Operating System Concepts

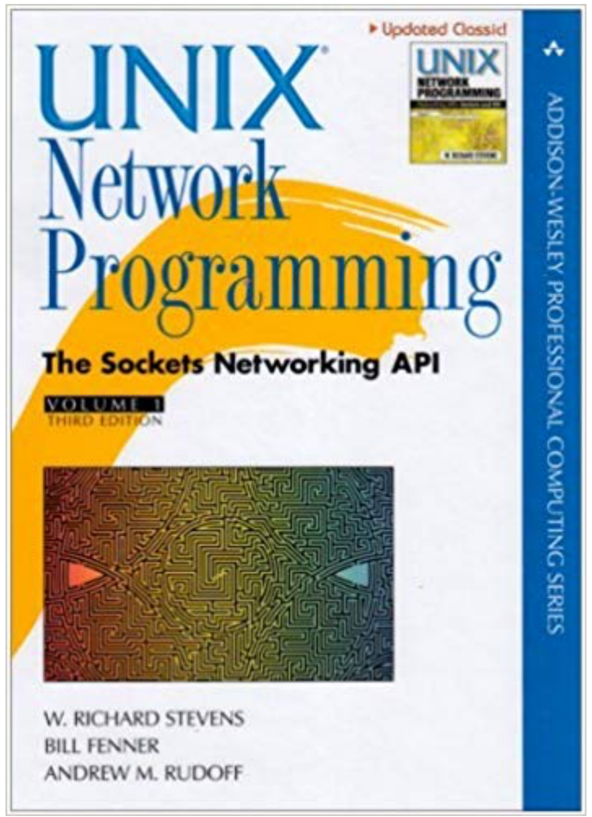
Lecture 11: Communication across the Network

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today's class

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



Client-Server communication

- Definition: one of the most common models for structuring distributed computation

Client-Server communication

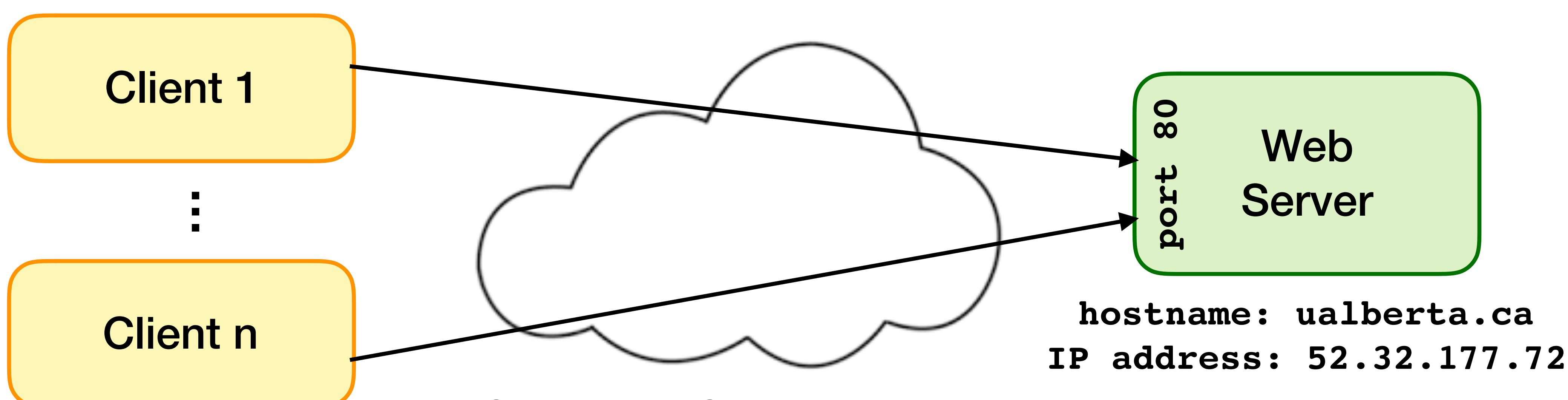
- Definition: one of the most common models for structuring distributed computation
- a server is a process or collection of processes that provide a service, e.g., name service, file service, database service
  - the server may exist on one or more nodes

Client-Server communication

- Definition: one of the most common models for structuring distributed computation
- a server is a process or collection of processes that provide a service, e.g., name service, file service, database service
  - the server may exist on one or more nodes
- a client is a program that uses the service
  - many clients typically access a common server

