Concurrent and Distributed Systems

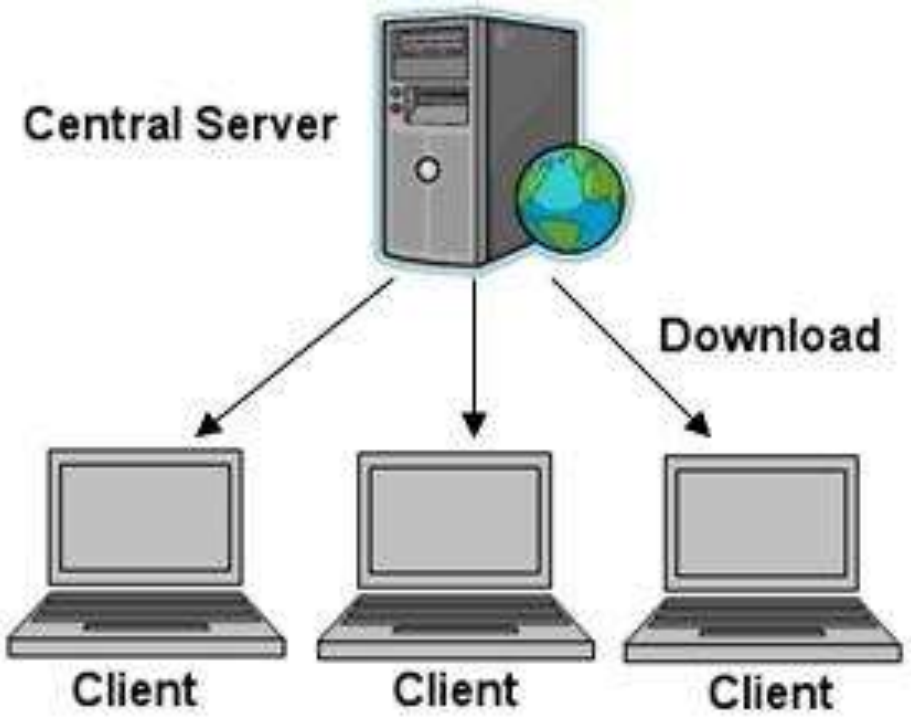
Talk Nine

Client-Server Programming with Sockets

Maxim Mozgovoy

Client-Server Architecture

Divides the tasks between service providers (servers) and service requesters (clients).



Client: connect to the server and request an operation or data.

Server: wait for the connection, then process client's request.

Examples: web browser / web server, PC / networked printer, computer game / game server, MySQL client / MySQL server.

Pro et Contra

Client-server systems appeared in 1970s and are still popular.

Advantages:

- Centralization (easier to setup and administer)
- Flexibility (easy to upgrade any server part)
- Interoperability (different systems work together easily)
- Security (sensitive data is not available outside the server)

Disadvantages:

- Dependability (the system critically depends on a server)
- Network congestion (too many clients overload a server)
- Suboptimal performance (clients' resources are not used)

Introducing Sockets

Sockets provide a natural way to organize a client-server architecture on most platforms. Sockets are standard objects, available through various libraries (in Java, they are contained in java.net package).

There are two types of sockets:
Connection-oriented (TCP): first establish the connection between the computers, then send data. This socket guarantees reliability: the data will arrive, and the order of packets will be preserved. Undelivered messages will be resent.

Connectionless (UDP): just send the data, and hope for the best. Used for fast, simple requests, and also for streaming music & video.

TCP Sockets

In our course we will deal with TCP sockets only.

They have two subtypes: client sockets and server sockets.

Client socket:
requests a connection
with the server

Server socket:
waits for a connection
request from a client

Establishing a Connection (1)

Sockets operate in TCP/IP networks, so each computer can be identified with its unique IP address (such as “66.249.89.99”).

If a computer has a DNS name,
it also can be used (“www.google.com”)

Some quick facts:

- Different computers cannot have the same IP in the same network.
- A computer can have several IP addresses and DNS names.
- The current machine always has an associated IP 127.0.0.1 and a DNS name localhost.
- You can display a list of IPs with ipconfig (Windows) / ifconfig (Linux, Mac) commands.

