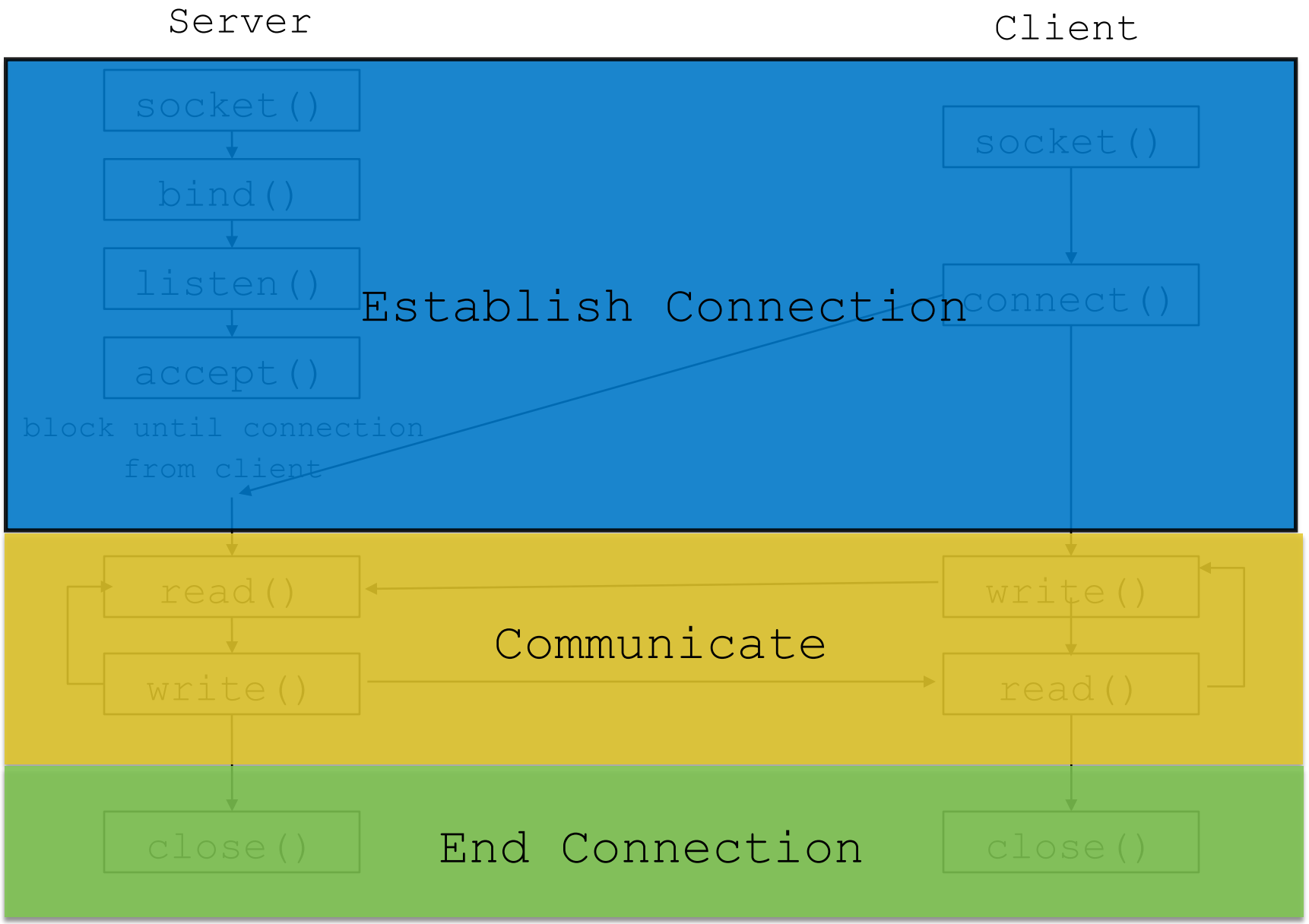
CSCB09: Software Tools and Systems Programming