Client-Server communication

- Definition: one of the most common models for structuring distributed computation
- a server is a process or collection of processes that provide a service, e.g., name service, file service, database service
    - the server may exist on one or more nodes
- a client is a program that uses the service
    - many clients typically access a common server



Socket

- a socket is an abstraction of a network I/O queue (first introduced in 4.2 BSD)

Socket

- a socket is an abstraction of a network I/O queue (first introduced in 4.2 BSD)
- a socket is one endpoint of a connection
  - each communication endpoint is identified by an IP address and a port number

Socket

- a socket is an abstraction of a network I/O queue (first introduced in 4.2 BSD)
- a socket is one endpoint of a connection
  - each communication endpoint is identified by an IP address and a port number
- a pair of processes communicating require **a pair of sockets**
  - communication over a network requires a pair of network sockets
  - communication on a local machine requires 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)

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 support a variety of address domains, e.g.,
  - INET domain: useful for communication between process running on the same or different machines that can communicate using IP protocols
  - 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

Socket creation

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

Socket creation

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

- returns a socket descriptor or -1 on error

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 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 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
  - `read(fd, readbuf, readlen)` and `write(fd, writebuf, writelen)` system calls can be used to work with a socket descriptor
  - a socket can be duplicated using the `dup( )` system call
  - but you cannot use all system calls which are being used with file descriptors, e.g., `lseek( )` doesn't work
  - 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 used by the target machine
  - the message might get lost
  - if you send multiple messages, the order of delivery is not guaranteed

Stream socket (`SOCK_STREAM`)

- must setup a connection first between the two sockets
  - just like making a phone call

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 (bidirectionally) with each other

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 (bidirectionally) with each other
- byte-stream: applications are unaware of message boundaries
  - reading the same number of bytes written may need several function calls

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 (bidirectionally) with each other
- byte-stream: applications are unaware of message boundaries
  - reading the same number of bytes written may need several function calls
- want to use message-based instead of byte-stream service?
  - use `SOCK_SEQPACKET` in this case the same amount of data is received as it was originally written
  - Stream Control Transmission Protocol (SCTP) provides a sequential packet service

