Socket API

The Socket API was created as part of Berkeley UNIX BSD4.2 in 1983. Its variants, such as WinSOCK, exist for all serious systems.

From the start, it was a library of `C` functions that allowed application–layer programs to communicate with transport–layer protocols TCP and UDP. The API uses IP addresses and port numbers, so it is Internet–oriented.

Hence, the Socket API is a pair of communication protocols: application software to TCP/IP or UDP/IP (with responses coming back).

There is plenty of literature on the Socket API; my favourites are Beej's Guide and Spencer's Socket site. Also: Java Socket Class, wikipedia and more.

Overview of Socket API

The Socket API uses virtual links called **sockets**. A socket is

- an imaginary point–point link between TLs on two machines (Transmission Control Protocol—TCP). This link exists only for the duration of a session.
- an imaginary omni-directional receiver combined with a point–to–point sender (User Datagram Protocol—UDP). This "device" exists either permanently or is activated and deactivated when needed.

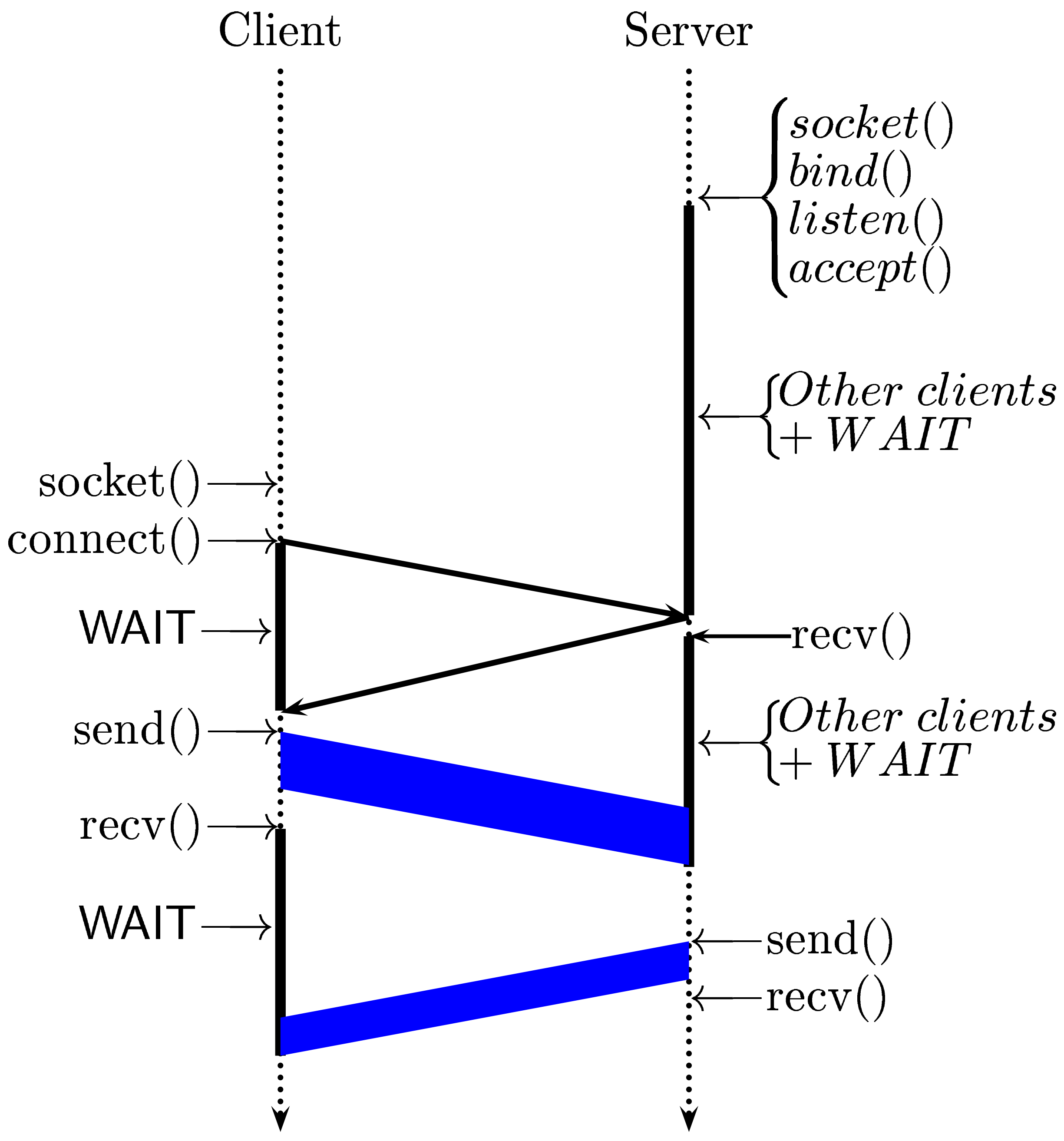
TCP Client/Server

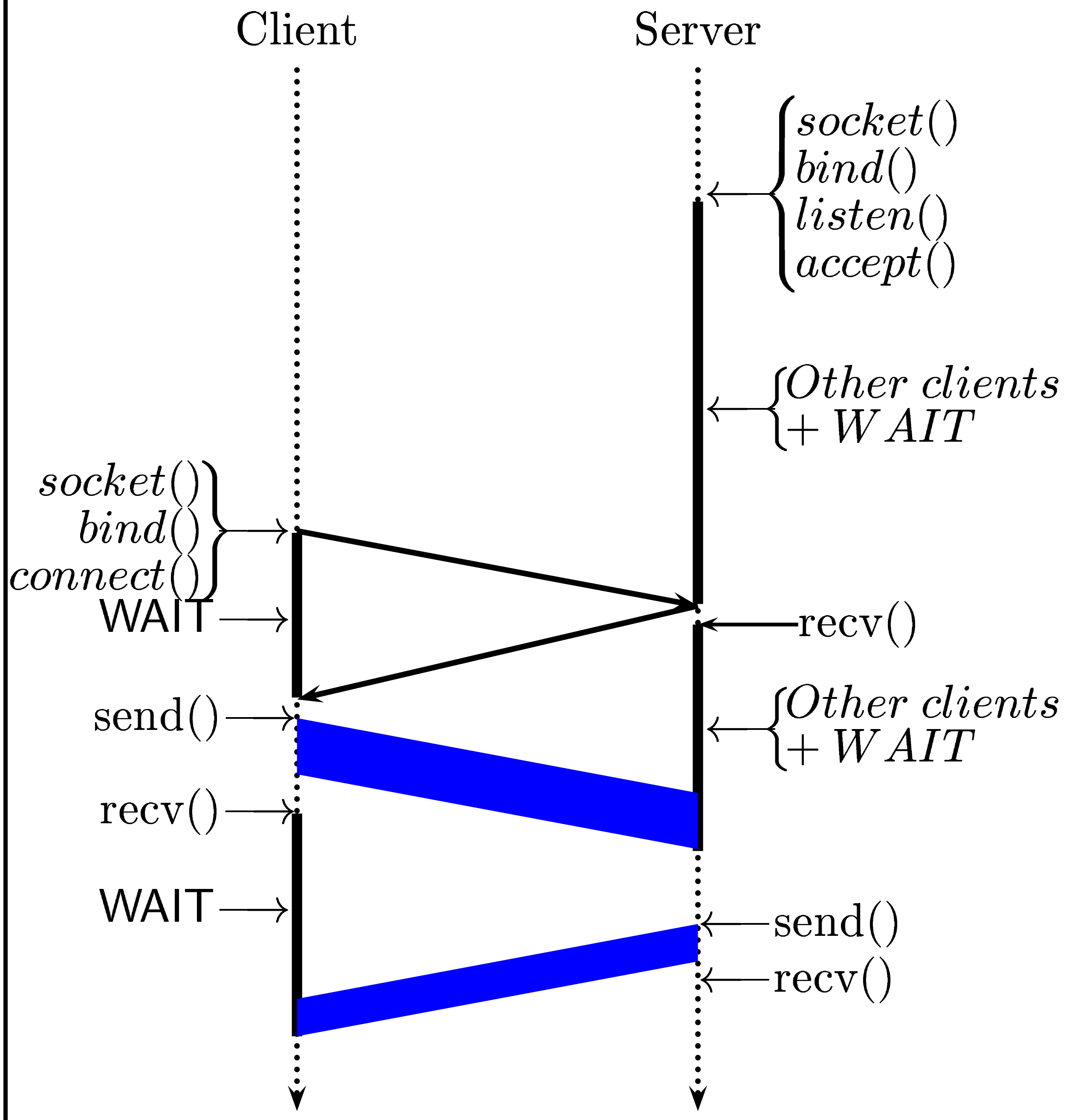
Typically, the API is a client–server interface. Its library covers the functionality of both of them.

