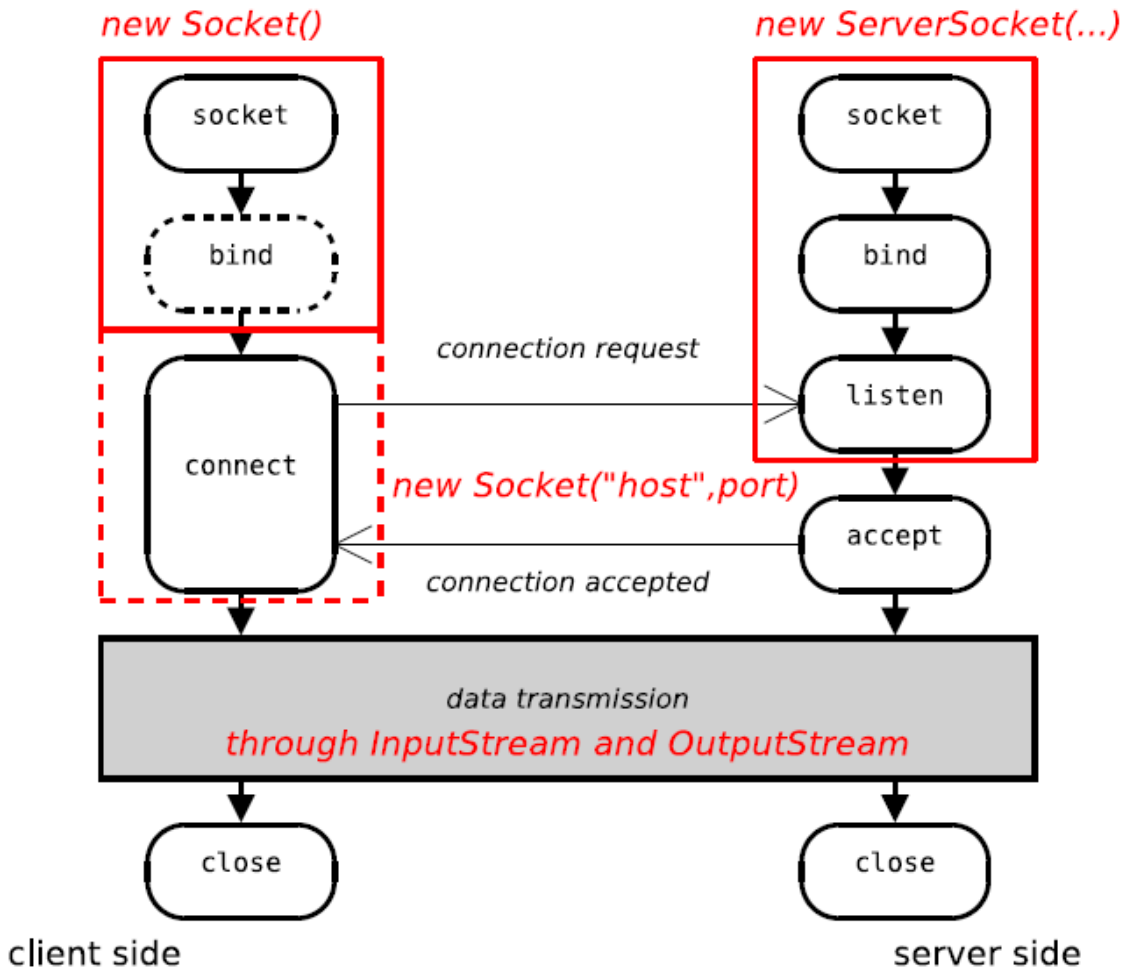
Implemented by TCP

- Reliable stream transfer
  - Chunks read may be different from chunks sent
    - ✓But streams are identical
- 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
- Programmer must check how many bytes should be read
  - Convention or application protocol header
  - InputStream can provide **readFully()**

Sockets' life cycle (syscalls)