Socket creation — examples

```
int sockfd;

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

// for UDP socket
// 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, IPv4 or IPv6, in standard dot notation)
  - port number represents a process on that computer

Addressing

- how to identify the process with which you wish to communicate?
  - host name (mapped to network address, 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

Addressing

- how to identify the process with which you wish to communicate?
  - host name (mapped to network address, 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

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 may **receive** 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 the system's network interfaces

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 may **receive** 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 the system's network interfaces
- if you don't care about which port to use, you may not call bind and leave this to the OS to pick a port

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 may **receive** 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 the system's network interfaces
- if you don't care about which port to use, you may not call bind and leave this to the OS to pick a port
- the `getsockname( )` system call can be used to discover the address bound to the specified socket

Accepting connection

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

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 passive (listening) socket

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 passive (listening) 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

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 passive (listening) 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
  - the connection socket is different from the **listening socket** created by `bind( )` and passed to `listen( )`; it remains available to receive additional 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

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

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.)

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 the OS
  - no need to call `bind( )`

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 the OS
  - no need to call `bind( )`
- server port number is usually a well-known port, e.g., 80 for HTTP

Data transfer

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

Data transfer

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

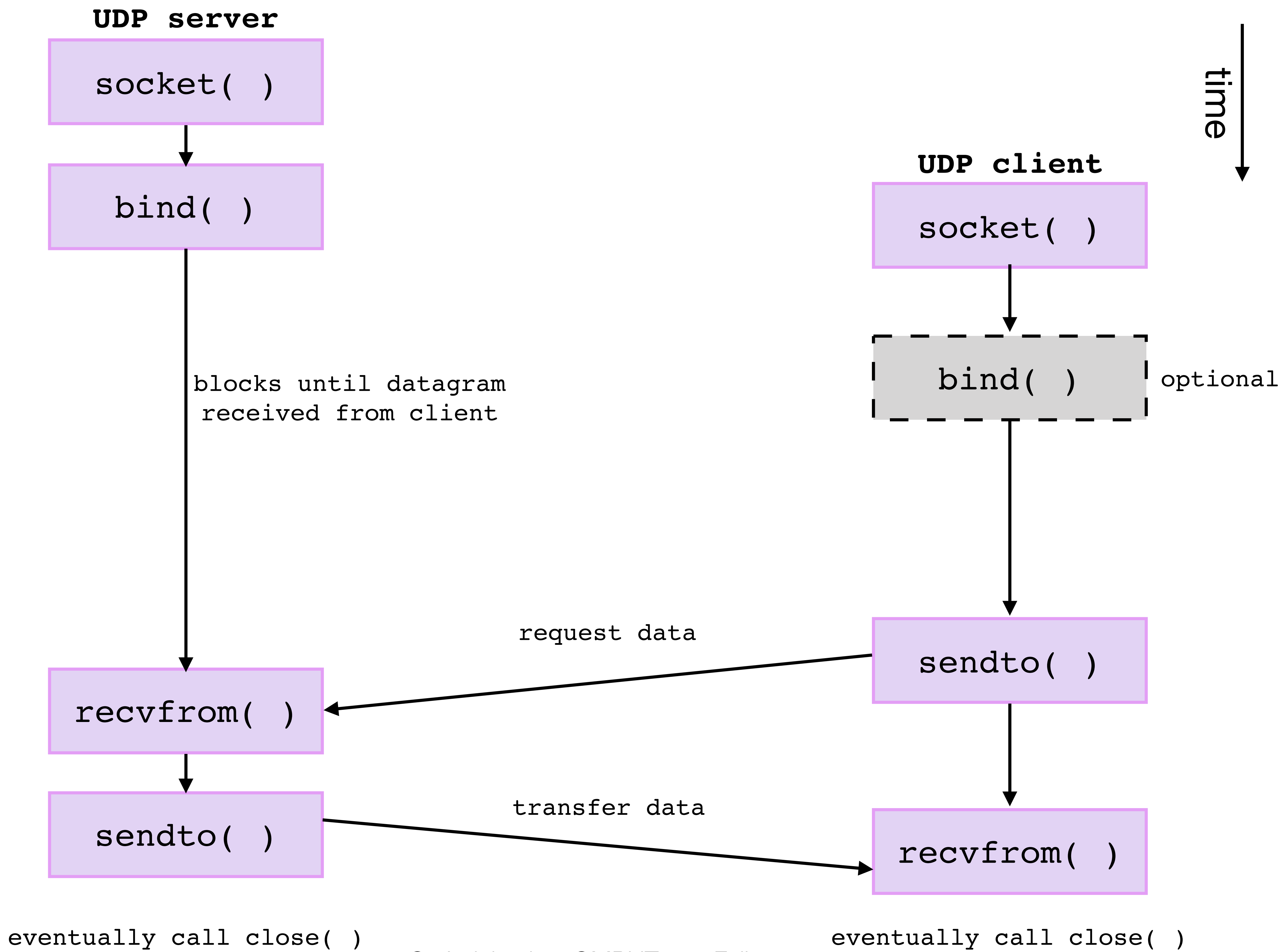
Data transfer

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

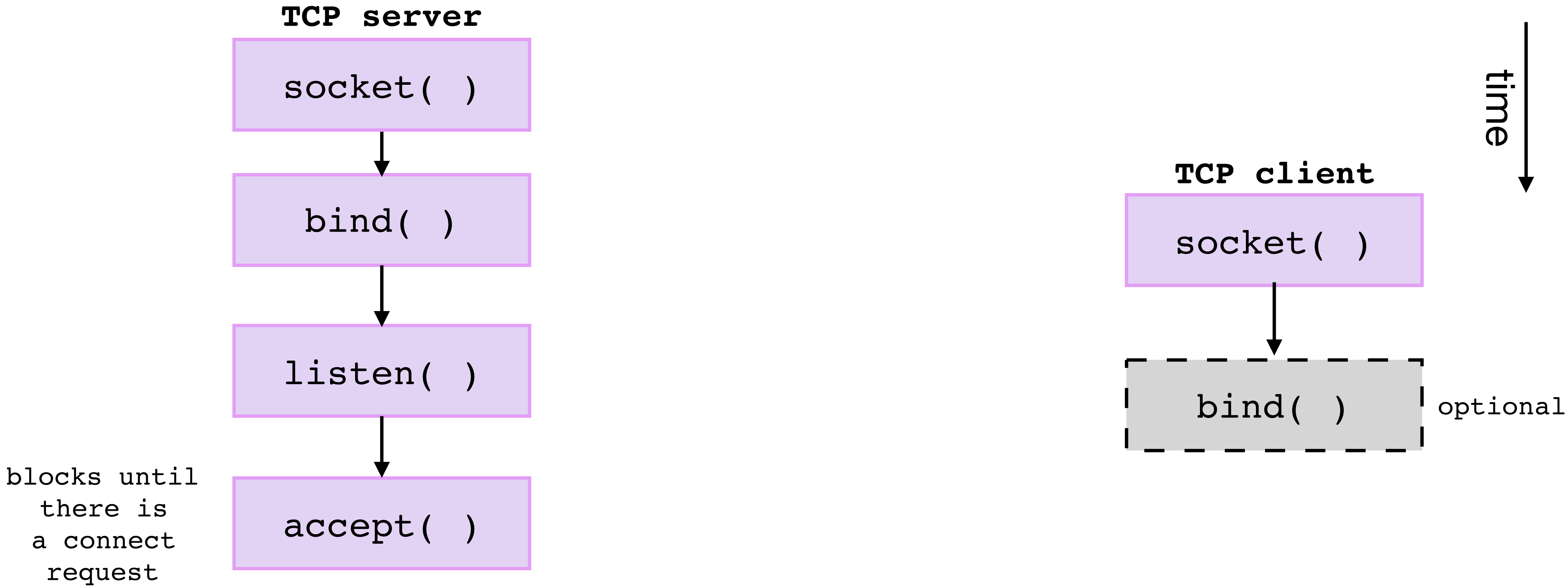
Data transfer

- socket functions for receiving data
  - recv( ) is similar to `read( )` but takes flags
    - with a byte-stream protocol, `recv( )` can receive less that than we requested; use `MSG_WAITALL` flag to prevent `recv( )` from returning until the data we 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
  - recv`msg( )` is similar to `readv( )` as you can specify multiple buffers to receive data into

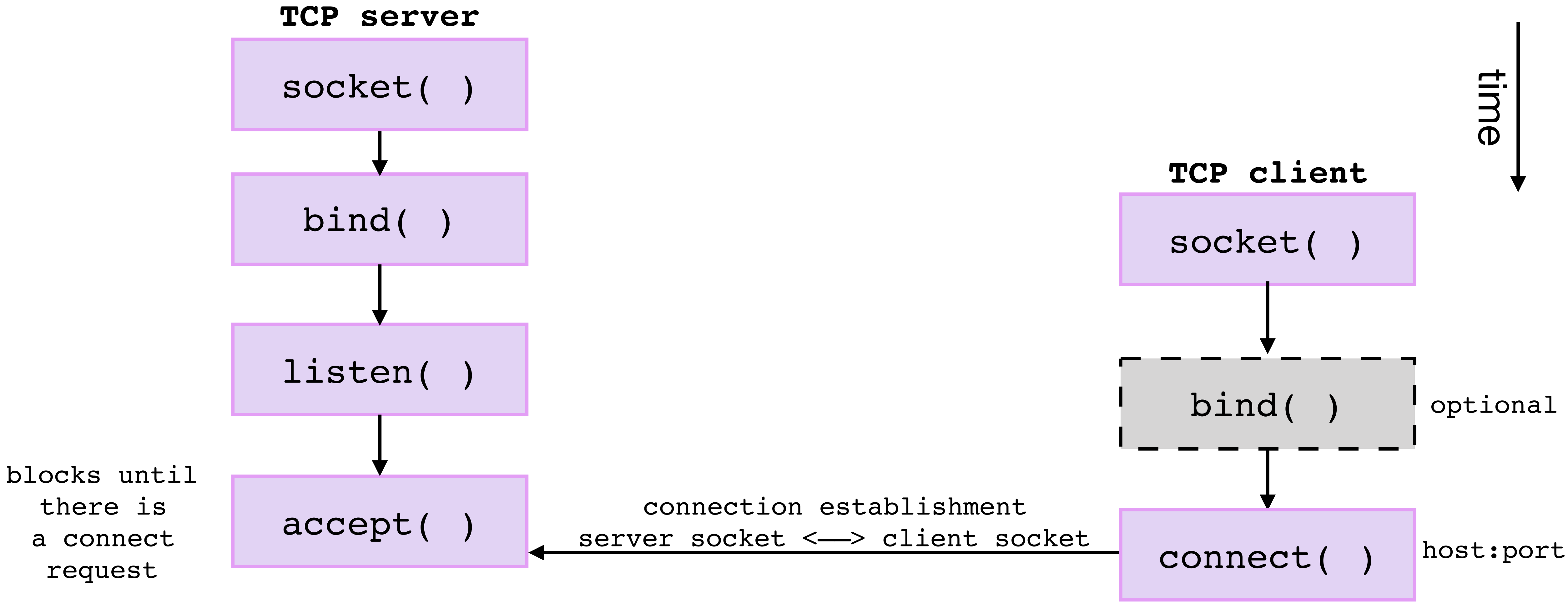
