CSCI-1680
Sockets and network programming

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, David Mazières, Phil Levis, John Jannotti

Administrivia

- Container setup: fill out form by TONIGHT (9/12)
  - Whether or not you have it working

Snowcast is out!

- Gearup Today 9/12 5-7pm CIT368 (+Zoom, recorded)
  - Look at the notes!
- Milestone due by Monday, 9/16 by 11:59pm EDT
  - Warmup + design doc

Topics for Today

- Working with sockets
- TCP & UDP
- Building a protocol

Sockets: Communication Between Machines

- Network sockets are file descriptors!
- UDP (“datagram sockets”)

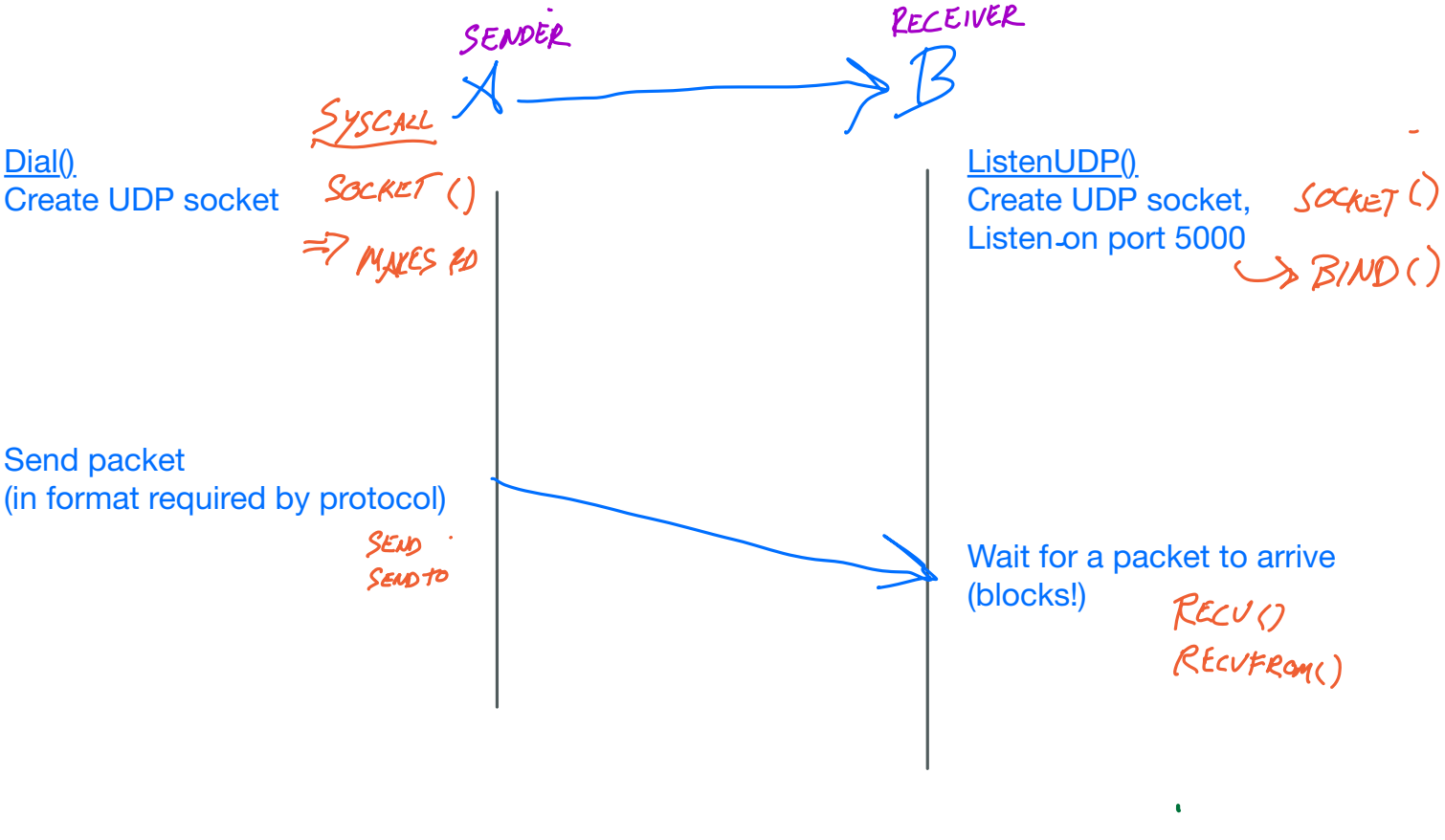
  => Connectionless: *unreliable* message delivery

- TCP (“stream sockets”)
  - *Reliable, connection-oriented...*

UDP Example

Last time we ended on talking about the differences between UDP and TCP. We're going to see a bit more of that, and then we're going to start building an application.

