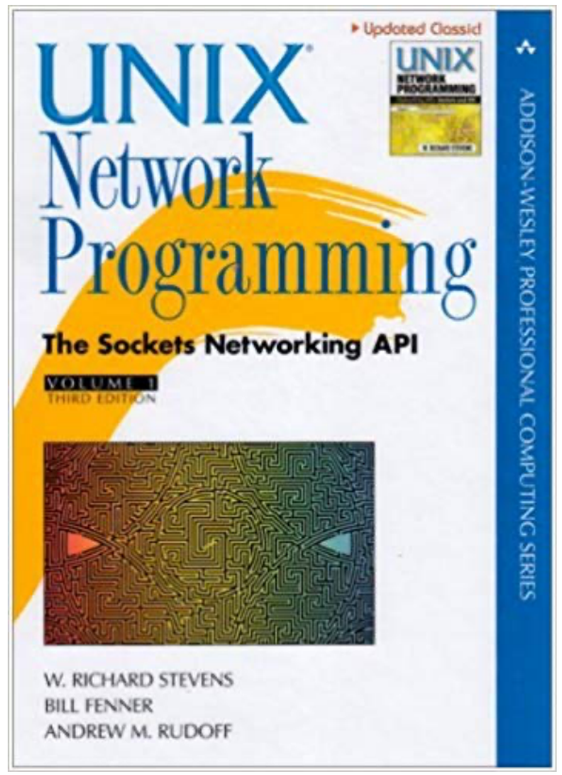
Operating System Concepts

Lecture 10: Sockets

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

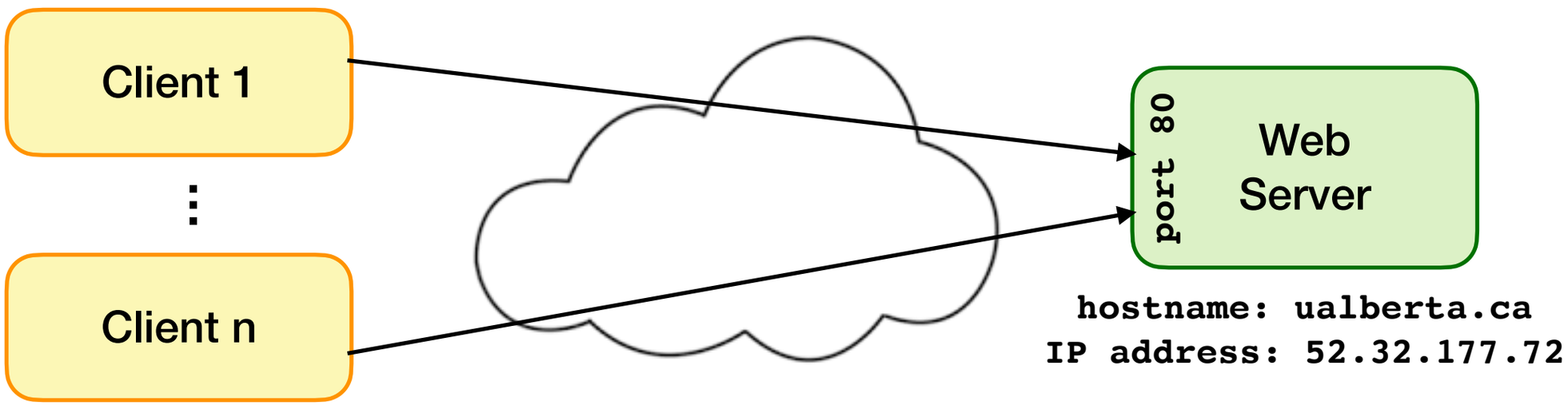
Today’s class

- Interprocess communication with sockets
  - socket families
  - POSIX.1 socket API
  - client/server example



The client-server model

- One of the most common models for structuring distributed computing
- Server: a process or collection of processes that provide a service, e.g., name service, file service, database service
- Client: a process that requests this service
  - many clients typically access a server



Socket

- An abstraction of a network I/O queue
- One endpoint of a connection
  - each communication endpoint is identified by an IP address and a port number
- Socket allows bidirectional communication
- Communication between a pair of processes requires a pair of sockets
  - communication over a network requires a pair of **network sockets**
  - communication on a local machine can be done using a pair of **UNIX domain sockets**

Socket

- There are two common types of sockets
  - stream sockets: support connection-oriented, reliable, duplex communication under the stream model (no message boundaries)
  - datagram sockets: support connectionless, best-effort (unreliable), duplex communication under the datagram model (message boundaries)
- Both types support a variety of address domains, for example:
  - INET domain: useful for communication between processes running on the same or different machines that can communicate using IP
  - UNIX domain: useful for communication between processes running on the same machine
    - more efficient than INET domain sockets for processes running on the same machine
  - Domains are defined in `sys/socket.h`