Client-Server communication over UDP



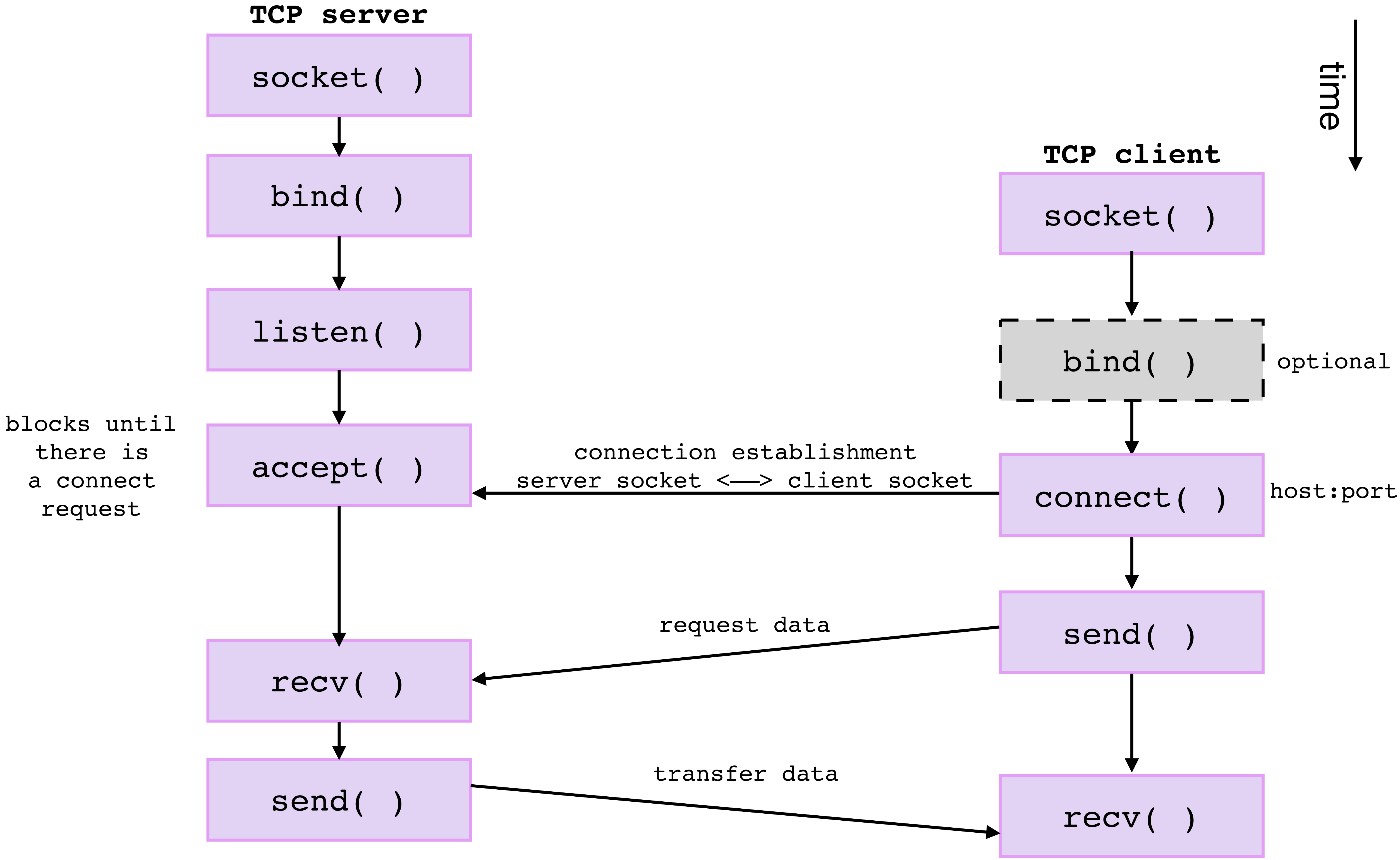
Client-Server communication over TCP



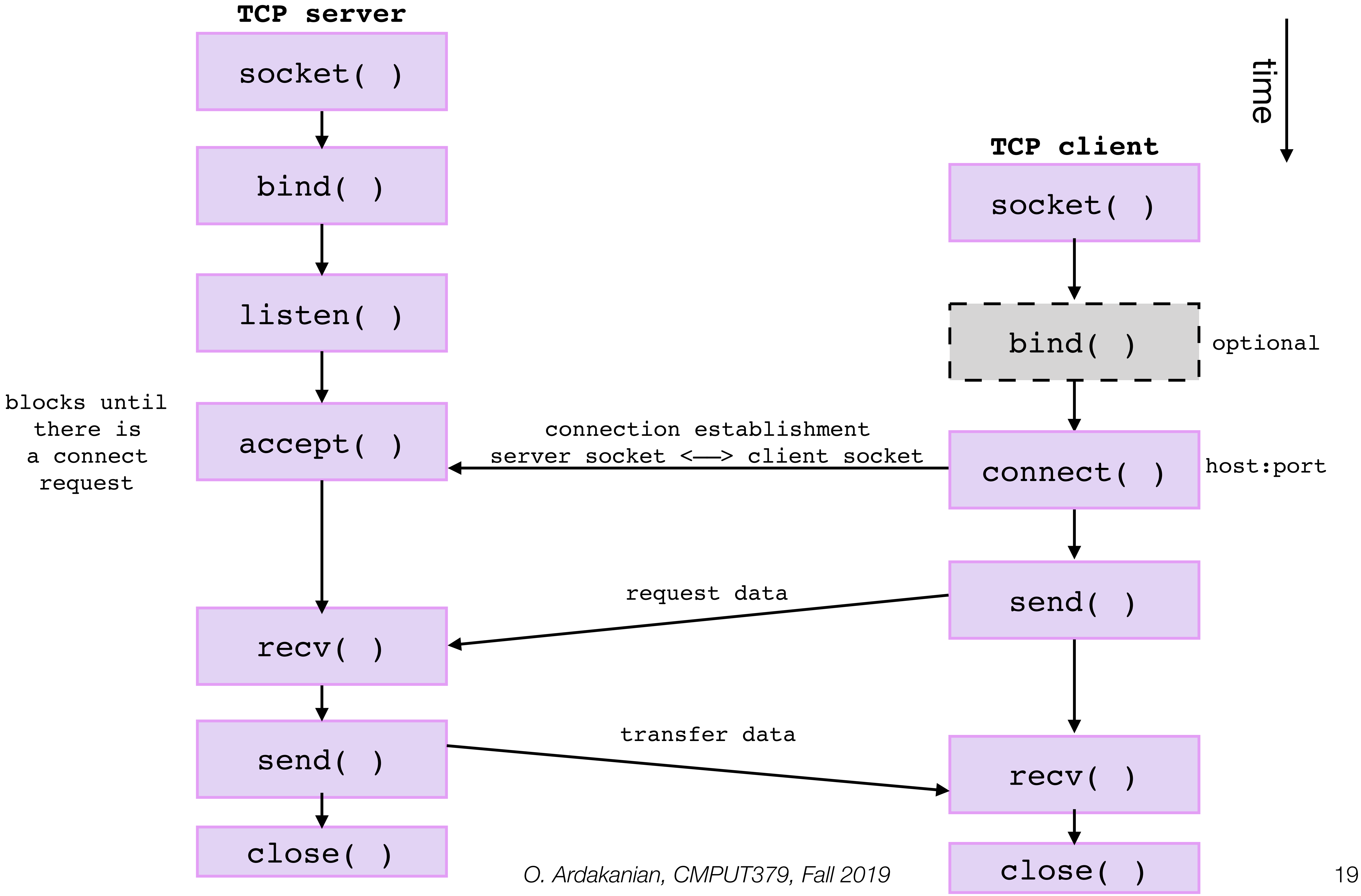
Client-Server communication over TCP



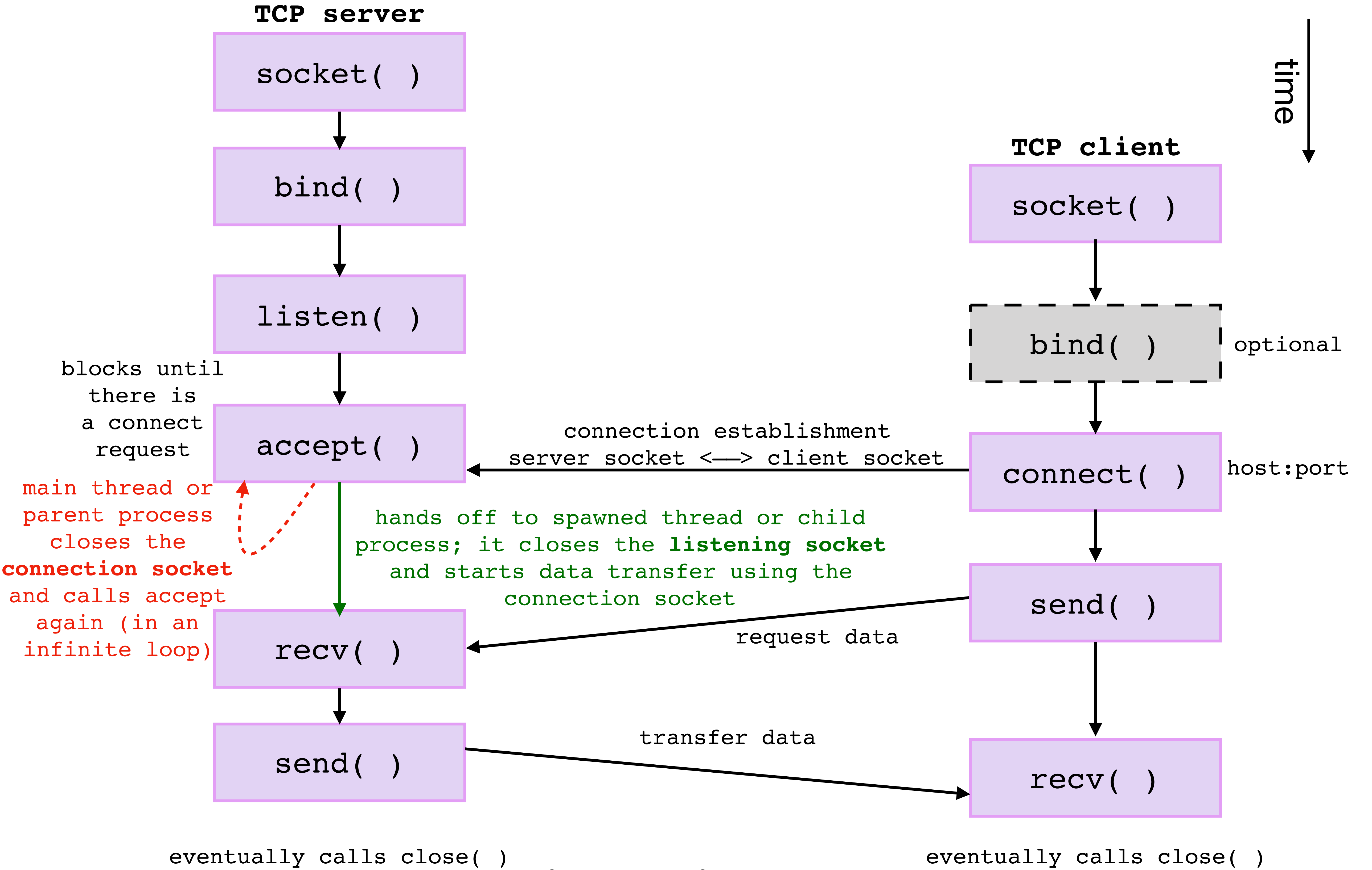
Client-Server communication over TCP



Client-Server communication over TCP



Client-Server communication over TCP: concurrent server



Remote Procedure Call (RPC)

Recap — what happens during a “local” procedure call?

- a user program calls a library function
- the library function formats arguments for the corresponding system call and issues system call exception
- the system call handler unpacks the arguments and calls the subsystem function that handles the call by performing some operation
- the system call handler puts the result in a register, and resumes the user thread
- the library function gets the system call’s result and returns to the user program

Remote Procedure Call (RPC)

- basic idea: make communication look like an ordinary function call
  - servers export procedures for some set of clients to call
  - the client does a procedure call (sends a request message) to use the server to execute a specified procedure with arguments sent across the network
  - the server processes the call and returns the response
  - OS manages the communication

Remote Procedure Call (RPC)

- basic idea: make communication look like an ordinary function call
  - servers export procedures for some set of clients to call
  - the client does a procedure call (sends a request message) to use the server to execute a specified procedure with arguments sent across the network
