```
#include<fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>


#define FILELENGTH 511  // This is not right
#define DEPTH 1
#define PACKETSIZE 1023

int lsd = 0 ;   // socket for establishing connections

#define SA_SIZE sizeof( struct sockadrr_in )
```

```
unsigned long atoip( char *text )
{
    unsigned long ip ;
    int i , t ;
    i = 0 ;
    ip = t = 0 ;

    while( text[i] != '\0' ) {
        if( text[i] == '.' ) {
            ip = (ip<<8) + t ;
            t = 0 ;
        } else
            t = t*10 + text[i] - '0' ;
        i++ ;
    }
    return htonl( (ip<<8) + t ) ;
}
```

```
struct sockaddr_in *sa ;
int rc ;
sa = (struct sockaddr_in *)malloc( SA_SIZE ) ;
sa->sin_family = AF_INET ;
sa->sin_port = htons(4950) ;
sa->sin_addr.s_addr = atoip( "127.0.0.1" ) ;
lsd = socket( PF_INET , SOCK_STREAM , 0 ) ;
if( lsd <= 0 ) {
    perror( "Socket not created" ) ;
    kill( getpid() , SIGINT ) ;
}
rc = bind( lsd , (struct sockaddr *) sa , SA_SIZE ) ;
if( rc == -1 ) {
    perror( "Bind unsuccessful" ) ;
    kill( getpid() , SIGINT ) ;
}
listen( lsd , 5 ) ;
```

```
fd_set lsmask ;
int new_client ;
struct timeval timeout ;
FD_ZERO( &lsmask ) ;
FD_SET( lsd , &lsmask ) ;
timeout.tv_sec = 0 ;
timeout.tv_usec = 10000 ;  // 10 msec
new_client = select( lsd+1 , &lsmask
        , NULL , NULL , &timeout ) ;
if( new_client ) {
    new_client = accept( lsd , NULL , NULL ) ;
    if( new_client == −1 ) {
        perror( "Accept failed" ) ;
        kill( getpid() , SIGINT ) ;
    }
}
```

```
rc = recv( cls , buffer , 200 , 0 ) ;
buffer[rc] = '\0' ;     // Just in case
sscanf( buffer , "%s %s\n%s" , user , pwd , dir ) ;
