ECE 650
Systems Programming & Engineering

Spring 2019

Programming with Network Sockets

Rabih Younes
Duke University

Slides are adapted from Brian Rogers and Tyler Bletsch (Duke)

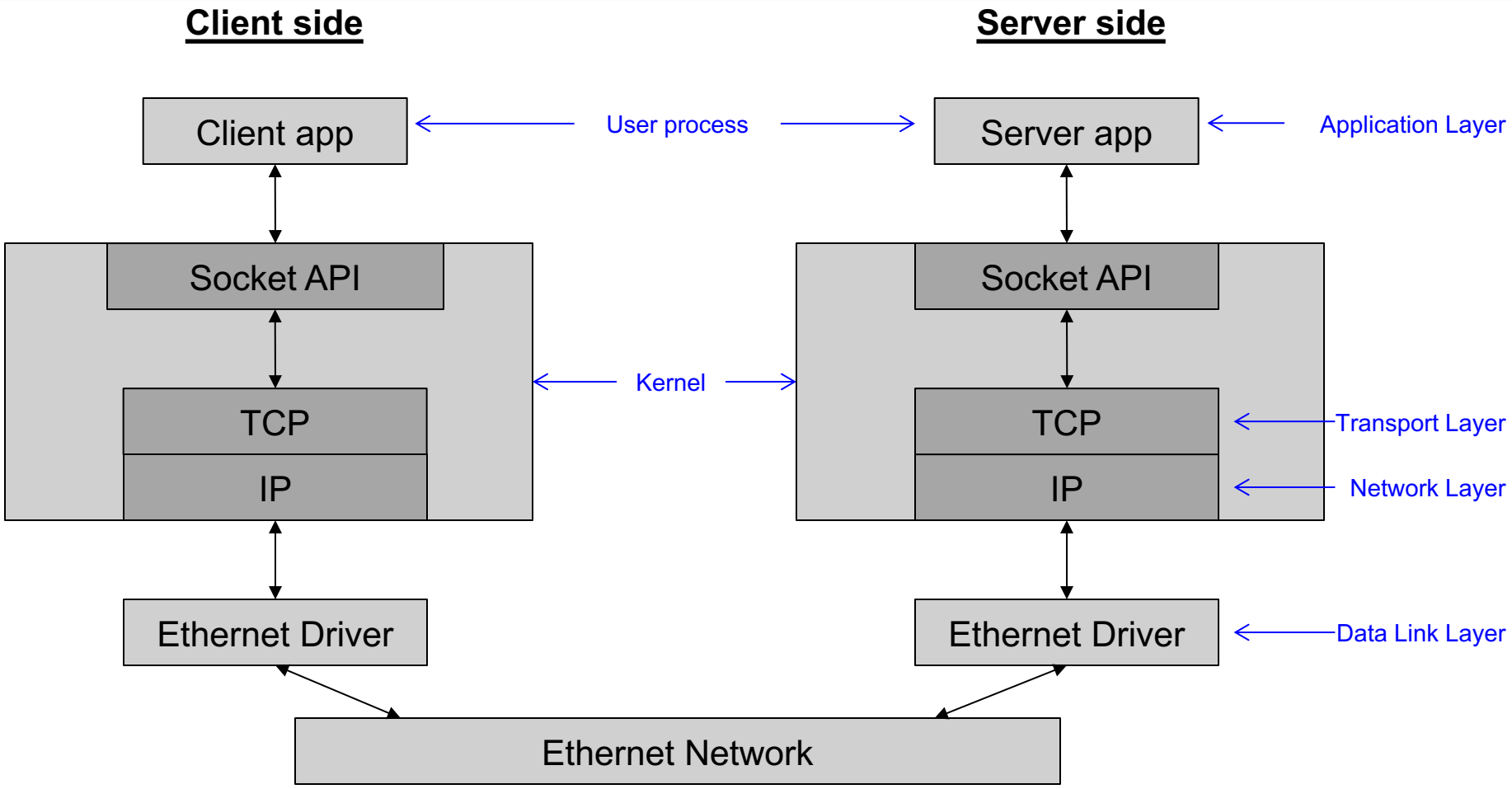
Sockets

- We've looked at shared memory vs. message passing
  - All on a single system (meaning running under a single OS)
- What about communication across distributed processes?
  - Running on different systems
  - Assume systems are connected by a network (e.g. the internet)
- We can program using **network sockets**
  - For creating connections and sending / receiving messages
  - Often follows a client / server pattern
- We will assume basic network knowledge
  - E.g. what is an IP address
  - We will cover the networking stack in more detail in next lectures