Establishing a Connection (2)

One computer can keep several connections at time (e.g. download a file with Firefox and chatting with Skype)

This is achieved with ports. Each computer has 65535 free ports that can be associated with sockets:

1 2 3 ... 65536

Establishing a Connection (3)

Creating a server socket: a port number should be specified (no need to tell IP because a computer knows its own IP)

Creating a client socket: an IP address / DNS name and a port number of a *remote server socket* should be specified

Connect to:

(myhost.com:12345)

Myhost.com

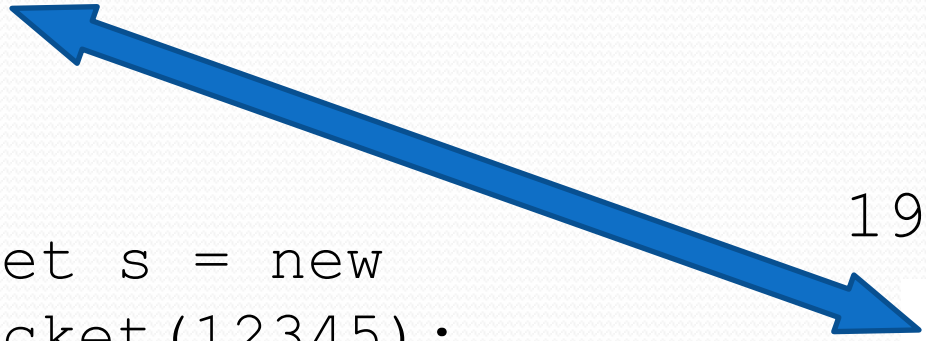
Listen on:

Port: 12345

Establishing a Connection (4)

Same with Java:

192.168.1.5



192.168.1.6

```
ServerSocket s = new
   ServerSocket(12345);
s.accept();
```

```
Socket s = new
   Socket("192.168.1.5", 12345);
```

When the connection is ready, you can send the data both ways!

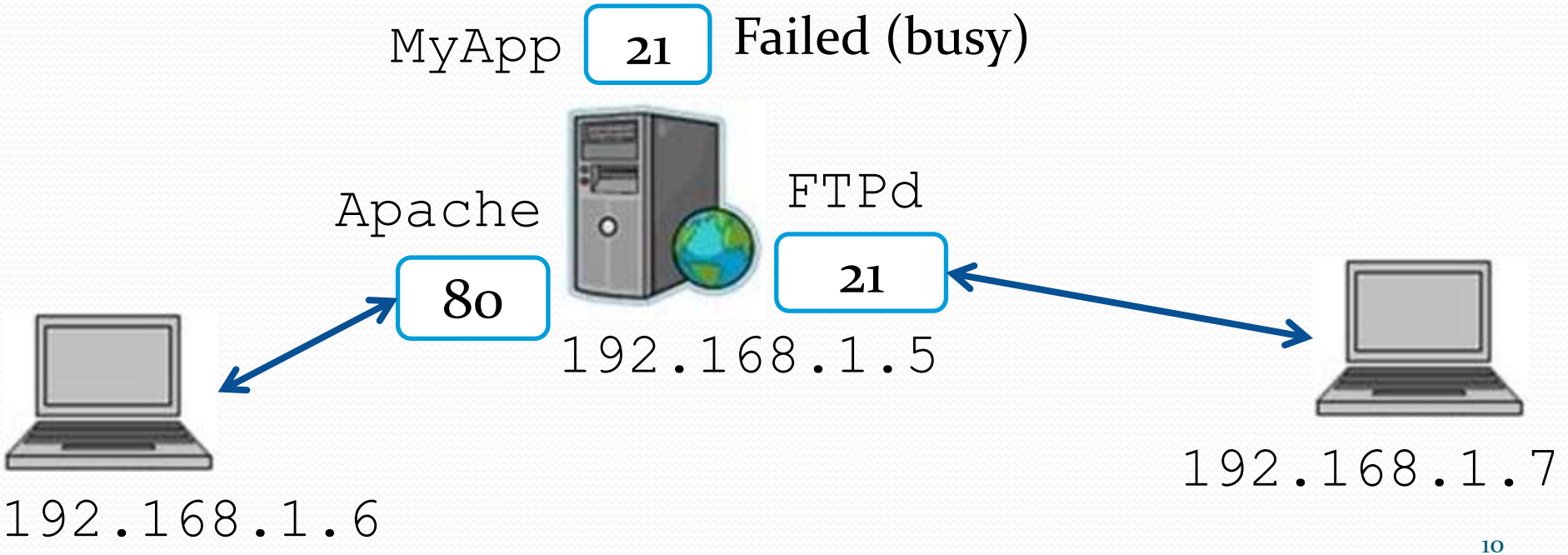
Some Notes

Ports [0, 1023] are reserved for well-established purposes.

Ex: 80: HTTP/Web; 25: SMTP/send mail; 110: POP3/get mail

Software tools can conflict if they use the same ports.

So the connection port is usually configurable.



