The Sockets API

[Wait! If you are not familiar with file descriptors and the UNIX read and write system calls, read chapter 10 of Bryant and O'Hallaron and/or my summary before going on.]

In this section we take a closer look at how to use TCP from C using the UNIX interface.

In UNIX and POSIX, TCP is accessed via the *Sockets interface*.

Sockets can be used with a number of networking protocols, not just TCP.

The main data structures include.

- **`struct`** `sockaddr` — includes
  - an IP address and
  - a port number
- **`struct`** `sockaddrinfo` — includes
  - a `sockaddr`
  - a family (e.g. IPv4 or IPv6)
  - a socket type (e.g. streaming or datagram)
  - a protocol (e.g. TCP or UDP)
- file descriptors
  - As with disk files, file descriptors are integers that are obtained from the operating system and that can be used with the `read` and `write` functions.

Making things confusing `sockaddr` structures look quite different depending on whether IPv4 or IPv6 is

being used. The calls use pointers of type **`struct`** `sockaddr`, but the actual objects should be stored in a **`struct`** `sockaddr_storage` structure, to ensure there is enough space.

Some useful functions in the API include

- `getaddrinfo`
  - Does DNS lookup
  - Input:
    - A host name. E.g. `"server.black.com"`
    - A port number.
    - A set of restrictions. E.g. must be TCP.
  - Output:
    - A list of `sockaddrinfo` records
  - It returns a list because a single host name may map to several IP addresses and the host may support both IPv4 and IPv6 and TCP and UDP.
- `socket`
  - Creates, but does not open, a “socket”
  - Input:
    - a family (e.g. IPv4 or IPv6)
    - a socket type (e.g. streaming or datagram)
    - a protocol (e.g. TCP or UDP)
  - Output:
    - a file descriptor for the socket
- `connect`
  - Used by the client to create a connection.

  - Analogous to `open` for the client
  - Input:
    - a file descriptor for a socket
    - the socket address (struct sockaddr) for the service

- `bind`
  - Used by the server to connect a server socket to a port number.
  - Input:
    - a file descriptor for a socket
    - the socket address (struct sockaddr) for the service

- `listen`
  - Used by the server to initiate listening on a port
  - Input:
    - a file descriptor for a bound socket
    - the number of connection requests that may be queued

- `accept`
  - Used by the server to create a new connection
  - Analogous to `open` for the server
  - Waits until a connection is requested by a client and the connection is established
  - Input:
    - a file descriptor for a server socket that is listening
  - Output:

    - A file descriptor for a new socket.
    - The socket address of the new client.

- `read`
  - Waits until there is at least one byte can be read or the connection is closed.
  - Input
    - A file descriptor for a socket.
  - Output
    - A sequence of 0 or more bytes.
  - 0 bytes means the connection was closed.
- `write`
  - Input:
    - A file descriptor for a socket.
    - A sequence of bytes to send.
- `close`
  - Input:
    - A file descriptor
  - Half closes the connection
- `select`
  - Input
    - A set of file descriptors
  - Output
    - A set of file descriptors that can be read from
  - select blocks until at least on file descriptor in the set can be read
  - The output is a subset of the input.

* Calling read on any file descriptor in the output set will not block.

