EE 382N: Distributed Systems **Fall 2017**

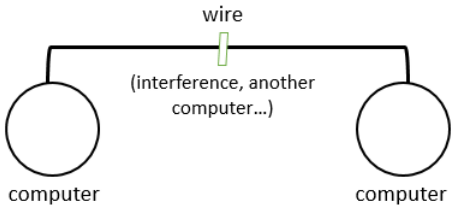
Lecture 2: August 18

Lecturer: Vijay Garg *Scribe: Audrey Addison*

2.1 Introduction

This lecture covered material from Chapter 6, specifically, the User Datagram Protocol (UDP), the Transmission Control Protocol (TCP), and Java Remote Method Invocation (RMI).

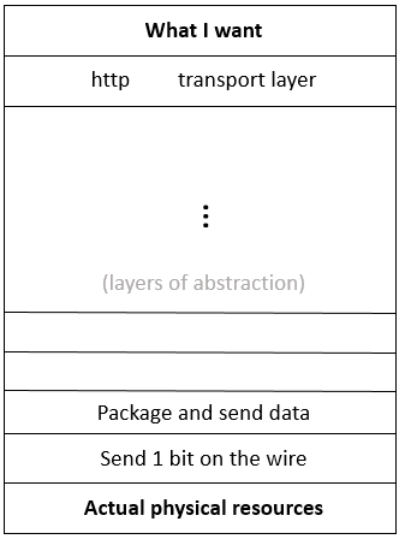
Flash back - How to change the world?
Thinking back to the early days of computing, the motivation for the lecture was: How do we connect 2 computers and change the world!? To begin, we supposed we had two computers connected by wires.



Key observations
The physical world is not ideal.
We could have messages passing through another machine (congestion).
We may need to worry about the route the messages take.
Many things can go wrong.

2.1.1 Layered software

To help with this problem, the approach is to design your software in layers. At the top, we have "What I want", and the bottom, we have "actual physical resources". In between, we have many layers. We start from the bottom with "actual physical resources" and then incrementally improve the interface (reaching greater abstraction) as we progress through the layers from bottom to top until we reach the "what I want".



For example, we may begin by sending 1 bit on the wire (that is a layer). Then in the next layer, we may send one link or a group of bytes, and so on.

This approach is modular. Individual layers can be swapped (provided the interfaces at the edges are consistent). This is useful if you have multiple ways of doing the same thing.

2.2 Communicating with a server

Flash back to modern day. Now we have a distributed system and want to send a message.

Q: Who does this message go to?

A: We can specify this with the IP address of the machine (which could have multiple users) and the Port #.

Q: How do we actually communicate?

A: By using either TCP or UDP protocols.

2.2.1 User Datagram Protocol (UDP)

Think of this like the US Postal Service. You send a message by putting it in a package (envelope), giving it a TO address (IP address and port) and a FROM address so the receiving server knows where the message came from.

Data is sent as "**Datagram packets**".

This method is considered "**not reliable**" in that one doesn't know if the message has been received.

But, it is **fast**. It is **scalable** (no need to keep 'per session' state). Could handle millions of users.

How it works: there is a computer listening on a port (and some device to pick out "bad packets") and anyone can send a packet. As you send messages, they may arrive in order but - depending on the route -

some packets could arrive out of order. UDP does not inherently preserve message order (**not FIFO**).

2.2.2 Transmission Control Protocol (TCP)

Think of this like the phone (**session oriented**) or a pipe. It is "**reliable**" (messages are acknowledged as they are received). But, there is a need to keep state for the session creating **more overhead** than UDP (e.g., slower).

TCP will slow you down if you try to send too much stuff (when the buffer is full). There are lots of parameters one can adjust. TCP **is FIFO** (it does preserve message order).

2.2.3 Other protocols

There is a Reliable UDP as well as many other protocols built upon TCP and UDP. Almost everything is written on top of one of these.

2.3 Implementing Protocols in java

2.3.1 UDP Implementation

See Datagram Packet (java class).

Two methods:
- send()
- receive()

To start, we'll implement an "echo server" (whatever message the server receives, it sends it back to the client).

See: DatagramClient.java

client creates
- two DatagramPackets (declared on line 8, defined on line 22, 24), one for sending data (need hostname and port #), one for receiving
- one DatagramSocket (a UDP socket, line 15)

We use two methods from the DatagramPacket:
- send (non-blocking call, line 23)
- receive (blocking call, line 25)

Receive is on the server side, it is a **blocking call** meaning it can get stuck, or "hang", until it receives a packet.

Class asides:
- One could define a non-blocking receive.
- We see the send call is made on line 23 and receive on line 25. In reality, many things could happen between

these calls.

See: DatagramServer.java

The server doesn't have to know who the client is until something is received. Lines 17-18 give you the return address info.

2.3.2 TCP Implementation

For TCP, there are two types of sockets:
- ServerSocket (created by server using IP address and port #)
- then make blocking call while you wait for a call ("accept")
- Socket (per session, established by the client)

Server will maintain a table (public class NameTable)
- table contains process name, IP address and port #
- table is searchable and editable by the client (see line 11, 19, 31)

"synchronized" keyword because server is multithreaded.

"getSocket()" call
- establishes connection with server
- socket is TCP Socket
- symbols — server name and port (line 8)
- get 1 input stream (read)
- get 1 output stream (write)

(think of these as an input pipe and an output pipe)

2.4 RMI

These methods work well, but the programmer does have to do a lot of work packing and unpacking strings, parsing them... We would like to abstract away some of the message passing.

Q: Can we do better?
Why do we have to accept a string and then unpack and parse it?

A: Remote Procedure Call (RPC), then... Remote Method Invocation (RMI)
With these, the compiler does the "marshalling" and "unmarshalling"
We have the notion of a "remote object" (e.g., the server table) with an interface (methods you can call)

Code example:
NameRMIClient: (NameServiceImpl.java?)
Client side:
- find the remote object (using the rmiregistry daemon which maintains list of registered objects)

- line 7 - port # is implicit
- just want a handle to the object (r is a reference, the object lives on the server side)

Also have to worry about security (line 22)

NameService:
- need this interface on both client and server side to compile
- throws Remote Exception

2.4.1 Parameter passing

If you have a function that sends a reference to an object (from client side to server side or vice versa), you will have trouble. That reference doesn't mean anything on the other machine.

Calling the function: foo(object p)
2 solutions:
1) serialize the object p (inefficient if p is large, complicated if p also contains references within itself)
2) make p also remote

Calling the function: foo(int z)
- send primitive types by value

Overall, be careful with parameters.
- send primitive types by value
- if an object, it can be local or remote
- send local objects by serialization - beware of overhead on large objects
- send remote objects by proxy

2.5 To do:

Read the class notes.

Security Policy File (specified by user, outlines constraints for the security manager to enforse)
- google Java RMI tutorial (Oracle)

Be aware there are many other types of protocols: http, web services...