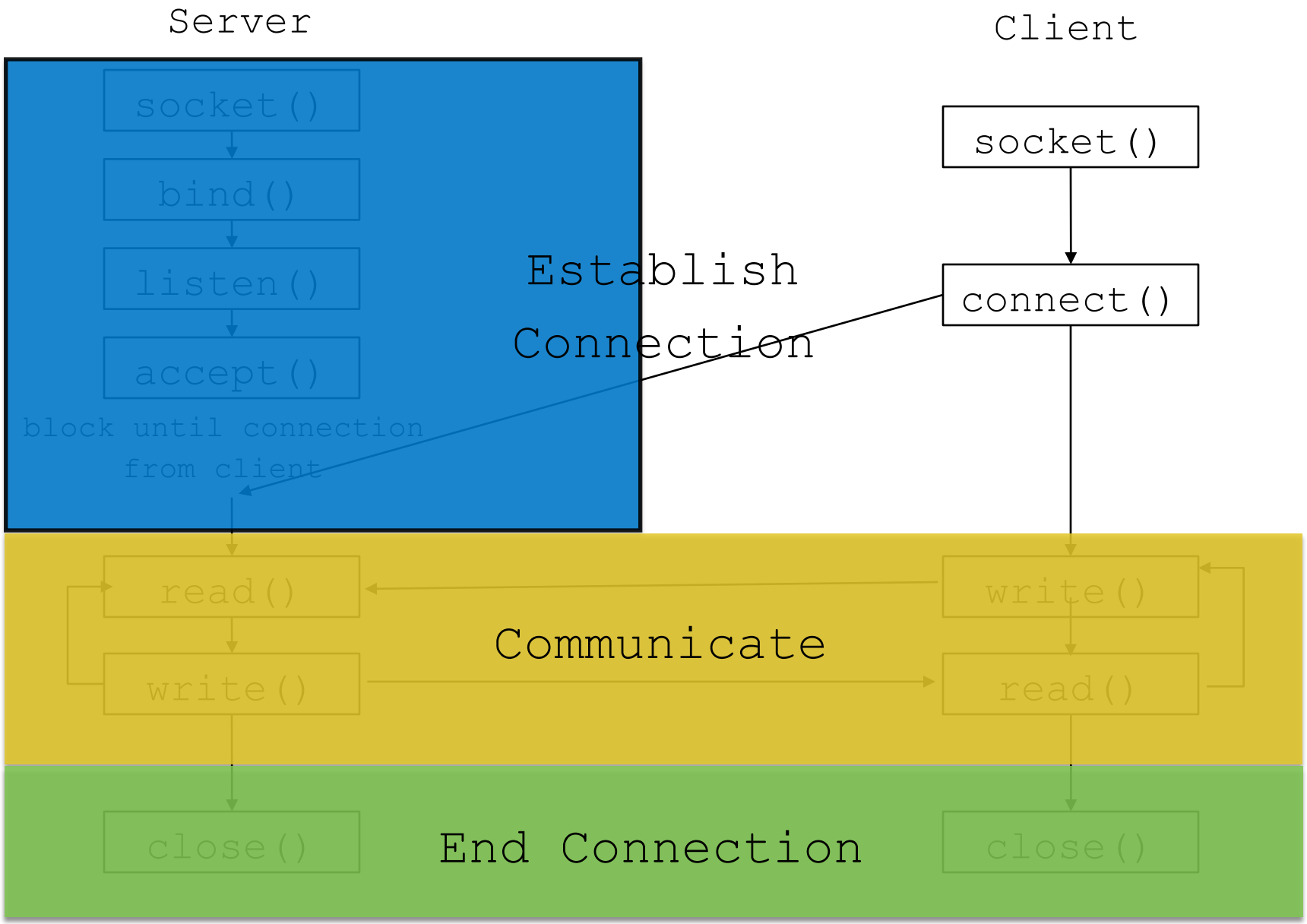
Bianca Schroeder
bianca@cs.toronto.edu
IC 460