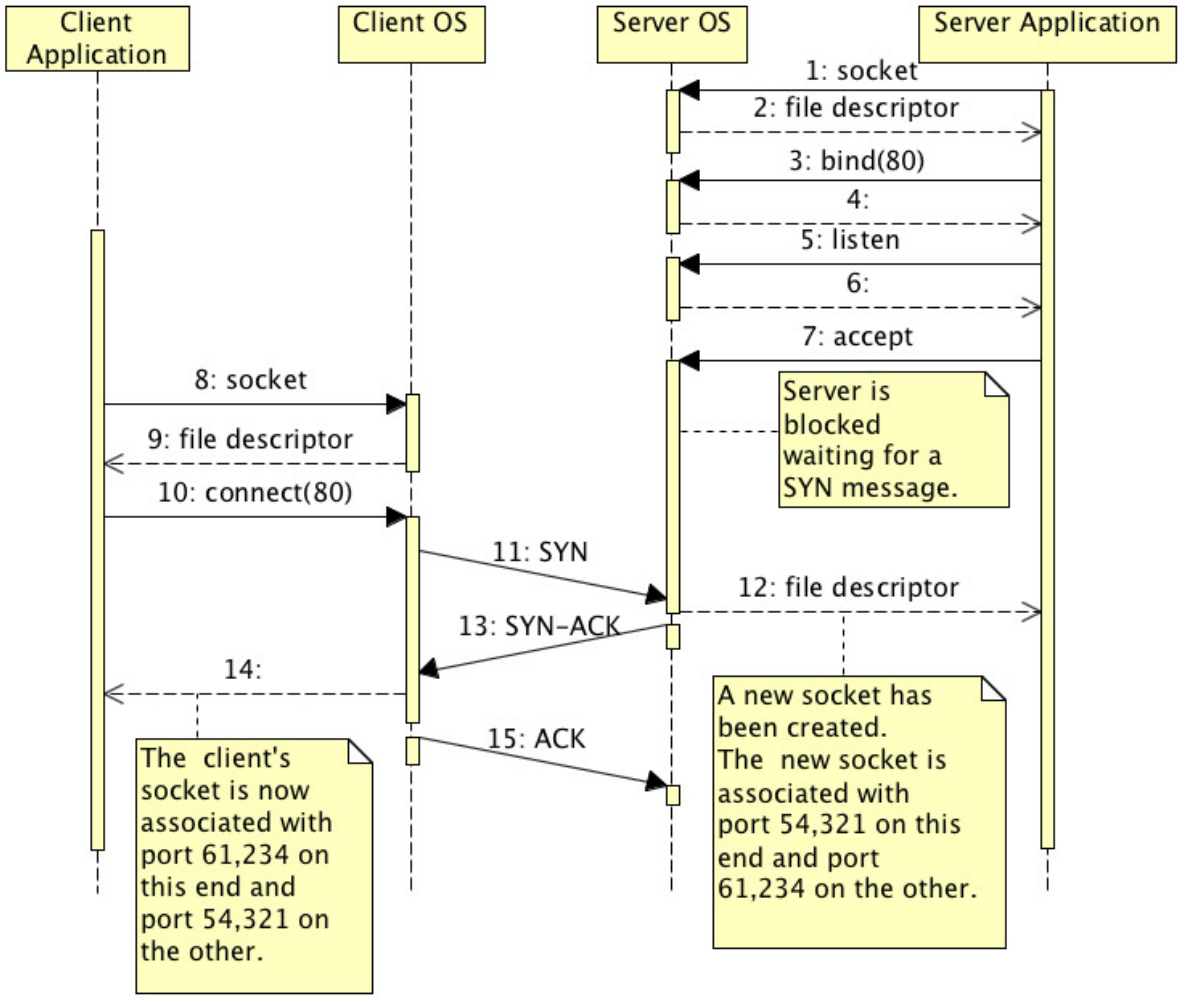
Some examples

The following images illustrate use of the sockets interface in UNIX

Legend

- Solid horizontal arrows represent calls from the application the OS.
- Dashed horizontal arrows represent returns from those calls.
- Diagonal solid arrows represent TCP messages.
- The thin yellow rectangles show activities. For example the execution of a subroutine.