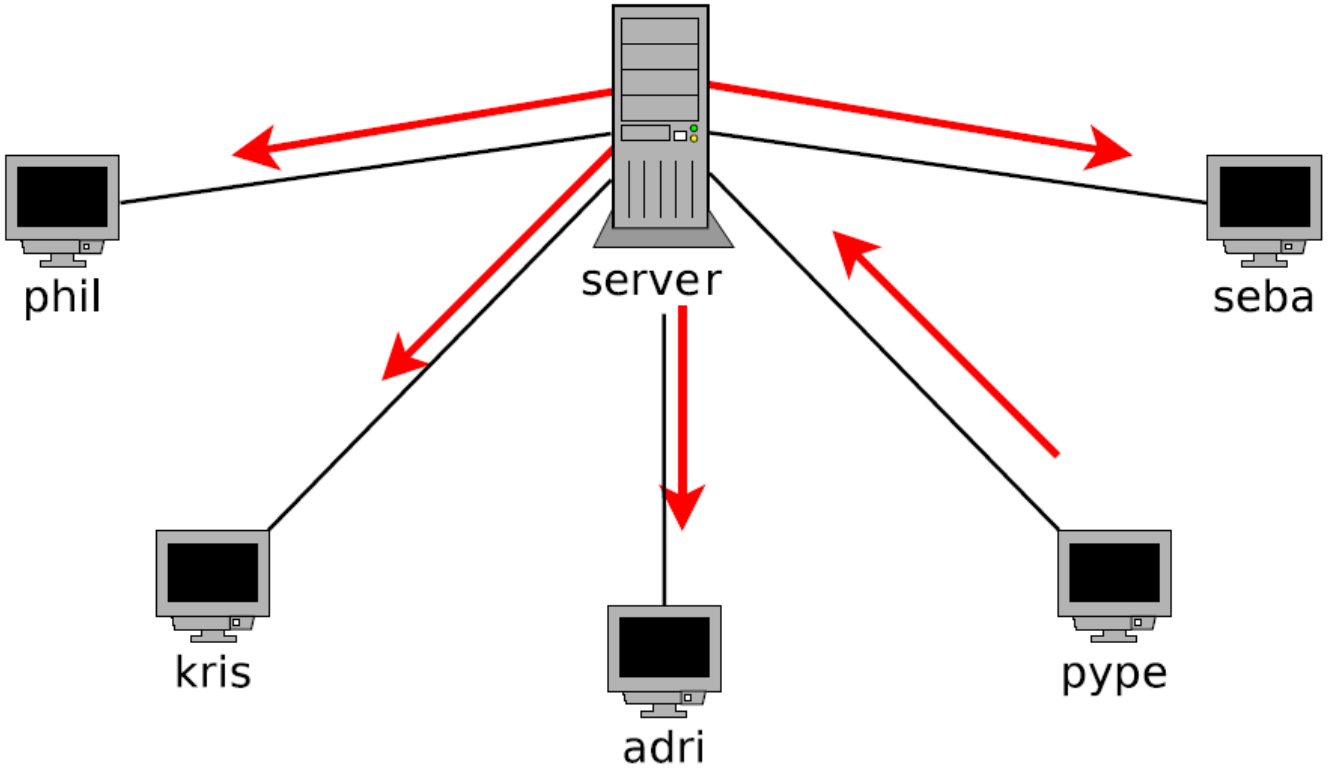


Socket's life cycle (Java)



Simple example : client/server chat

- `telnet` towards a well-known server, from each client
- The server duplicates each incoming message to all the clients.



The Client Side – a bot

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

```
import java.i o.*;
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.\n". getBytes ( ) ) ;
                }
                s.close ( ) ;
        }
}
```

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

```
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