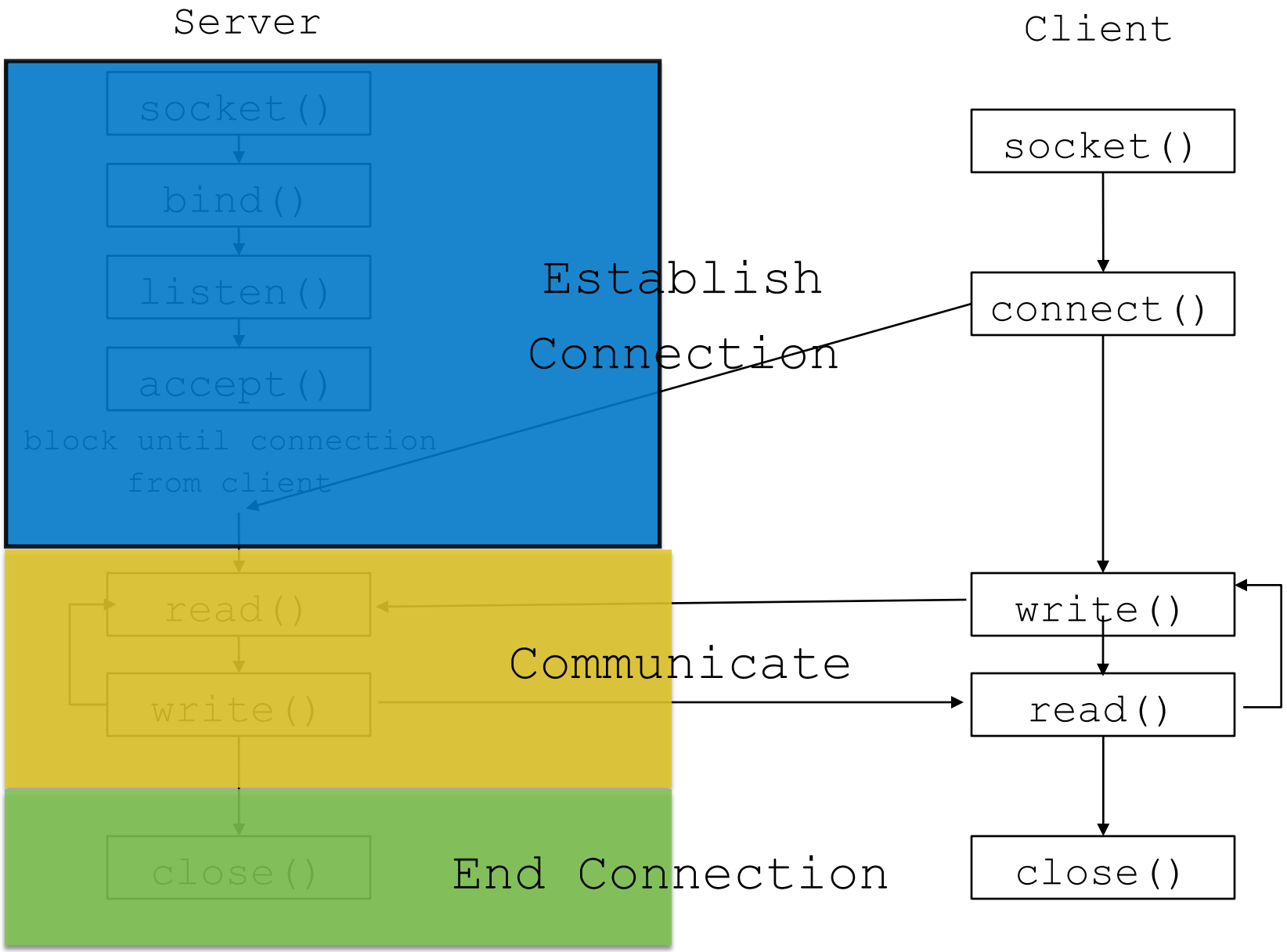
COURSE EVALUATIONS

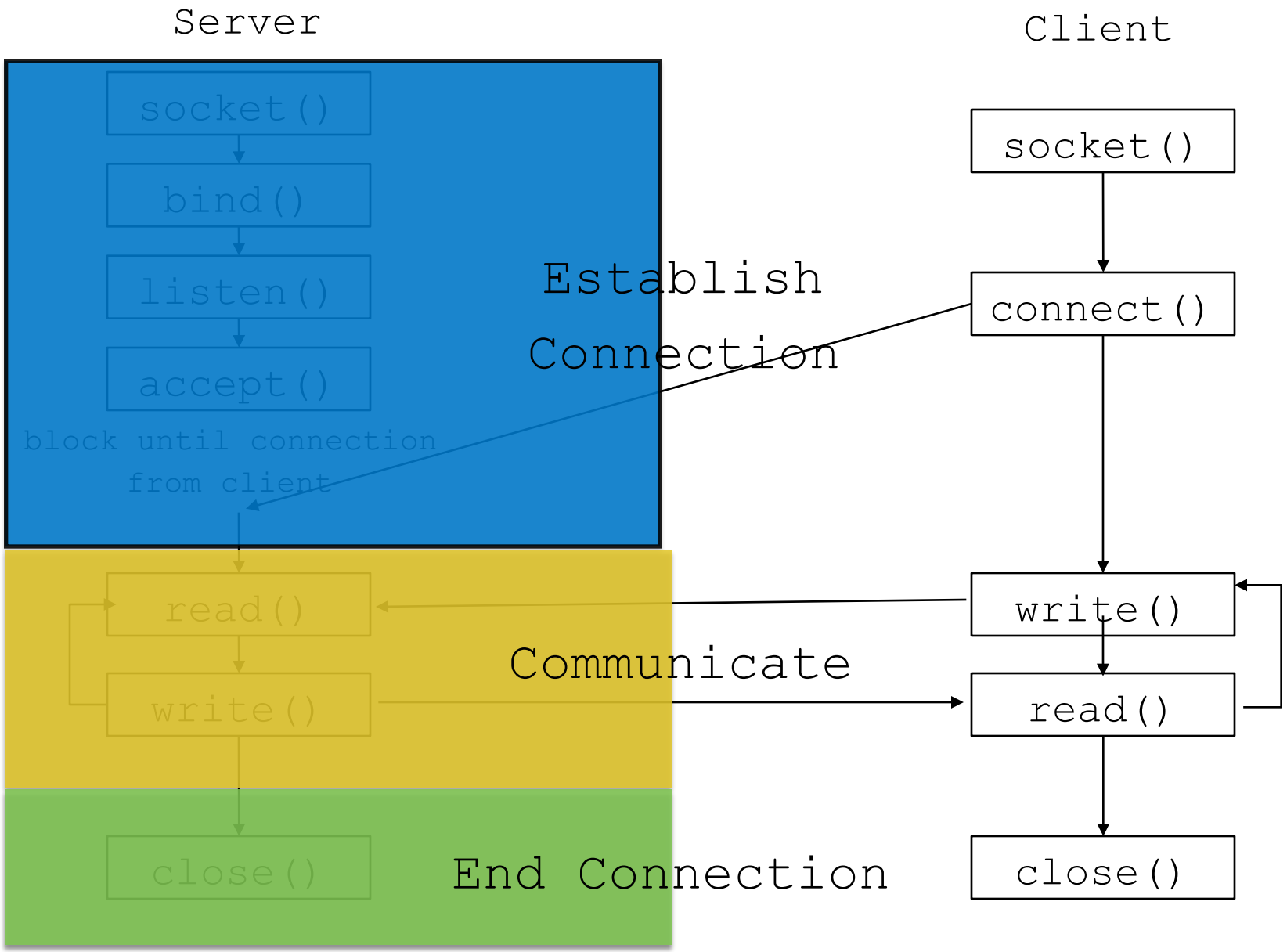