Establishing a connection

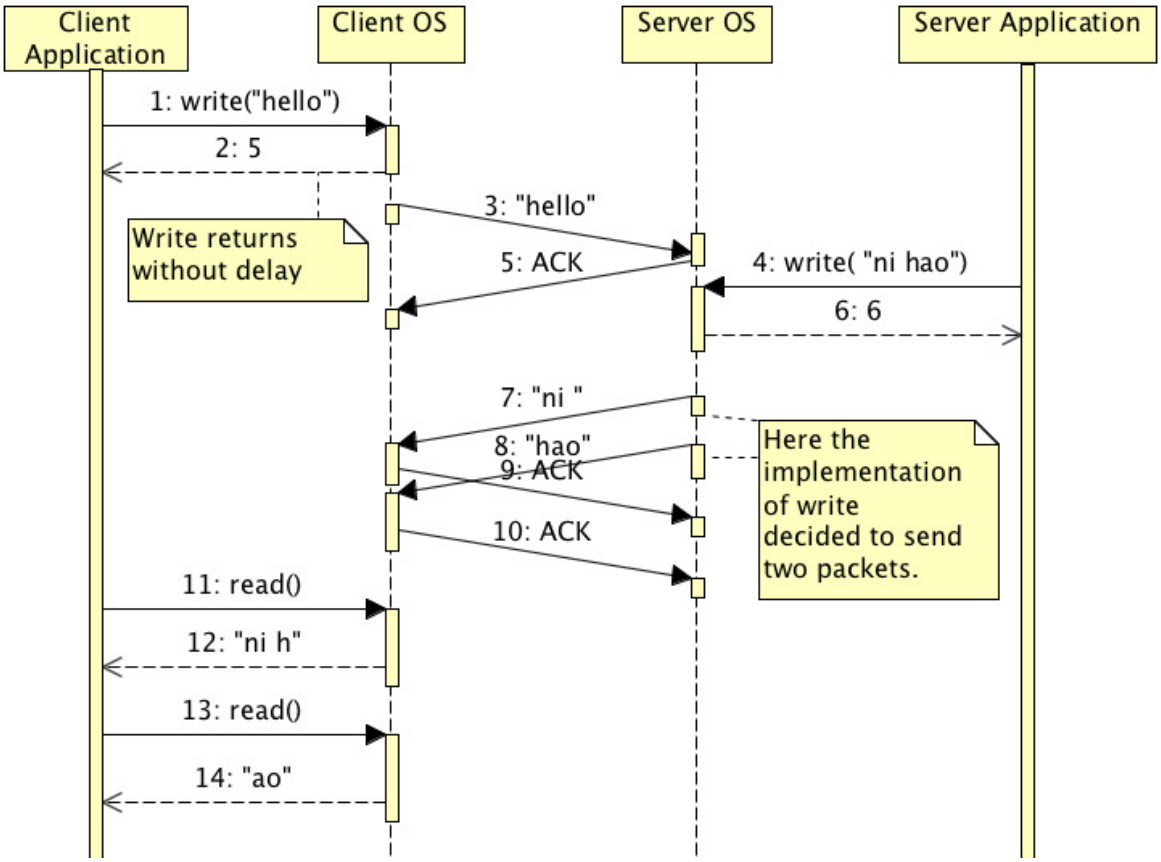


Buffered communication.

It is important to understand that each socket endpoint has two buffers one for sending and one for receiving.

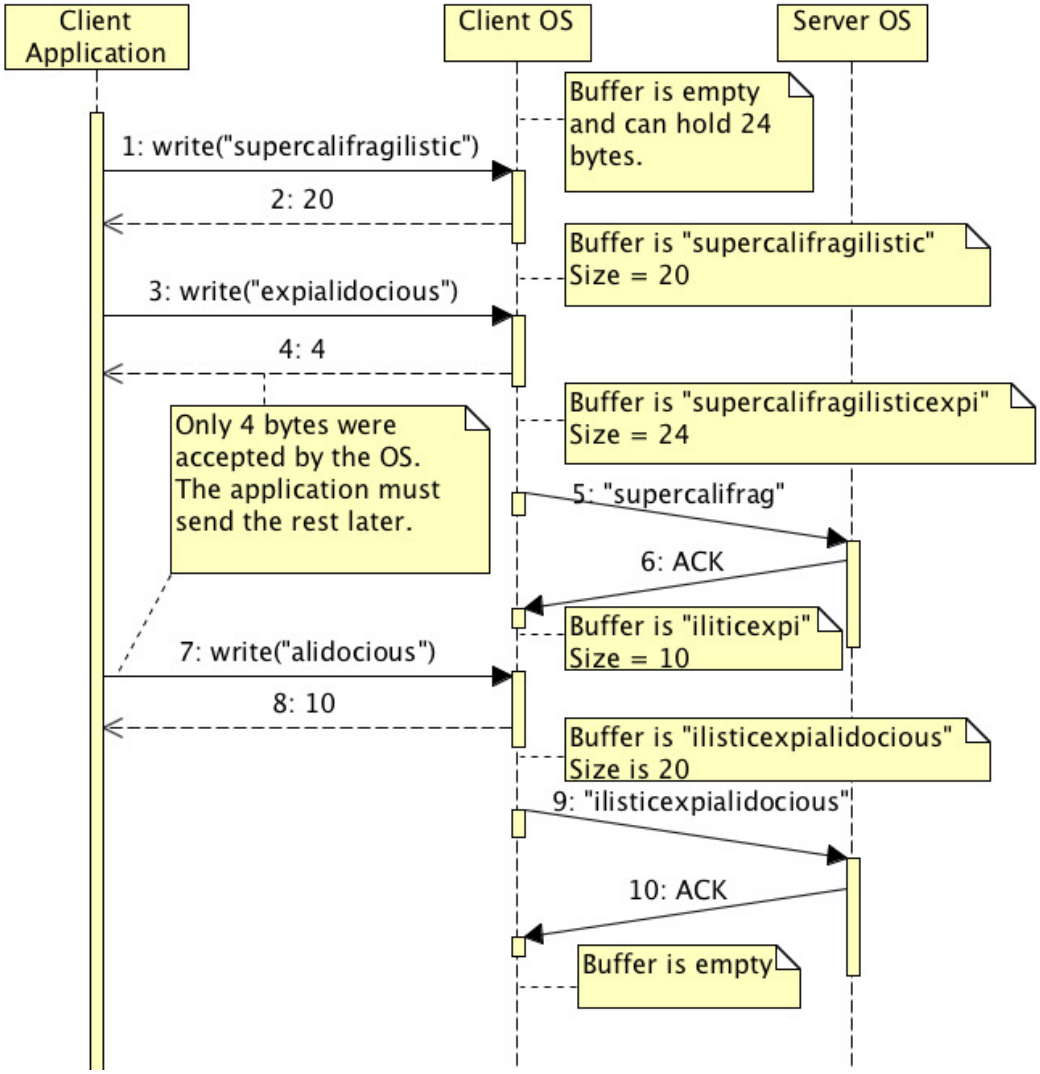
- `read` calls transfer data from the read buffer to the application.
- `write` calls transfer data from the application to the write buffer.
- The OSs use TCP to transfer from the write buffer on one host to the read buffer on the other.