Client-Server Model

- Common communication model in networked systems
  - Client typically communicates with a server
  - Server may connect to multiple clients at a time
- Client needs to know:
  - Existence of a server providing the desired service
  - Address (commonly IP address) of the server
- Server does not need to know about the existence of clients and their addresses
  - It waits for service requests from clients
  - It responds using the addresses in the request packets
    - The source address in the client's request would become the destination address in the server's response

Client-Server Overview



- Client and Server communicating across Ethernet using TCP/IP

TCP – Connection-Oriented Service

- **Transmission Control Protocol**
  - Designed for end-to-end byte stream over unreliable network
  - Robust against failures and changing network properties
- TCP transport entity
  - E.g., Library procedure(s), user processes, or a part of the kernel
  - Manages TCP streams and interfaces (OSI layer 4) to the IP layer (OSI layer 3)
  - Accepts user data streams from processes
  - Breaks up data into pieces that can fit in 1 Ethernet frame (w/ IP + TCP headers)
  - Sends each piece separately as IP datagram
  - Destination machine TCP entity reconstructs original byte stream
  - Handles retransmissions & re-ordering
- **Connection-oriented** transport layer
  - Provides error-free, reliable communication
  - Can think of communication between two processes on different machines as just like UNIX pipes or fifos
    - One process puts data in one end, other process takes it out

Network Sockets

- Network interface is identified by an IP address
  - Or a hostname, which translates into an IP address
  - E.g., 127.0.0.1 (same as "localhost") or login.oit.duke.edu
- Interface has 65536 ports (0-65535)
- Processes attach to ports to use network services
  - Port attachment is done with `bind()` operation
- Allows application-level multiplexing of network services
  - E.g., SSH vs. Web vs. Email use different ports
  - Many ports are standard (e.g., 80 for web server, 22 for SSH)
  - You may have seen URLs like http://127.0.0.1:4444
    - 127.0.0.1 is the IP address
    - 4444 is the port number

TCP Service Model

- TCP service setup:
  - Two endpoint processes create endpoints (sockets)
  - Each socket has an address:
    - IP address (32 bit for IPv4 or 128 bits for IPv6)
    - Port number (16 bits)
  - API functions used to create & communicate on sockets
- Port numbers:
  - Numbers below 1024 called "well-known ports"
    - Reserved for standard services, like FTP, HTTP, SMTP
      http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml
    - But not all services usually used & active all at once
      - Don't want them all active, just waiting for incoming connections
    - Special daemon: inetd (Internet daemon)
      - Attaches to multiple ports
      - Waits for incoming connection
      - fork()'s of the new, appropriate process to handle that connection