We care what you think,

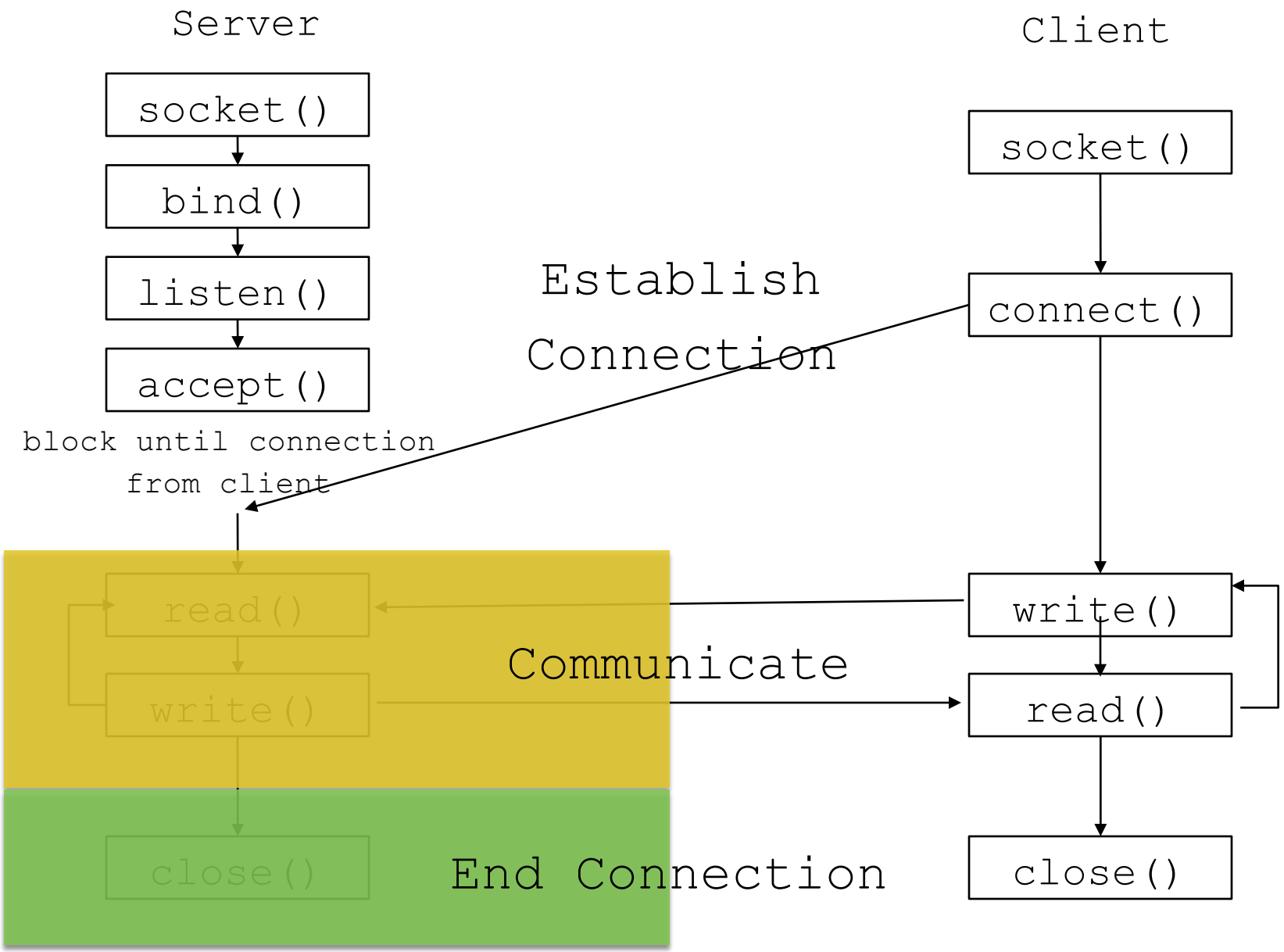
TELL US!