TCP

Client	Server
socket()	socket()
bind() (?)	bind()
	listen()
connect()	accept()
$[\text{send}() + \text{recv}()]^*$	$[\text{send}() + \text{recv}()]^*$
close()	close()

The server closes not the socket created by socket(), but the one given by accept().


Client
Server
socket()
bind()
listen()
accept()
Other clients
+ WAIT
socket()
connect()
WAIT
recv()
send()
Other clients
+ WAIT
recv()
WAIT
send()
recv()


Client
Server
socket()
bind()
listen()
accept()
Other clients
+ WAIT
socket()
bind()
connect()
WAIT
recv()
Other clients
+ WAIT
send()
recv()
WAIT
send()
recv()

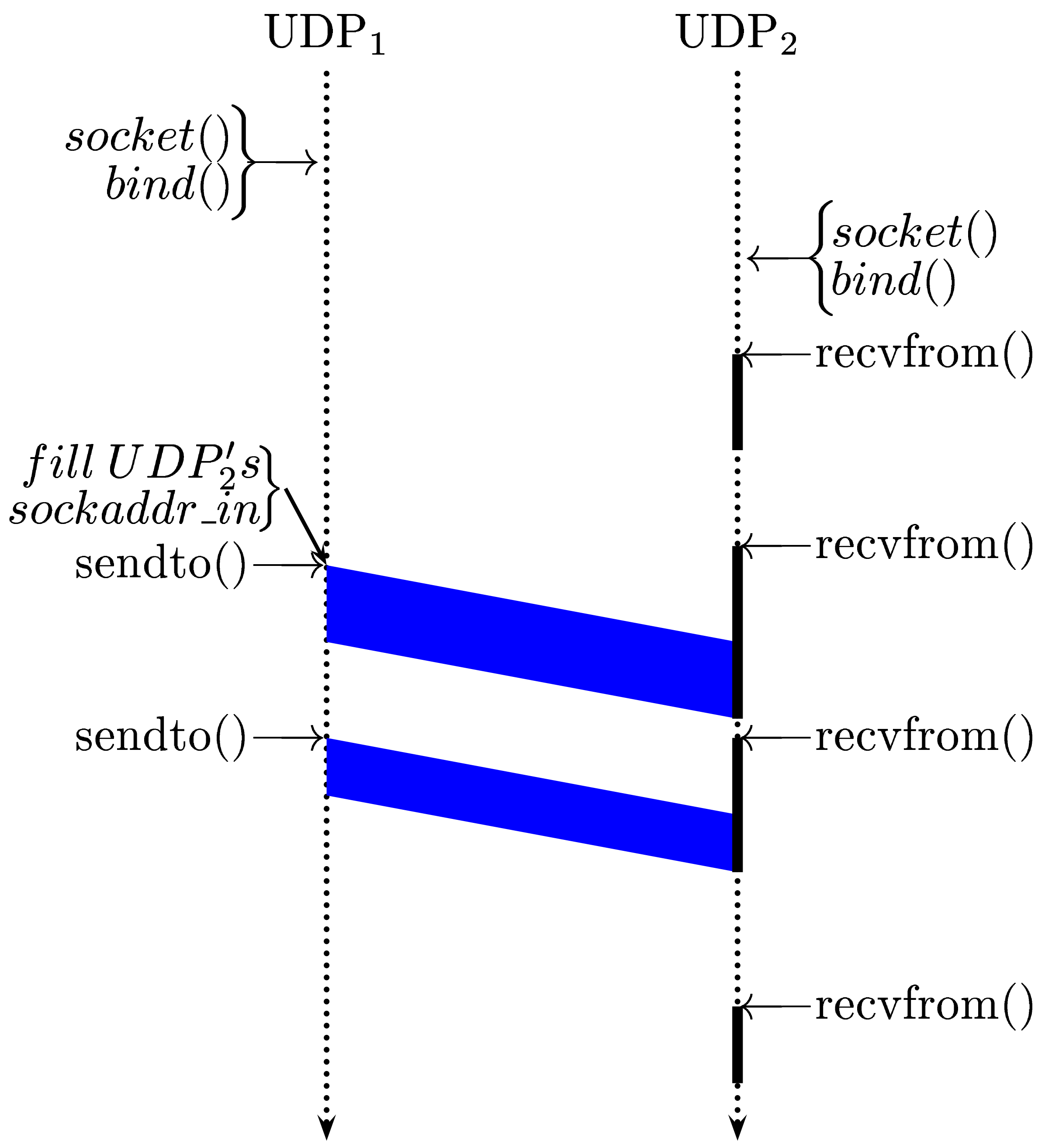
UDP Server/Server

In UDP there is no real distinction between servers and clients, other than the vague persistent/dormant behaviour (a client will become dormant if it issues a close() request).