Socket creation

```
int socket(int domain, int type, int protocol)
```

- returns a **socket descriptor** or -1 on error
- socket domains (address families) specified by POSIX.1
  - IPv4 Internet domain: `AF_INET`
  - IPv6 Internet domain: `AF_INET6`
  - UNIX domain: `AF_UNIX` (or `AF_LOCAL`)
- socket types specified by POSIX.1
  - connectionless message (`SOCK_DGRAM`), connection-oriented byte-stream (`SOCK_STREAM`), connection-oriented message (`SOCK_SEQPACKET`),…
- socket protocol: UDP, TCP, ICMP, IP, IPV6, …
  - set to 0 to select the default protocol for the given socket domain and type

Socket descriptor

- Socket descriptor is a file descriptor in UNIX
  - calling `socket( )` is similar to calling `open( )` as it returns a file descriptor; in both cases, you have to call `close( )` to free up the file descriptor when you are done
  - but you cannot use all system calls which are being used with file descriptors, e.g., `lseek( )` does not work
  - `read(fd, readbuf, readlen)` and `write(fd, writebuf, writelen)` system calls can be used to work with a socket descriptor
    - but there are socket specific system calls for read and write too
  - a socket can be duplicated using the `dup( )` system call
  - a socket can be disabled for reading, writing, or both in one direction or both directions using the `shutdown( )` system call

Datagram socket (`SOCK_DGRAM`)

- No need to establish a connection first
  - connectionless service
- Send a message addressed to the socket of the target machine
  - so a message might be lost
  - if you send multiple messages, the delivery order is not guaranteed

Stream socket (`SOCK_STREAM`)

- Must setup a connection first between the two sockets
  - just like making a phone call
- When the connection is established, the two computers can communicate with each other
- Byte-stream: applications are unaware of message boundaries
  - reading the same number of bytes written may need several function calls
- How to use reliable message-based instead of byte-stream service?
  - use `SOCK_SEQPACKET`; in this case, the same amount of data that was originally written is received

Socket creation — examples

```
#include <sys/socket.h>

int sockfd;


// TCP provides reliable connection-oriented byte stream
sockfd= socket(AF_INET, SOCK_STREAM, 0);
// sockfd= socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
// UDP provides unreliable connectionless datagram
sockfd= socket(AF_INET, SOCK_DGRAM, 0);
// sockfd= socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

Addressing

- How to identify the process with which you wish to communicate?
  - host name (mapped to network address, i.e. IPv4 or IPv6 in standard dot notation)
  - port number represents a process on that computer
- Use the `getaddrinfo(name, portnumber, ...)` function to obtain a list of `addrinfo` structures, each struct contains a local address (`ai_addr`) which can be assigned to a socket using `bind( )`
  - `name` can be the host name, or IPv4 or IPv6 address