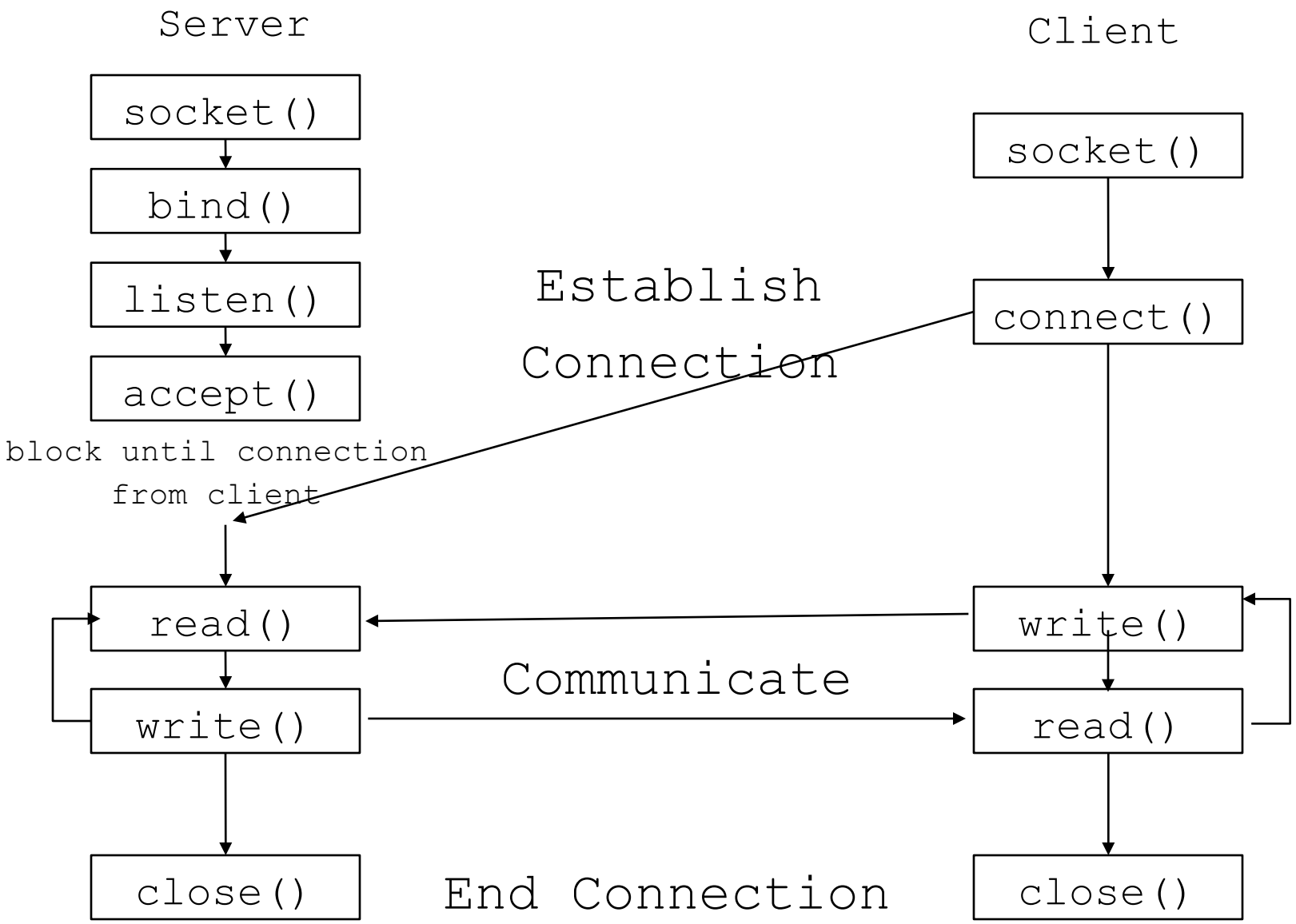

Server
Client
socket()
bind()
listen()
accept()
block until connection from client
socket()
connect()
Establish Connection
read()
write()
write()
read()
Communicate
close()
close()
End Connection