UDP

Client	Server
socket()	socket()
bind()	bind()
$[\text{sendto() + recvfrom()}]^*$	$[\text{sendto() + recvfrom()}]^*$
close()	

There is no separate data socket, so the server has nothing to close.



recvfrom() accepts datagrams from anyone and returns the address of the sender.

Socket API

desc = socket(protocolfamily , type , protocol) ;

protocolfamily: PF_INET (for Internet) etc.

type: SOCK_STREAM, SOCK_DGRAM etc.

protocol 0 (normally) or a pointer to a struct manufactured by getprotobyname("tcp") or similar (see /etc/protocols).

Assigning a port to a socket

```
returncode = bind( desc , localaddress , addresslength ) ;
```

The second argument is of type `(struct sockaddr *`:

```
struct sockaddr {
    short sa_family /* protocol family */
    char sa_data[14] ; /* address */
} ;
```

The address field is protocol–dependent.

Protocol description of address field

This is the `sockaddr` structure for TCP:

```
struct sockaddr_in {
    short sin_family ; // = AF_INET = PF_INET
    u_short sin_port ; // port number
    struct in_add sin_addr ; // IP address - 4 bytes
    char sin_zero[8] ; // nothing
} ;
```

An IP address INADDR_ANY should be used (unless the machine has several IP addresses and we want to restrict incoming messages only to those using one of the addresses). Likewise, INADDR_ANY can be used in the port field.

Starting a TCP server

```
listen( desc , queuesize ) ;
```

The queuesize argument gives the maximum number of pending connect() requests.

listen() activates the port and makes it wait for incoming connect() requests. Note that listen() terminates as soon as it activates the port.

If the protocol is connectionless, listen() does not do anything good.

A new TCP session

A session has to be started by creating a virtual circuit, so listen() should be followed by a handshake.

```
    newsocket = accept( desc , client_addr , int *cl_addr_len ) ;
```

client_addr is of type (struct sockaddr *). accept() returns in it the address of the client (with its length returned in cl_addr_len).

Now the server process is ready to accept messages from this particular client using the newsocket socket returned by accept(). When the circuit is closed, this is the socket to be closed, not the one created by socket().

The code in the server

A connection–oriented server works like this:

```
getprotobyname( ... ) ;
s_sock = socket( ... ) ;
bind( s_sock , .. ) ;
listen( s_sock , .. ) ;
// server is running now
cl_sock = accept( s_sock , .. ) ;
recv( cl_sock , ... ) ;
send( cl_sock , ... ) ;
close( cl_sock ) ;
```

The part between accept() and close() is a complicated loop, often involving a call to select().

Client starts TCP session

returncode = connect(sock , server_address , server_addresslen) ;

Client side

```
sock = socket( protofamily , type , protocol ) ;
struct sockaddr_in server_address ;
server_address.sin_family = AF_INET ;
// fill the IP address here
server_address.sin_port = htons( SRV_TCP_PORT ) ;
returncode = connect( sock , server_address , server_addresslen ) ;
send( sock , data_address , length , flags ) ;
.........
```

connect() is needed for connection–oriented sessions only. It also works for connectionless service allowing to use send() instead of sendto() (unclear what for).

Byte ordering

The Internet protocols require numeric values to be passed in a specified byte order which happens to be different than the host ordering of many machines (e.g. Intel).

To convert from/to host to/from network order use:

```
u_long htonl( u_long hostlong ) ;
u_short htons( u_short hostshort ) ;
u_long ntohl( u_long netlong ) ;
u_short ntohs( u_short netshort ) ;
```

These functions always work; for portability, they cannot be omitted.

Names and addresses

struct hostent *gethostbyname(char *host) ;

struct *getservbyname(char *servname , char *protocol) ;

shutdown()

Another function that belongs to the Socket API is shutdown() which was designed to allow closing one direction of the two–directional stream between the two hosts.