Sockets in Java (1)

```
class java.net.Socket: Client socket

// make a socket and connect it to <host, port>
// throws an exception if connection has failed
Socket(String host, int port)

void close() // close the connection (disconnect)

InputStream getInputStream()    //get read stream
OutputStream getOutputStream() //get write stream

// the streams are standard
// (same as the ones we use to read/write files)
```

Sockets in Java (2)

```
class java.net.ServerSocket: Server socket

// make a socket, bound to <port>
ServerSocket(int port)

// wait for the incoming connection
// when the connection is made, return a
// new client socket, which should be used to
// read/write data
Socket accept()

// The original socket can be used again
// for new connections!
```

Continuous Listening

```
// to organize independent data communication
// sessions, we use threads:
class DataExchange implements Runnable {
    public DataExchange(Socket c) { … }
    public void run() { /* send/receive */ }
}
…
void Serve() { // continuously await clients
    ServerSocket s = new ServerSocket(12345);
    for(;;) {
        Socket c = s.accept();
        new Thread(new DataExchange(c)).start();
    }
}
```

Example 1: Reading a Web page (1)

Let's try first a client socket:

there are many ready servers we can test!

E.g.: How to read a web page:

1) Connect to a server to the port 80.
2) Send a HTTP command of a form

```
GET <resource> HTTP/1.0
Host: <host>
<blank line>
```

3) Receive data from the server.

Example 1: Reading a Web page (2)

```
import java.net.*;
public class w09_readHttp {
  public static void main(String args[]) {
    try {
      Socket s = new
                 Socket("en.wikipedia.org", 80);
      String cmd = "GET /wiki/Main_Page";
      cmd += "HTTP/1.0\n";
      cmd += "Host:en.wikipedia.org\n\n";

      s.getOutputStream().write(cmd.getBytes());
      s.getOutputStream().flush();  // make sure
                                    // to flush
```

Example 1: Reading a Web page (3)

Now let's print the webpage we got:

```
        int c;
        while((c=s.getInputStream().read()) != -1)
          System.out.print((char)c);
      }
      catch(Exception e) {
        System.out.println("Connection error!");
      }
    }
}
```

Result: first comes response header (starts with HTTP/1.0 200 OK), then one blank line, then the page body.

See complete code in w09_readHttps.java

Example 1: Reading a Web page (4)

The code

```
Socket s = new Socket("en.wikipedia.org", 80);
```

worked fine for HTTP connections.

Now nearly all sites use HTTPS, so you have to modify it:

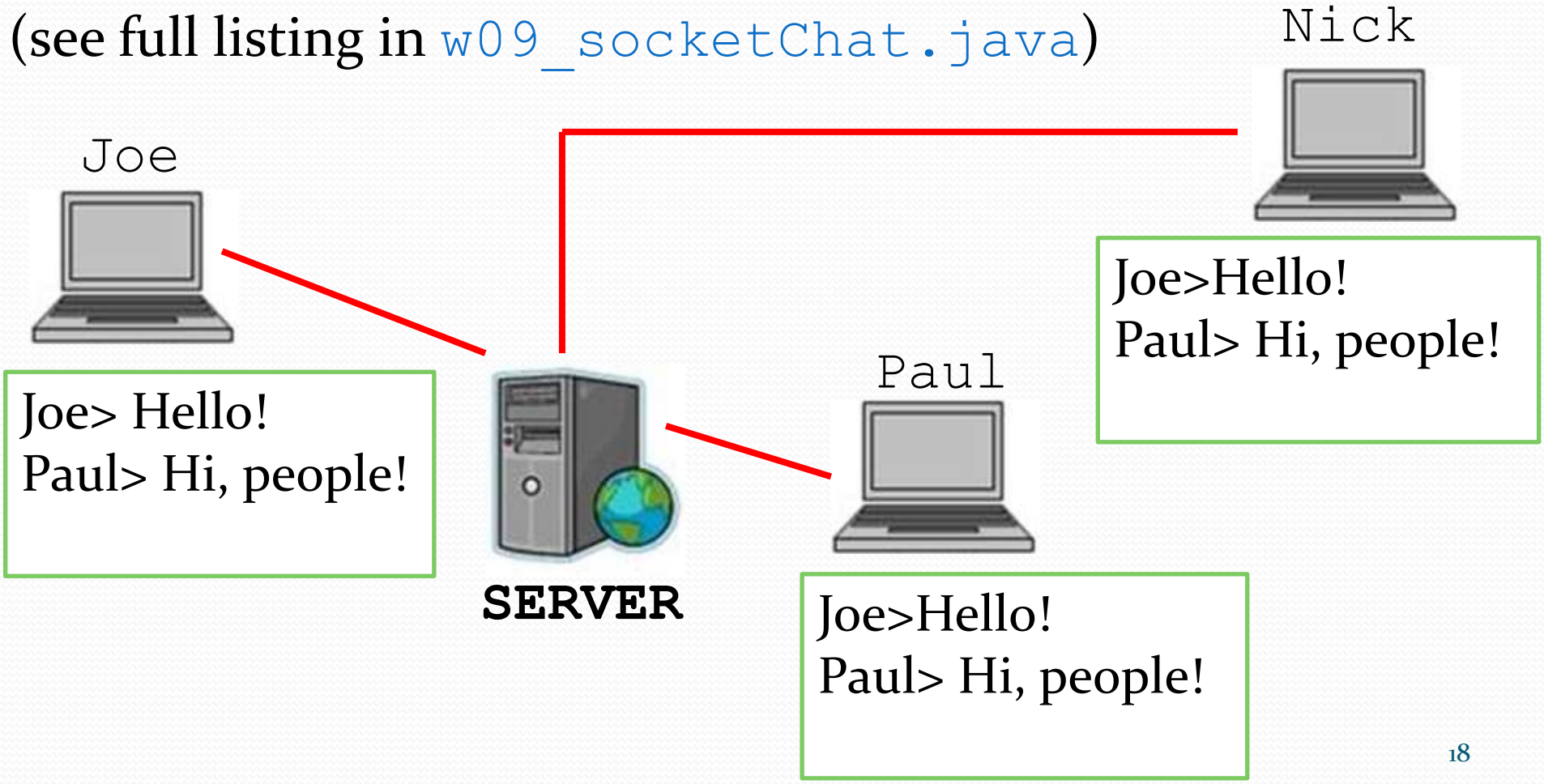
```
import javax.net.ssl.SSLSocketFactory;
...
Socket s = SSLSocketFactory.getDefault().
           createSocket("en.wikipedia.org", 443);
```