Server
Client
socket()
bind()
listen()
accept()
block until connection from client
socket()
connect()
Establish Connection
read()
write()
write()
read()
Communicate
close()
close()
End Connection


Server
Client
socket()
bind()
listen()
accept()
block until connection from client
Establish Connection
connect()
read()
write()
Communicate
write()
read()
close()
End Connection
close()


Server
Client
socket()
bind()
listen()
accept()
block until connection from client
Establish Connection
connect()
read()
write()
Communicate
write()
read()
close()
End Connection
close()


Server
socket()
bind()
listen()
accept()
block until connection
from client
Client
socket()
connect()
Establish
Connection
read()
write()
Communicate
write()
read()
close()
End Connection
close()


Server
Client
socket()
bind()
listen()
accept()
block until connection
from client
socket()
connect()
Establish
Connection
read()
write()
write()
read()
Communicate
close()
End Connection
close()

Server side: socket() works same as for client

```
int socket(int family, int type,
           int protocol);
```

- family specifies protocol family:
  - `AF_INET` – IPv4
  - `AF_LOCAL` – Unix domain
- type
  - `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW`
- protocol
  - set to 0 except for RAW sockets
- returns a socket descriptor

bind() socket to a name

```
int bind(int sockfd,
      const struct sockaddr *servaddr,
       socklen_t addrlen);
```

- Servaddr sets IP address & port at which to accept requests
- Returns negative value on error, zero on success

- ```
  struct sockaddr_in {
    short          sin_family; /*AF_INET */
    u_short        sin_port;
    struct in_addr sin_addr;
    char           sin_zero[8]; /*filling*/
  };
  ```
- `sin_addr` can be set to `INADDR_ANY` to communicate on any network interface.
  ```

Set up queue in kernel

```
int listen(int sockfd, int backlog)
```

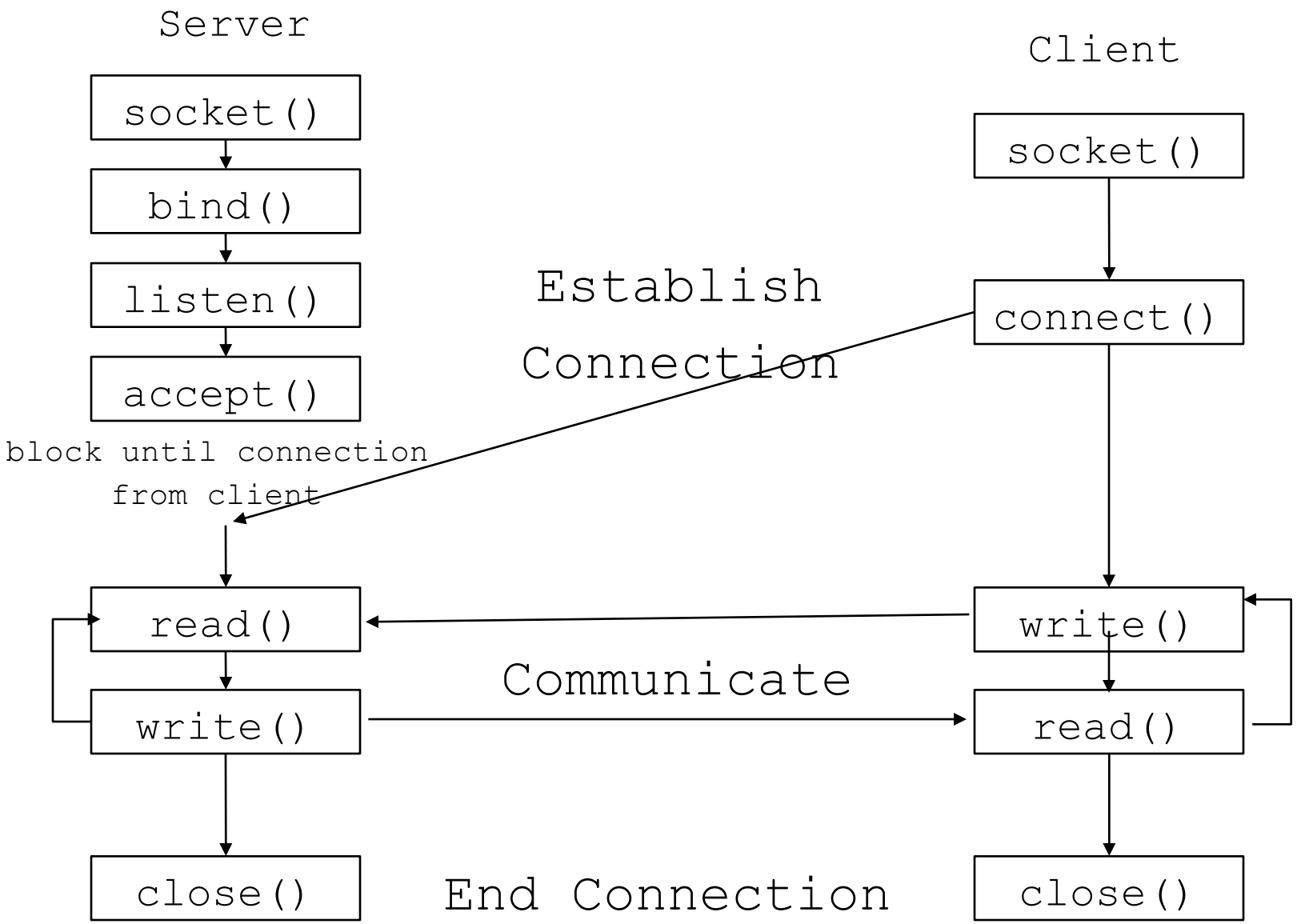
- after calling `listen`, a socket is ready to accept connections
- prepares a queue in the kernel where partially completed connections wait to be accepted.
- `backlog` is the maximum number of partially completed connections that the kernel should queue.

Complete the connection