```
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) ;
                }
        }
}
```

Extending the "Thread" class

- One way to create thread in Java
  1. Define class that '*extends*' class called **java.lang.Thread**
  2. Create object of that class

- Must override the **public void run ( );** method and state so by adding **@Override** in the method signature to prevent a warning

Implementing the “Runnable” interface

- Another way to create thread in Java
  1. Define class that ‘*implements*’ interface called **java.lang.Runnable**
  2. Create object of class Thread, passing object of above class to Thread constructor

- N.B. an *interface* is like a class with method signatures but no method implementations
  - “implementing” interface is like extending class
  - But, in addition, must provide implementations of all method signatures in specified interface
  - In Runnable there’s only one signature
    - **public void run ( );**

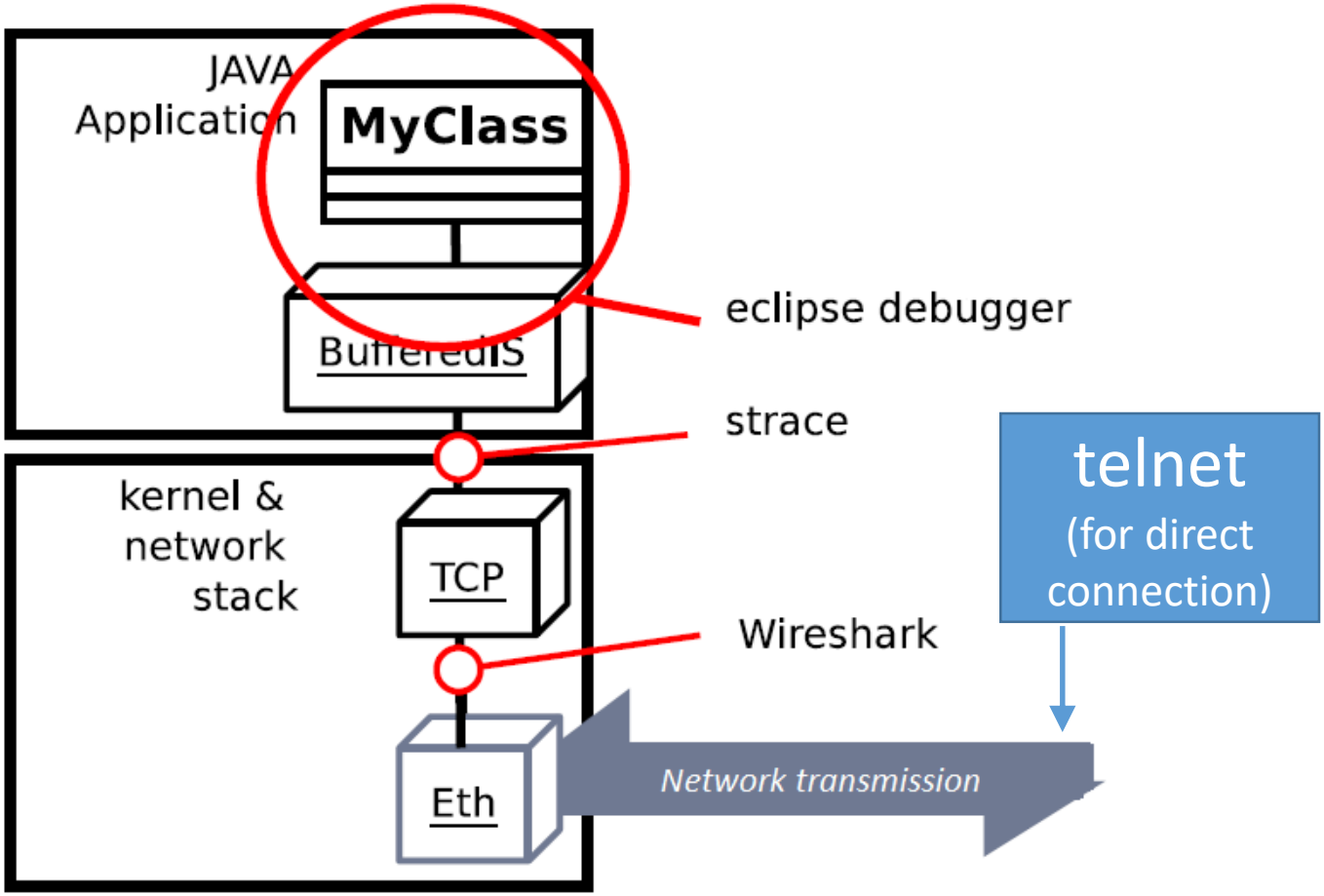
Shared objects

- 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

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 .
                        }
                }
        }
}
```