```
struct addrinfo {
  int ai_flags;
  int ai_family;                /* indicates the socket domain */
  int ai_socktype;              /* indicates the socket type */
  int ai_protocol;              /* indicates the protocol
  socklen_t ai_addrlen;
  struct sockaddr *ai_addr;     /* contains IP address and port number */
  char *ai_canonname;
  struct addrinfo *ai_next;
};
```

Binding to a specific address

- No address (port number) is assigned to a socket created using the `socket( )` system call
- The `bind( )` system call associates a local address with a socket; this is necessary before a connection-oriented socket **receives** connections
  - the port number in the address cannot be less than 1024 (unless it is called by superuser)
  - if we specify the special IP address `INADDR_ANY`, the socket endpoint will be bound to all network interfaces of the system
- If you don't care which port to use, you may not call bind and leave this to the OS to pick a port when `listen( )` or `connect( )` is called
- The `getsockname( )` system call can be used to discover the address bound to a socket

What identifies a connection?

- A 5-tuple **uniquely** identifies a connection
  - source IP address
  - source port number
  - destination IP address
  - destination port number
  - protocol (TCP, UDP, etc.)
- Client port number is usually assigned randomly by OS
  - so no need to call `bind( )` on the client side
- Server port number is usually a well-known port, e.g., 80 for HTTP

Accepting connection

How does the server know a client wants to make a connect request?