```
int accept(int sockfd,
           struct sockaddr *cliaddr,
           socklen_t *addrlen);
```

- blocks waiting for a connection (from the queue)
- returns a new descriptor which refers to the TCP connection with the client
  - This is the descriptor to be used for reads and writes from/to the client.
- `sockfd` is the listening socket
- `cliaddr` is used to return the address of the client

Done!


Server
socket()
bind()
listen()
accept()
block until connection
from client
Client
socket()
Establish
Connection
connect()
read()
write()
Communicate
write()
read()
close()
End Connection
close()

Next: Writing a server that can handle multiple clients

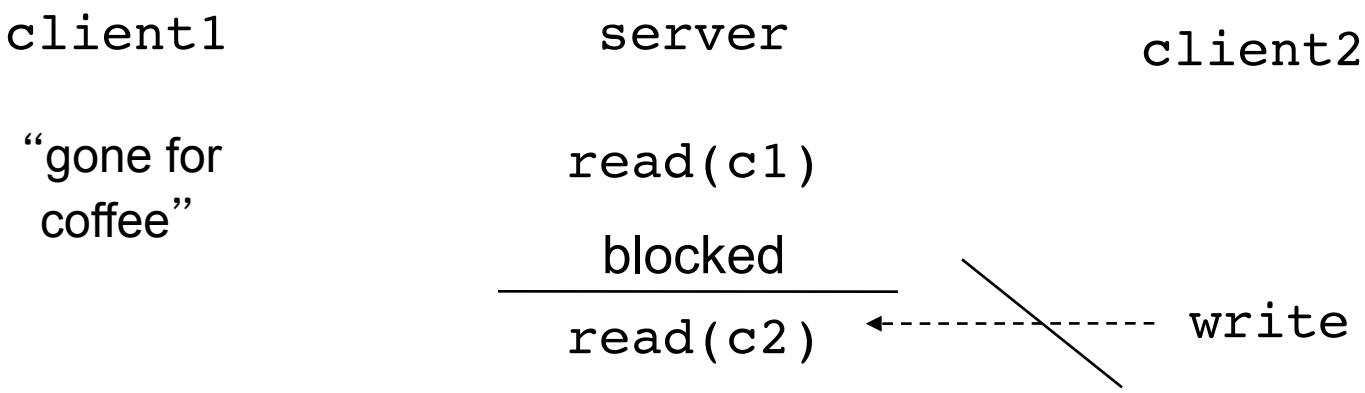
```
client1                server                client2

                   while(1){
                      read(c1);
                      read(c2);
                   }
```

The above is a draft for a server that reads from two clients.

Problems?

The problem



When reading from multiple sources, blocking on one of the sources could be bad.

One solution: one process for every client. What are the pros and cons of this solution?

The solution: Multiplexing