Let's go back to our UDP example: we had a sender and a receiver



Where does this API come from?
This is defined by the OS interface, and most OSes follow something similar to the "Berkeley socket API". In Linux, the core operations are system calls, and every language wraps the system calls in some way. Here are those syscalls

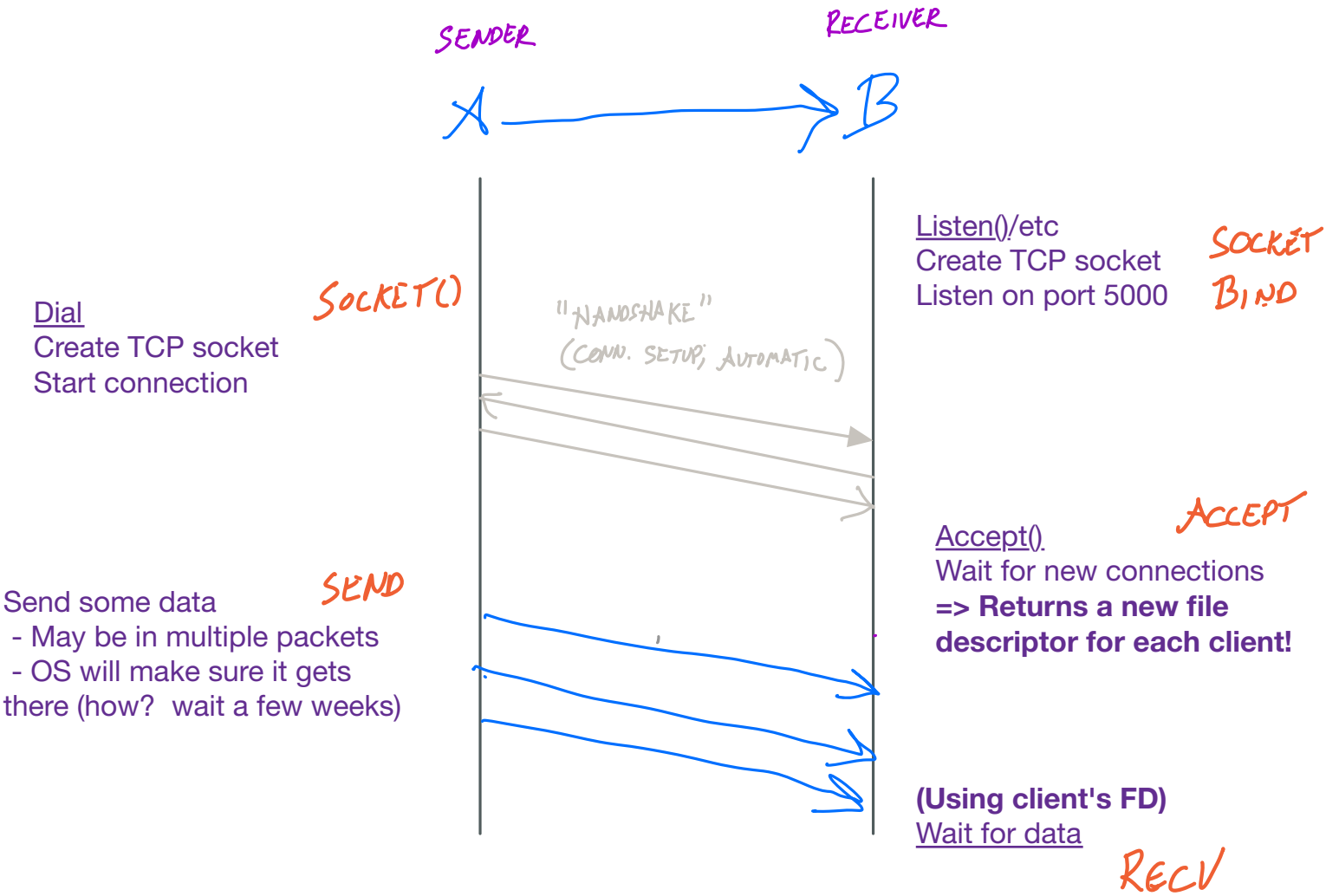
=> Regardless of what language you use, or what new languages come about, you want to think about this in terms of the system calls. Because when you interact with the OS networking stack, no matter how much pretty stuff your language adds, you're always working in terms of the system calls

	UDP	TCP
Reliability	Unreliable: Don't know if data arrived at its destination	Reliable (we'll see how later)
How connections work	"Connectionless" => *Can send even with no receiver online!*	"Connection-oriented" Unique socket for each client connection (eg. A<->B, C <-> B)
How data is sent	"Datagram service" => Order doesn't matter => Sending discrete things Better when latency matters, or you don't care about the data if it's late (video call)	"Reliable, in-order, byte stream" => Data you send can be any size (TCP will reconstruct it in-order on the other end)

WE'LL LEARN MORE ABOUT THIS LATER!