The follow image shows reading and writing



Note that the amount read by each read may not relate to the amount written. In this case, the server sends ‘ni hao’ with one write, but it takes 2 reads to read everything written.

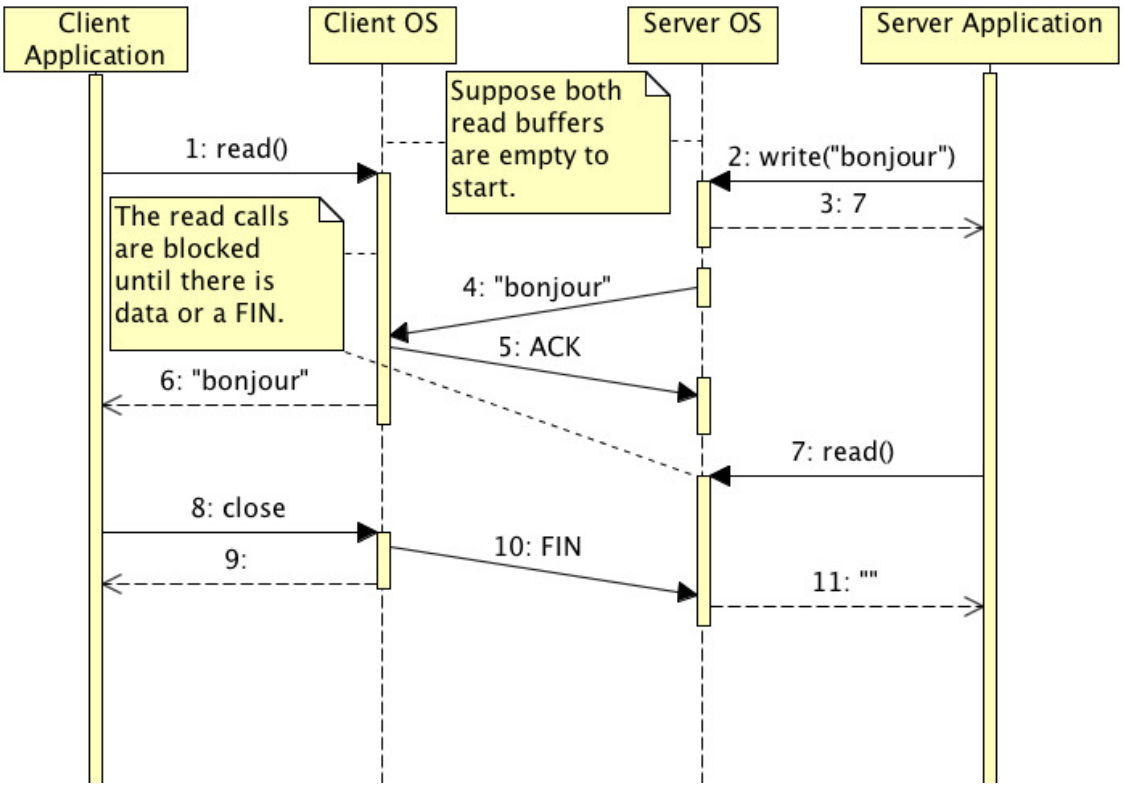
Nonblocking writes: Calls to `write` might not write the entire message. We may have to call `write` many times to completely transfer our message to the write buffer.