  - the server processes the call and returns the response
  - OS manages the communication
- application: Network File System (NFS)

Remote Procedure Call (RPC)

- basic idea: make communication look like an ordinary function call
  - servers export procedures for some set of clients to call
  - the client does a procedure call (sends a request message) to use the server to execute a specified procedure with arguments sent across the network
  - the server processes the call and returns the response
  - OS manages the communication
- application: Network File System (NFS)
- remote procedure call to look like a local procedure call by using a **stub** (just like the wrapper function) that hides the details of remote communication
  - programmer does not need to code the remote interaction
  - still much slower and less reliable than a local call
  - OS has to ensure that an RPC call is acted upon exactly once

Remote Procedure Call (RPC)

- basic idea: make communication look like an ordinary function call
  - servers export procedures for some set of clients to call
  - the client does a procedure call (sends a request message) to use the server to execute a specified procedure with arguments sent across the network
  - the server processes the call and returns the response
  - OS manages the communication
- application: Network File System (NFS)
- remote procedure call to look like a local procedure call by using a **stub** (just like the wrapper function) that hides the details of remote communication
  - programmer does not need to code the remote interaction
  - still much slower and less reliable than a local call
  - OS has to ensure that an RPC call is acted upon exactly once
- check out the map page of RPC: `rpc(3)`

Remote Procedure Call (RPC)

- the RPC mechanism uses the procedure signature (number and type of arguments and return value) for each procedure
  - to generate a client stub that bundles up the RPC arguments (**marshalling**) and sends them off to the server
  - to generate the server stub that unpacks the message (**unmarshalling**), and makes the procedure call

Remote Procedure Call (RPC)

- the RPC mechanism uses the procedure signature (number and type of arguments and return value) for each procedure
  - to generate a client stub that bundles up the RPC arguments (**marshalling**) and sends them off to the server
  - to generate the server stub that unpacks the message (**unmarshalling**), and makes the procedure call
- each message is addressed to an RPC daemon listening to a port on a remote system
  - the message contains an identifier specifying the function to execute and parameters to pass to that function

Remote Procedure Call (RPC)

- the RPC mechanism uses the procedure signature (number and type of arguments and return value) for each procedure
  - to generate a client stub that bundles up the RPC arguments (**marshalling**) and sends them off to the server
  - to generate the server stub that unpacks the message (**unmarshalling**), and makes the procedure call
- each message is addressed to an RPC daemon listening to a port on a remote system