TCP Example

=> Has concept of "connections" between client and server
=> Reliable protocol (will retry packets)

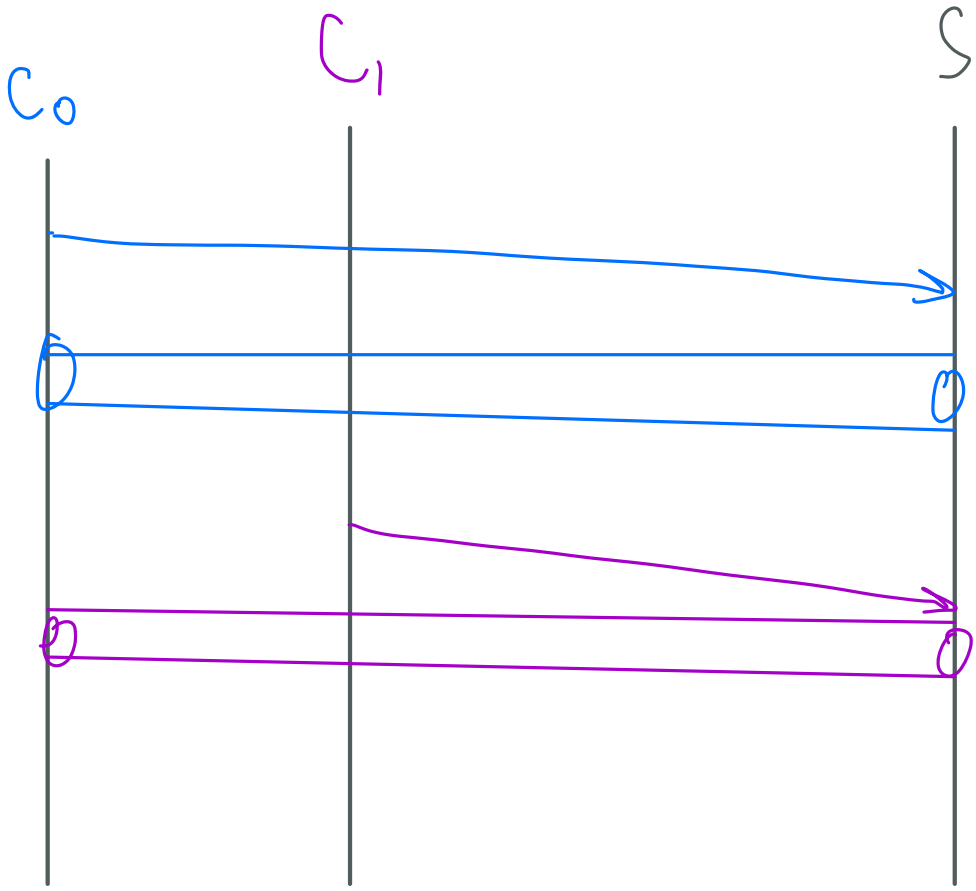


Some key differences with TCP

=> Each client connection gets its own socket on the server => can be used to hold a long-term communication between the server and one client (can send a lot of bytes over multiple packets, just between A and B (like a pipe for each)

=> How does this work? Accept returns a NEW SOCKET for each client that connects

=> OS will make sure data is delivered reliably (or will return error)