```
int select(int maxfdp1,
           fd_set *readset,
           fd_set *writeset,
           fd_set *exceptset,
           const struct timeval *timeout);
```

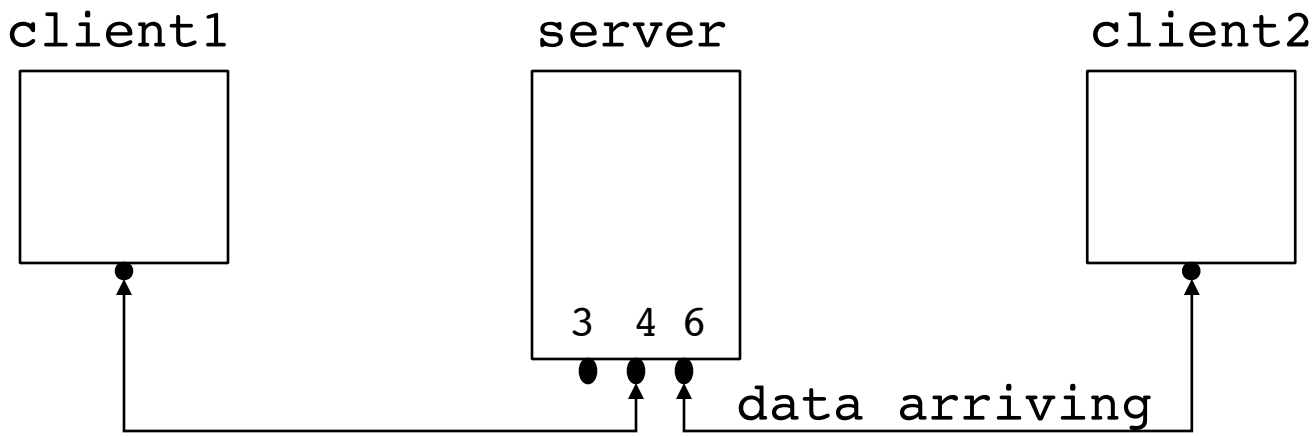
A call to select returns when one of the file descriptors in one of the sets is ready for I/O.

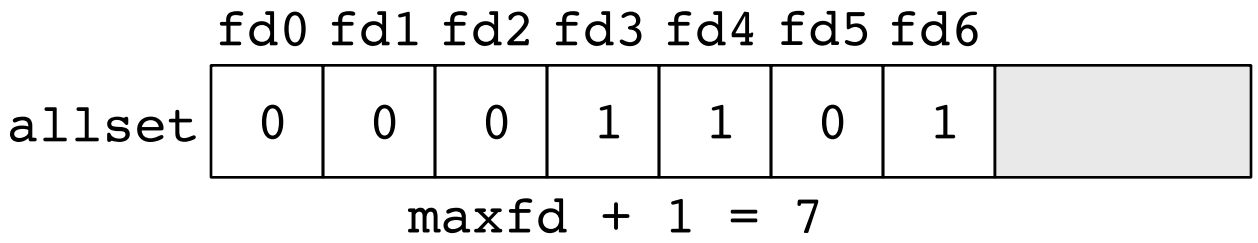
Upon return, the file descriptor set contains the file descriptors that are ready for I/O.

If timeout is not NULL, then select returns when a descriptor is ready or timeout time has passed.

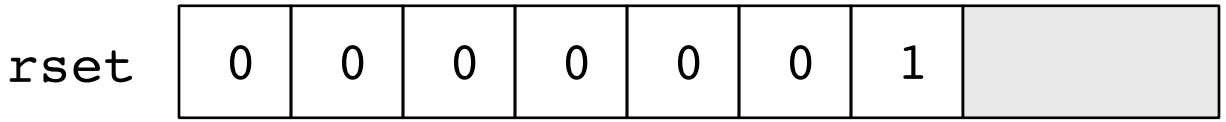
If timeout is 0, select returns immediately after checking descriptors.

Descriptor sets





After select (assuming client2 has written some data):



Readiness

Ready to read when

- there is data in the receive buffer to be read
- end-of-file state on file descriptor
- the socket is a listening socket and there is a connection pending
- a socket error is pending

Ready to write when

- there is space available in the write buffer
- a socket error is pending

Exception condition pending when

- TCP out-of-band data

We are typically interested in when bytes are available to be read, but sometimes we use select on write or exception sets

Descriptor sets

- Implementation is hidden in the `fd_set` data type
- `FD_SETSIZE` is the number of descriptors in the data type
- `maxfdp1` specifies the number of descriptors to test

Macros:

```
void FD_ZERO(fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
int  FD_ISSET(int fd, fd_set *fdset);
```

Let's put this into practice on the worksheet

```
FD_ZERO(fd_set *set)
FD_SET(int fd, fd_set *set)
FD_ISSET(int fd, fd_set *set)

select(int max_fd,
       fd_set *read_fds,
       fd_set *write_fds,
       fd_set *except_fds,
       struct timeval *timeout)
```

STDIN_FILENO
STDOUT_FILENO

} File descriptors for stdin and stdout