  - the message contains an identifier specifying the function to execute and parameters to pass to that function
- marshalling may require converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.

Remote Procedure Call (RPC)

```
procedure P1
begin
    ...
    call Function(args)
    ...
end P1
```

```
procedure
realFunction(args)
begin
    ...
    return(results)
end realFunction
```

call

return

call

return

```
procedure Function(args)
begin
  <marshall parameters>
  send(FunctionServer,params)
  receive(FunctionServer,results)
  <unpack results>
  return(results)
end Function
```

Client stub

receive

send

```
process FunctionServer
begin
  loop
    sender := select()
    receive(sender,params)
    <unpack parameters>
    call realFunction(args)
    <marshall results>
    send(sender,results)
  end loop
end FunctionServer
```

Server stub

Remote Procedure Call (RPC)

How does the client know the right port?

- the binding can be static: fixed at compile time
- the binding can be dynamic: fixed at runtime

Remote Procedure Call (RPC)

How does the client know the right port?

- the binding can be static: fixed at compile time
- the binding can be dynamic: fixed at runtime
- in most RPC systems, dynamic binding is performed using a name service
  - when the server starts up, it exports its interface and identifies itself to a network name server
  - the client, before issuing any calls, asks the name service for the address of a server whose name it knows and then establishes a connection with the server

Homework

- Implement a concurrent TCP server