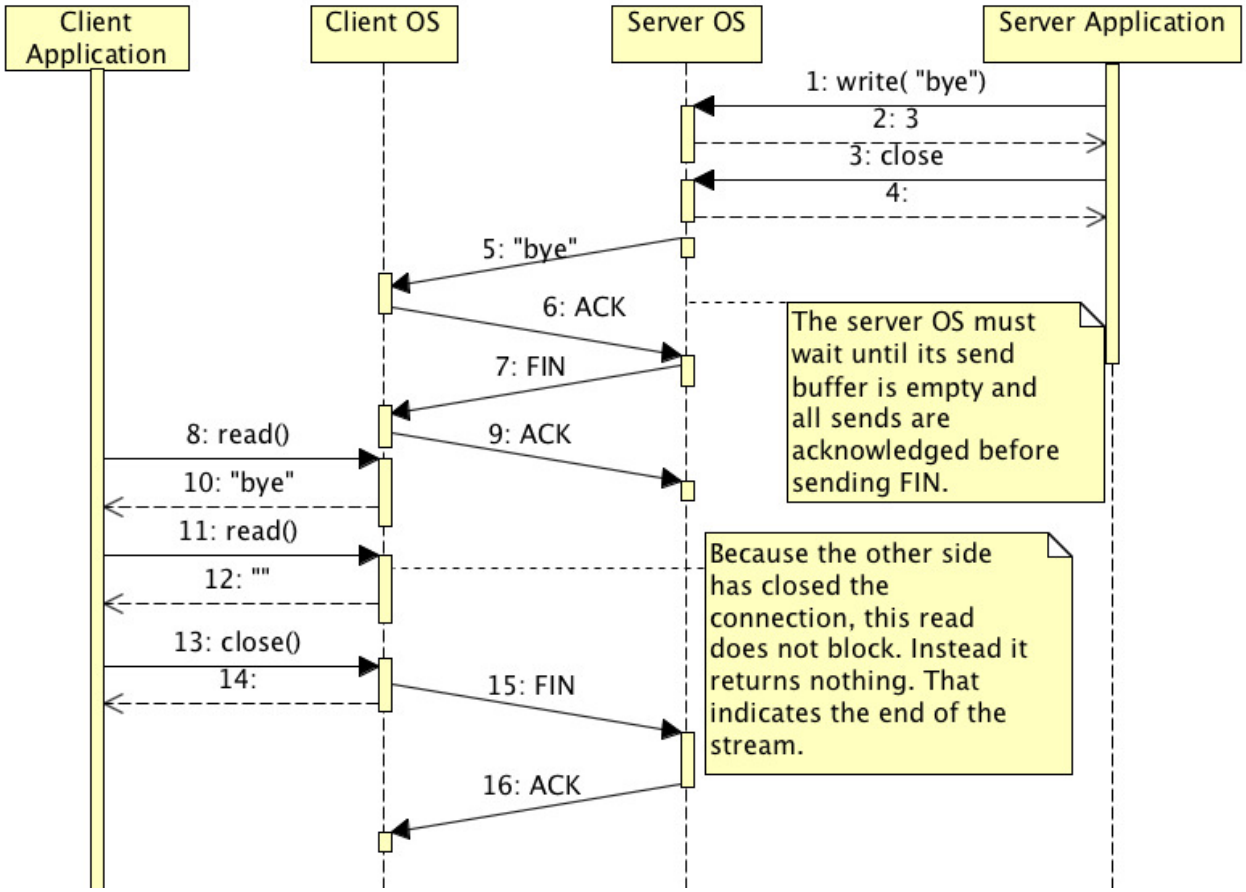
Blocking reads. `read` will block (i.e. wait) until one of the following happens

- There is a positive number of bytes it can read.
- The socket is closed.
- An error happens.

The diagram below shows the first two cases



Closing a connection



Each process closes its endpoint.

End of networks-02.