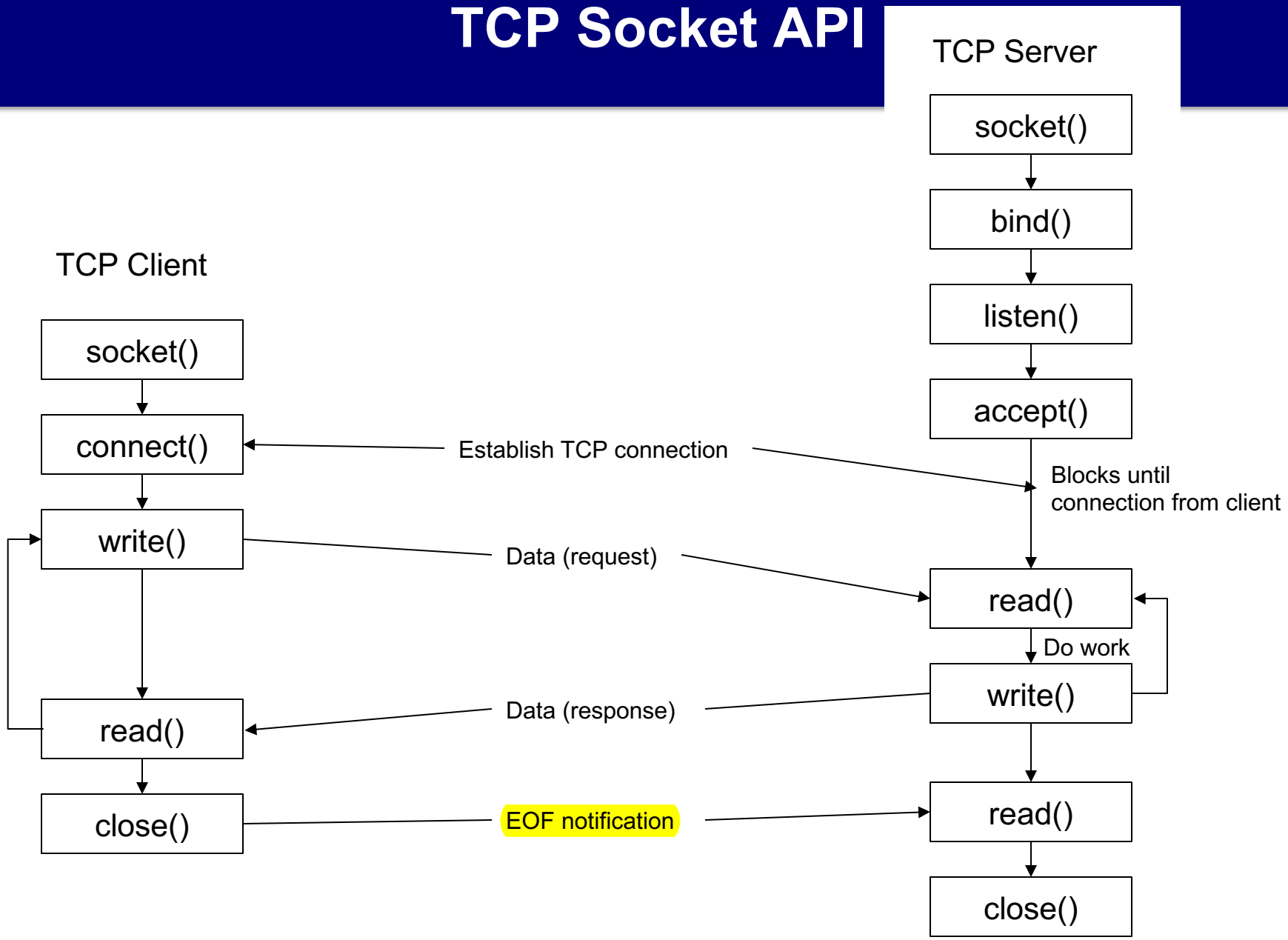
(Still around, but less common nowadays)