A few debugging tools



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

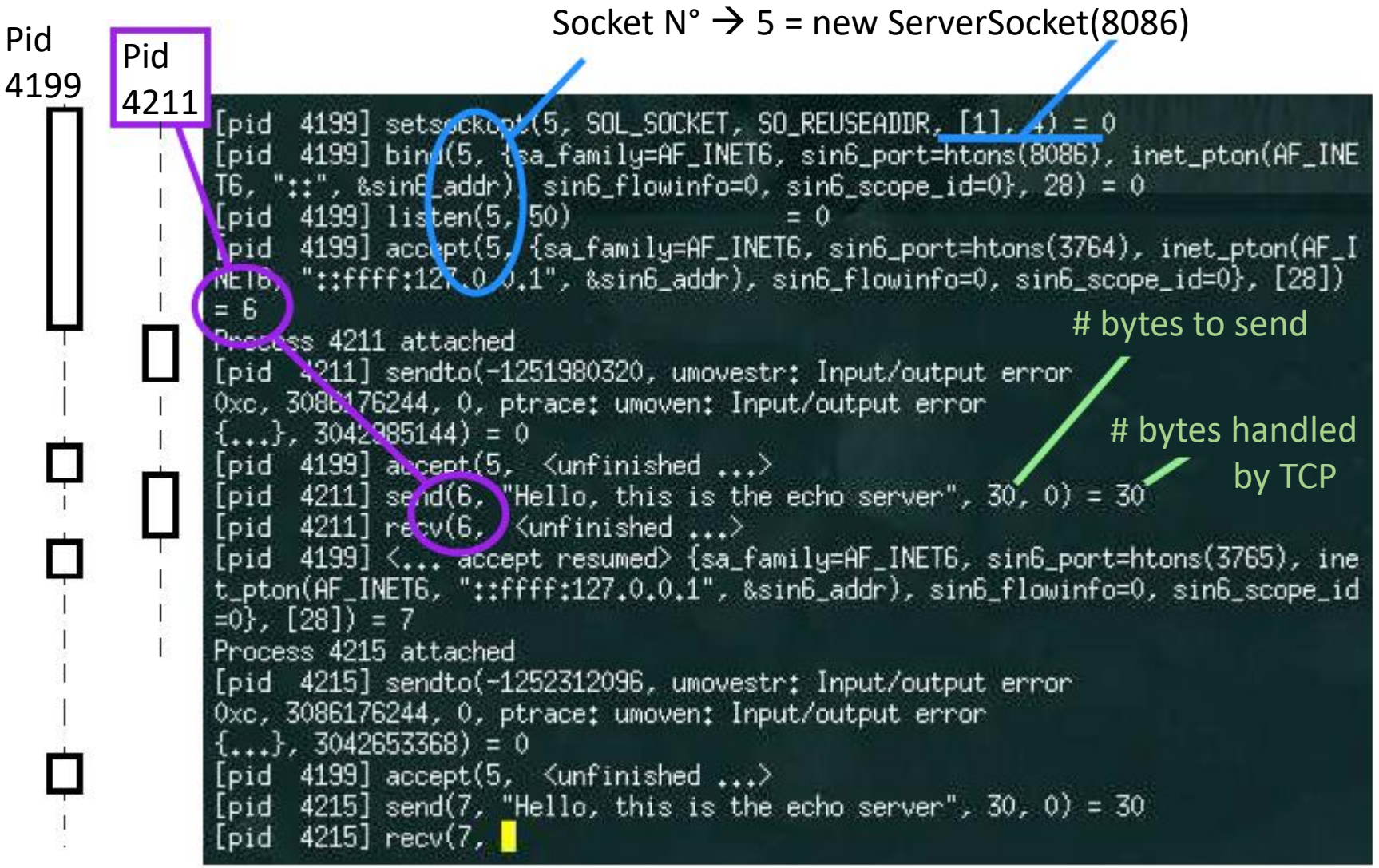
strace –enetwork –f java Server

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_INE
T6, "::", &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_I
NET6, "::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, 308617244, 0, ptrace: umoven: 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), ine
t_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, 308617244, 0, ptrace: umoven: Input/output error
{...}, 3042653368) = 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

strace –enetwork –f java Server



Some command lines

- (examples are better commented on the web).
- `javac t1.java` to compile
- `java Server` to launch
- `telnet localhost 8086` to test
- `strace -e trace=network -f java Server` to track system calls issued by your program
- `netstat -tlp` to list server sockets
- `netstat -tcp` to list running connections.
- `curl -I http://www.perdu.com` shows the headers for a HEAD method
- `curl http://www.perdu.com` dumps URL content on standard output.