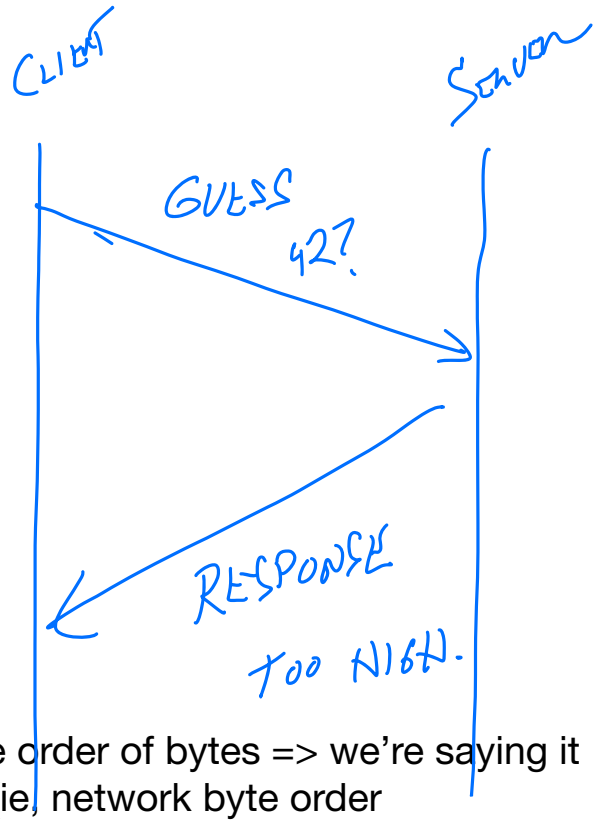


In TCP: each client gets its own connection--you can think of this as a unique "pipe" with which the server can communicate to each client independently.

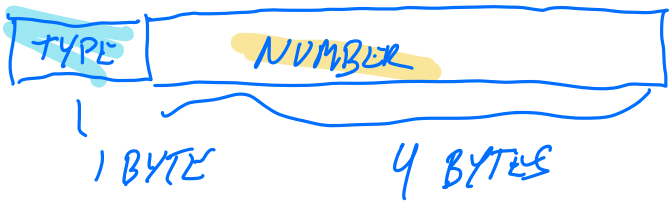
In this way, TCP lets us send large amounts of data (which can't fit into a single packet), since the protocol will ensure that it all arrives correctly, and in-order (eg. sending files, web traffic, etc.)

Client-server example: Guessing game

Server picks a random number
Clients connect and can guess numbers
Server responds with too high, too low, or correct
First client to respond wins, restarts game



As the designers, we get to decide on the format for how messages are exchanged Here's our format. In this version, every message is 5 bytes:

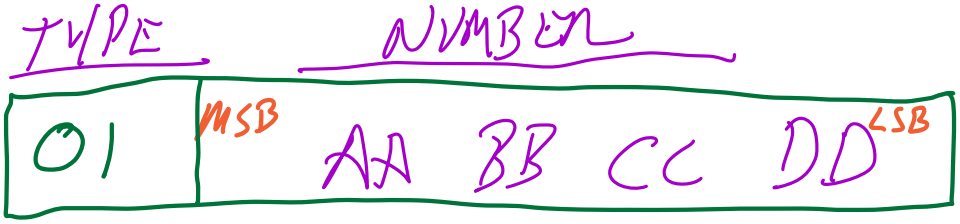


Protocol must give the order of bytes => we're saying it should be big endian (ie, network byte order

GUESS
TYPE = 0
NUMBER = GUESS

RESPONSE
TYPE = 1
NUMBER = 1 TOO HIGH
0 CORRECT!
-1 TOO LOW

When we format the message as a byte array, we order each field as in the picture above: first the type, then the number. For multi-byte data like integers, our protocol needs to specify the byte order (ie, the endianness) used to send the data "over the wire". In our protocol, we'll use big endian, or "network byte order." If our guess were the number 0xaabbccdd, we'd format it like this:



BIG ENDIAN (NETWORK BYTE ORDER)

INDEX (IN BYTE ARRAY) 0 1 2 3 4

0 DD CC BB AA

LITTLE ENDIAN

In Go, we specify the byte order when marshaling the struct. In C, you would need to convert the fields of your struct using helpers like ntohs(), htons(), etc, before casting your struct to a byte array and sending it.

Demo: guessing game

Sockets: Communication Between Machines

- Network sockets file descriptors!
- Datagram sockets (eg. UDP): unreliable message delivery
  - Send atomic messages, which may be reordered or lost
- Stream sockets (TCP): bi-directional pipes
  - *Stream* of bytes written on one end, read on another
  - Reads may not return full amount requested, must re-read

System calls for using TCP

Client	Server
	`socket` – make socket
	`bind` – assign address, port
	`listen` – listen for clients
`socket` – make socket	
`bind*` – assign address	
`connect` – connect to listening socket	
	`accept` – accept connection

- This call to bind is optional, connect can choose address & port.

Socket Naming

- TCP & UDP name *communication endpoints*
  - IP address specifies host (128.148.32.110)
  - 16-bit port number demultiplexes within host
  - Well-known services listen on standard ports (*e.g.* ssh – 22, http – 80, mail – 25)
  - Clients connect from arbitrary ports to well known ports

- A connection is named by 5 components
  - Protocol, local IP, local port, remote IP, remote port

Dealing with Data

- Many messages are binary data sent with precise formats

- Data usually sent in Network byte order (Big Endian)
  - Remember to always convert!
  - In C, this is htons(), htonl(), ntohs(), ntohl()