UNIX TCP sockets

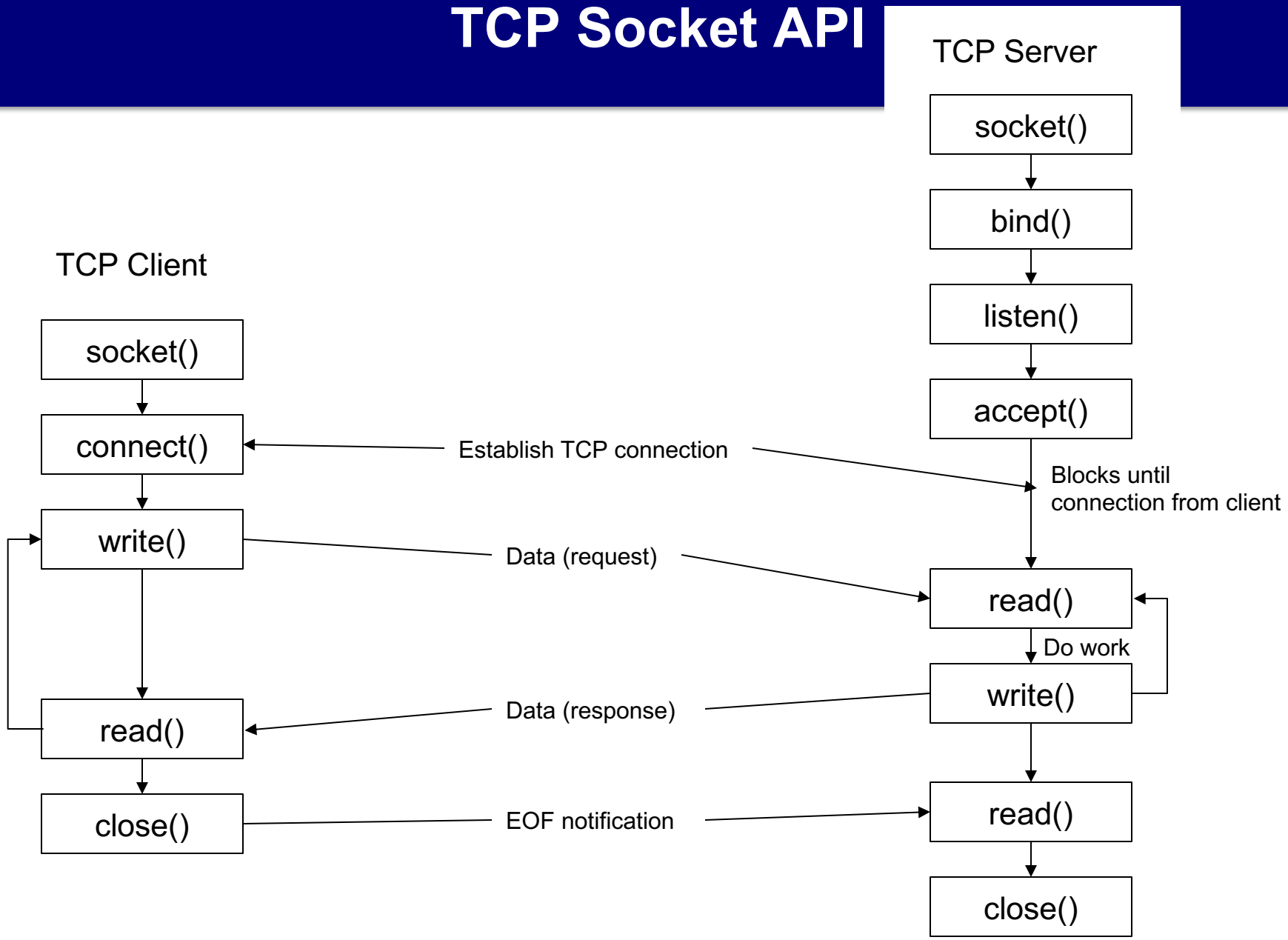
- Here is a great reference for use of socket-related calls
  - http://beej.us/guide/bgnet/

Primitive	Meaning
`socket()`	Create a new communication end point
`bind()`	Attach a local address to a socket
`listen()`	Announce willingness to accept connections; give queue size
`accept()`	Block the caller until a connection attempt arrives
`connect()`	Actively attempt to establish a connection
`send()`	Send some data over the connection
`recv()`	Receive some data from the connection
`close()`	Release the connection

TCP Socket API



TCP Socket API



Server-Side Structure

- Often follows a common pattern to serve incoming requests

```
pid_t pid;
int listenfd, connfd;
listenfd = socket(...);

/***fill the socket address with server's well known port***/

bind(listenfd, ...);
listen(listenfd, ...);

for ( ; ; ) {

   connfd = accept(listenfd, ...); /* blocking call */

   if ( (pid = fork()) == 0 ) { /* create a child process to service */

      close(listenfd); /* child closes listening socket */

      /***process the request doing something using connfd ***/
      /* ................. */

      close(connfd);
      exit(0);  /* child terminates
    }
    close(connfd);  /*parent closes connected socket*/
  }
}
```

Example

- tcp_example
  - server.cpp
  - client.cpp

- Read readme.txt first to know how to execute the files