- the server should call `listen( )` to start allowing clients to connect
  - converts an unconnected socket into a **listening socket** (passive socket)
- the `listen( )` system call takes the socket descriptor along with an integer defining the number of outstanding connect requests that should be queued by the kernel on behalf of the process
  - if the queue is full, new connect requests will be rejected
- the `accept( )` system call is then used to create a new socket (**connection socket**) for a particular client connection
  - it will **block** until there is a pending connect request unless the socket descriptor is in nonblocking mode
  - it returns a new socket descriptor for the connection socket. The connection socket is different from the listening socket created by `bind( )` and passed to `listen( )` which remains available to receive other connect requests

Establishing connection

- For connection-oriented network services (like TCP), we need to establish a connection between the client's socket and the server's socket
  - the `connect( )` system call creates a connection, i.e., connects the socket to the specified remote socket address
  - if no address is bound to the caller's socket, connect binds a default address
- Connection may not be created (`connect( )` returns -1) if
  - the target machine is not up and running
  - the target machine is not bound to the address we are trying to connect to
  - there is no room in the target machine's connect queue

Data transfer

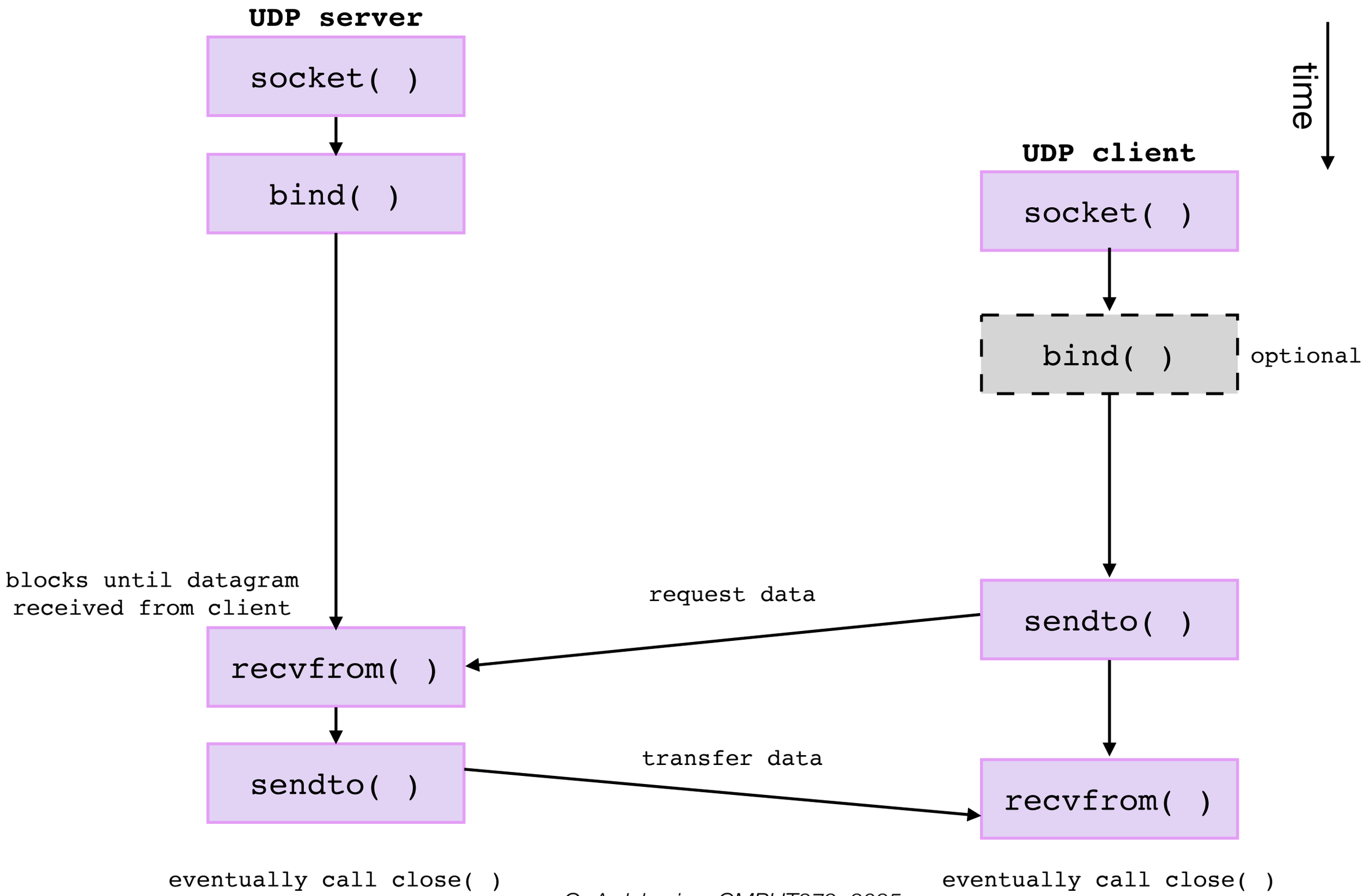