Note: a different port is used!

Example 2: A Network Chat

A network chat is a real distributed system, where participants (clients) communicate by through a centralized server

(see full listing in `w09_socketChat.java`)



A Network Chat (2)

Helper: prints everything that appears in the stream reader

```
class EchoIncoming implements Runnable {
   BufferedReader reader;
   ...
   public void run() {
      try {
         for(;;)
            System.out.println(reader.readLine());
      }
      catch(Exception e) {}
   }
}
```

A Network Chat (3)

Helper: handles all sockets in the system

```
class AllSockets
  static private Vector<Socket> allSockets =
                                    new Vector<Socket>();
  synchronized static public void
  Add(Socket client) { allSockets.add(client); }

  synchronized static public void
  Broadcast(Socket from, String str)
                                    throws Exception {
    for(Socket s : allSockets)
      if(s != from) SendStringTo(str, s);
  }                    // don't broadcast to itself!
}
```

A Network Chat (4)

Helper: broadcasts all messages coming from the client to all other clients

```
class Broadcast implements Runnable {
   Socket client;
   ...
   public void run() {
      BufferedReader in = …; // get client stream
      for(;;) {
         String msg = in.readLine();  // read next
         AllSockets.Broadcast(client, msg);
      }
   }
}
```

A Network Chat (5)

Server:

```
ServerSocket s =
            new ServerSocket(port); // e.g. 12345

for(;;)
{
    System.out.println("listening");
    Socket client = s.accept();
    AllSockets.Add(client);
    new Thread(new Broadcast(client)).start();
}
```

A Network Chat (6)

Client:

```
Socket s = new Socket(host, port);
                        // e.g. "127.0.0.1", 12345

BufferedReader r = …; // input stream of s
new Thread(new EchoIncoming(r)).start();

for(;;)
{
    read_line_from_the_console();
   send_line_to_the_server();
}
```

A Network Chat (7)

Threads:
Listen()
Broadcast(Joe)
Broadcast(Paul)
Broadcast(Nick)

SERVER

AllSockets:
[Joe, Paul, Nick]

Paul

Threads:
SendKeyboard()
EchoIncoming()

Joe

Threads:
SendKeyboard()
EchoIncoming()

Nick

Threads:
SendKeyboard()
EchoIncoming()

Notes

- The server keeps a list of all clients (AllSockets vector).
- Access to AllSockets is synchronized to avoid simultaneous element addition and for-loop through the elements.
- The server always listens for new connections.
- Also the server runs a Broadcast() thread for each client c. This thread sends all incoming messages from c to others.

- Each client runs a thread that sends incoming keyboard input to the server.
- Also it runs an EchoIncoming() thread that displays messages received from other clients in real time.

Conclusions

- Classic client-server architecture can be built with sockets.
- Note that P2P systems also can be implemented with sockets (in P2P, each host serves both as a client and as a server).
- Also with sockets you can access existing Internet infrastructure: web-servers, e-mail servers, FTP, etc.
- Normally, you should establish a connection between a client socket and a server socket.
  Then you can send data in both ways!
- The underlying TCP protocol automatically handles low-level issues with data transfer (ensure correct order of packages, resend undelivered messages, select the best transfer speed).
- Don't forget to flush streams when you send the data over the network to ensure that the data block was actually sent.
- In Java, sockets are built-in.