- `read( )` and `write( )` can be used to communicate with a socket, as long as it is connected (for `SOCK_STREAM` or `SOCK_SEQPACKET` only)
  - can be used together with the `poll( )` or `select( )` system call to wait for the descriptor to become **ready** for I/O
  - but we can't specify options if you use them, so we typically use **socket-specific functions** instead

- Socket-specific functions for sending data
  - `send( )` is similar to `write( )` but takes flags
    - with a byte stream protocol `send( )` blocks until the entire amount of data has been transferred
  - `sendto( )` is similar to `send( )` but takes the destination address for connectionless sockets
    - the destination address is ignored for connection-oriented sockets

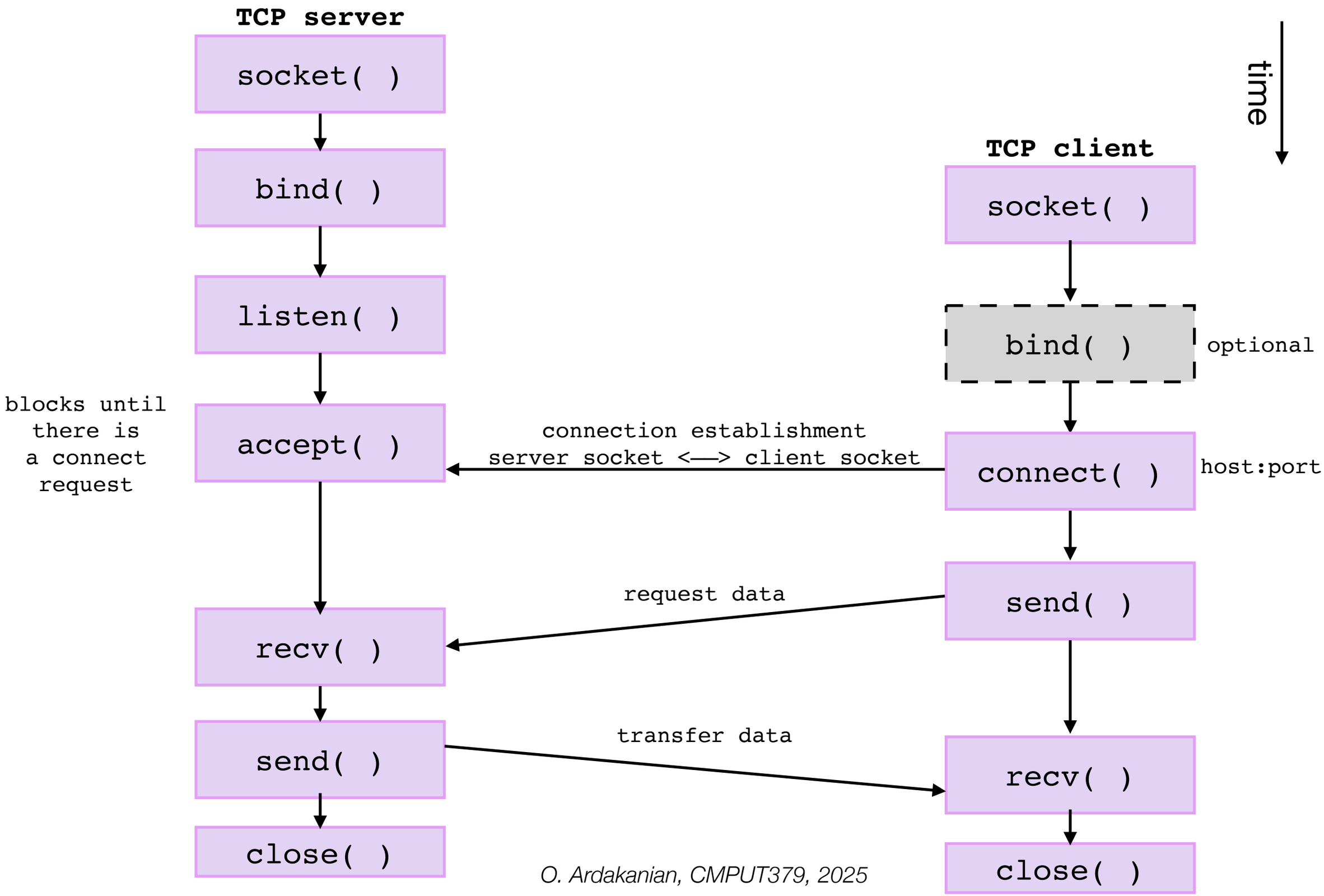
Data transfer

- Socket-specific functions for receiving data
  - recv( ) is similar to `read( )` but takes flags
    - with a byte-stream protocol, `recv( )` might receive less data than requested; use `MSG_WAITALL` flag to prevent `recv( )` from returning until the amount data requested has been received
  - `recvfrom( )` is similar to `recv( )` but takes the source address for connectionless sockets
    - the source address is ignored for connection-oriented sockets
- Trying to send or receive data on a broken socket causes a SIGPIPE to be generated

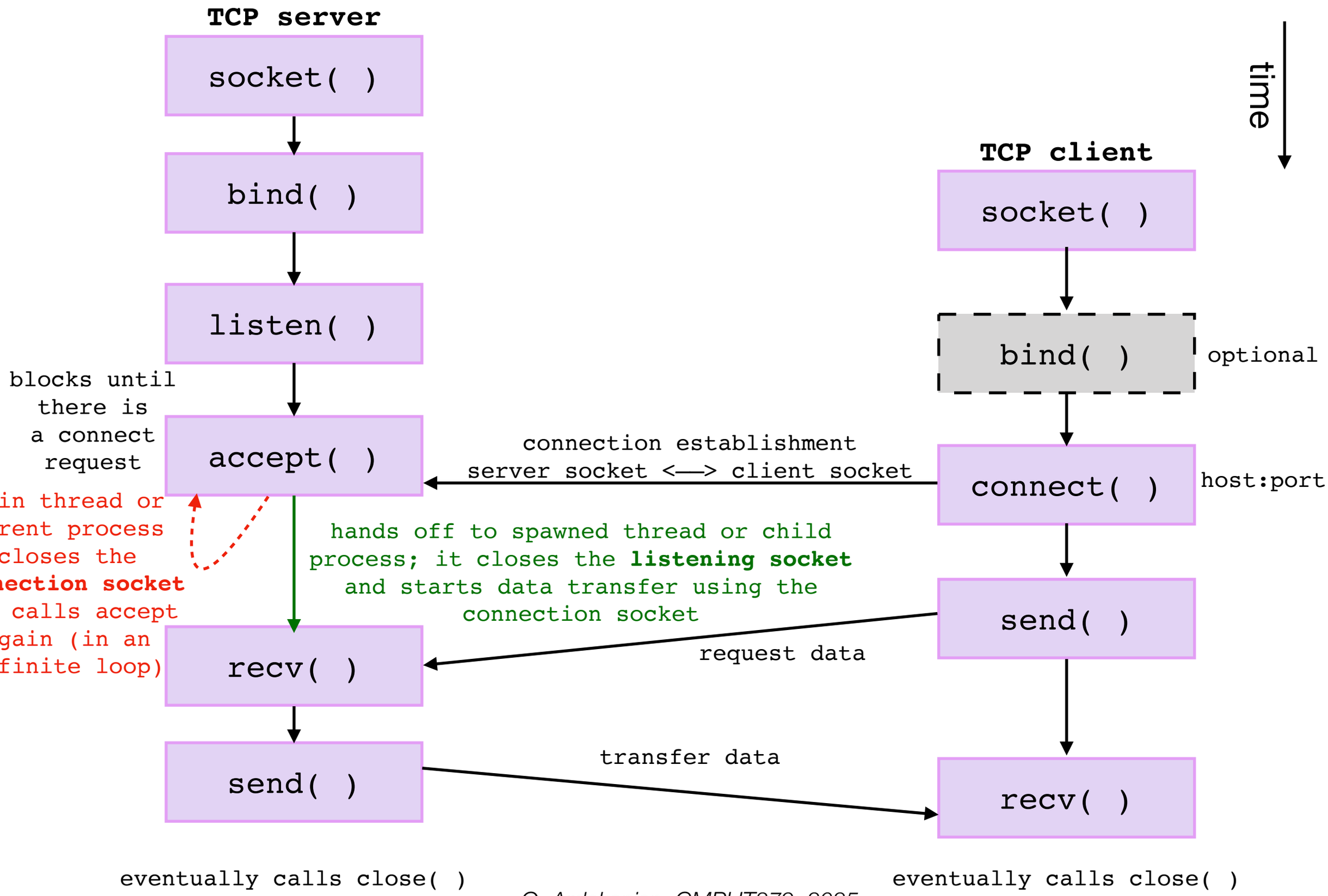
Client-Server communication over UDP



Client-Server communication over TCP



Concurrent server



Homework

- Implement a concurrent TCP server!