FILE *PF = fopen( "./.pwd" , "rw" ) ;
if( PF == NULL ) {
    PF = fopen( "./.pwd" , "w+" ) ;
    if( PF == NULL ) {
        perror( "Failed to create ./pwd" ) ;
        kill( getpid() , SIGINT ) ;
    }
    fprintf( PF , "%s %s\n" , user , pwd ) ;
} else {
    fseek( PF , 0 , SEEK_SET ) ;
    while( (rc = fscanf( PF , "%s %s\n" , PU , PP )) > 0 )
        if( strcmp( PU , user ) == 0 ) break ;
    if( rc <= 0 ) {
        fprintf( PF , "%s %s\n" , user , pwd ) ;
    } else if( strcmp( PP , pwd ) != 0 )
        kill( getpid() , SIGINT ) ;
}
```

```
void backup_directory( int cls , char *dir )
{
    char buffer[FILELENGTH] , file[FILELENGTH] ;
    int size , rc ;
    while( (rc = recv( cls , buffer , FILELENGTH , 0 )) > 0 ) {
        buffer[rc] = '\0' ;
        sscanf( buffer , "%s %d" , file , &size ) ;
        send( cls , "ACK" , 4 , 0 ) ;
        copy( cls , file , size ) ;
        send( cls , "ACK" , 4 , 0 ) ;
    }
    close( cls ) ;
}
```

```
int start_connection()
{
   struct sockaddr_in *sa ;
   int rc , sd ;
   sa = (struct sockaddr_in *)malloc( SA_SIZE ) ;
   memset( (char *)sa , 0 , SA_SIZE ) ;
   sa->sin_family = AF_INET ;
   sa->sin_port = htons(4950) ;
   sa->sin_addr.s_addr = htonl( atoip("127.0.0.1") ) ;
   sd = socket( PF_INET , SOCK_STREAM , 0 ) ;
   if( sd <= 0 ) {
      perror( "Socket not created" ) ;
      kill( getpid() , SIGINT ) ;
   }
   rc = connect( sd , (struct sockaddr *) sa , SA_SIZE ) ;
   if( rc == -1 ) {
      perror( "Connect failed" ) ;
      kill( getpid() , SIGINT ) ;
   }
   return sd ;
}
```

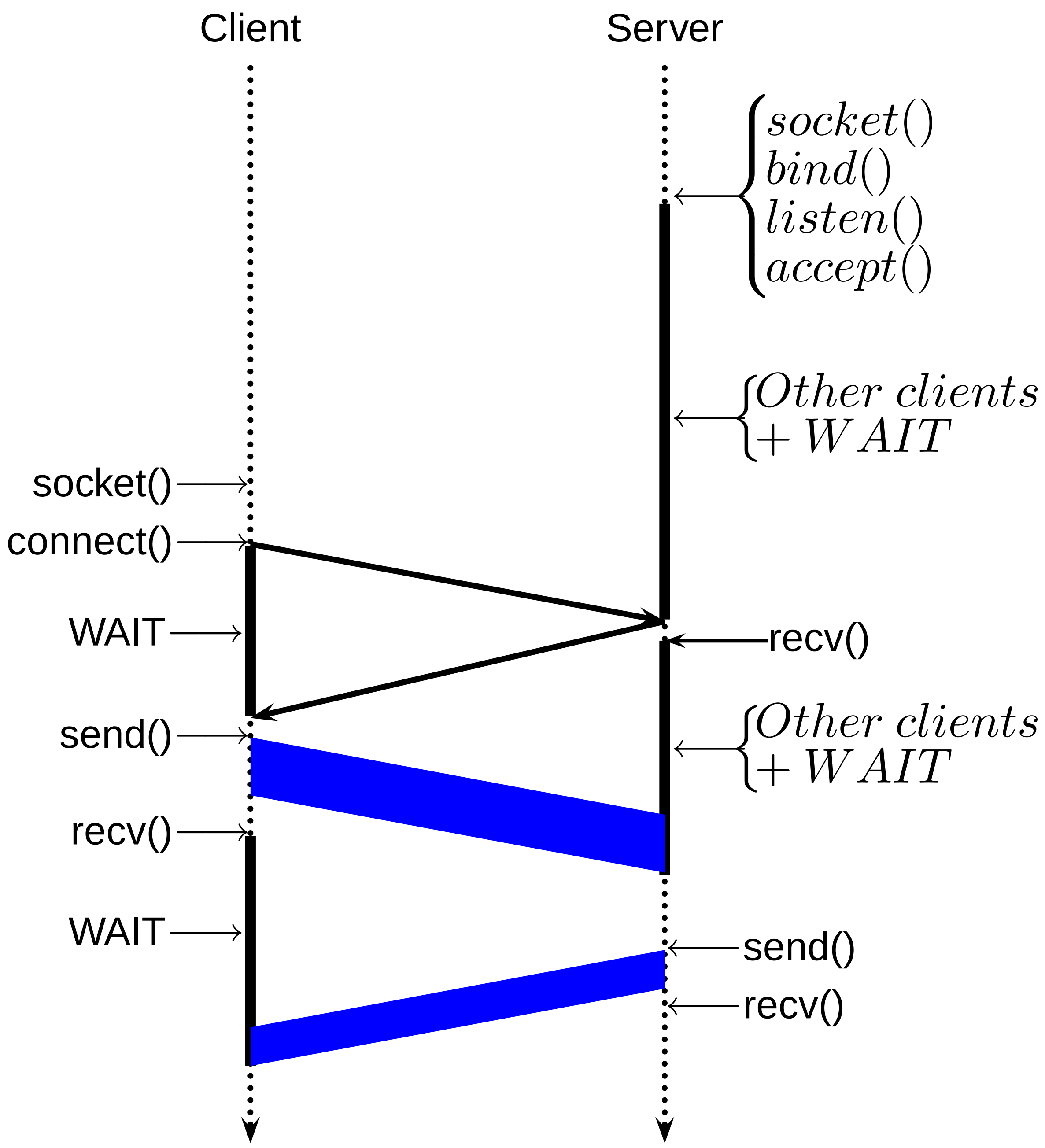
TCP Client/Server

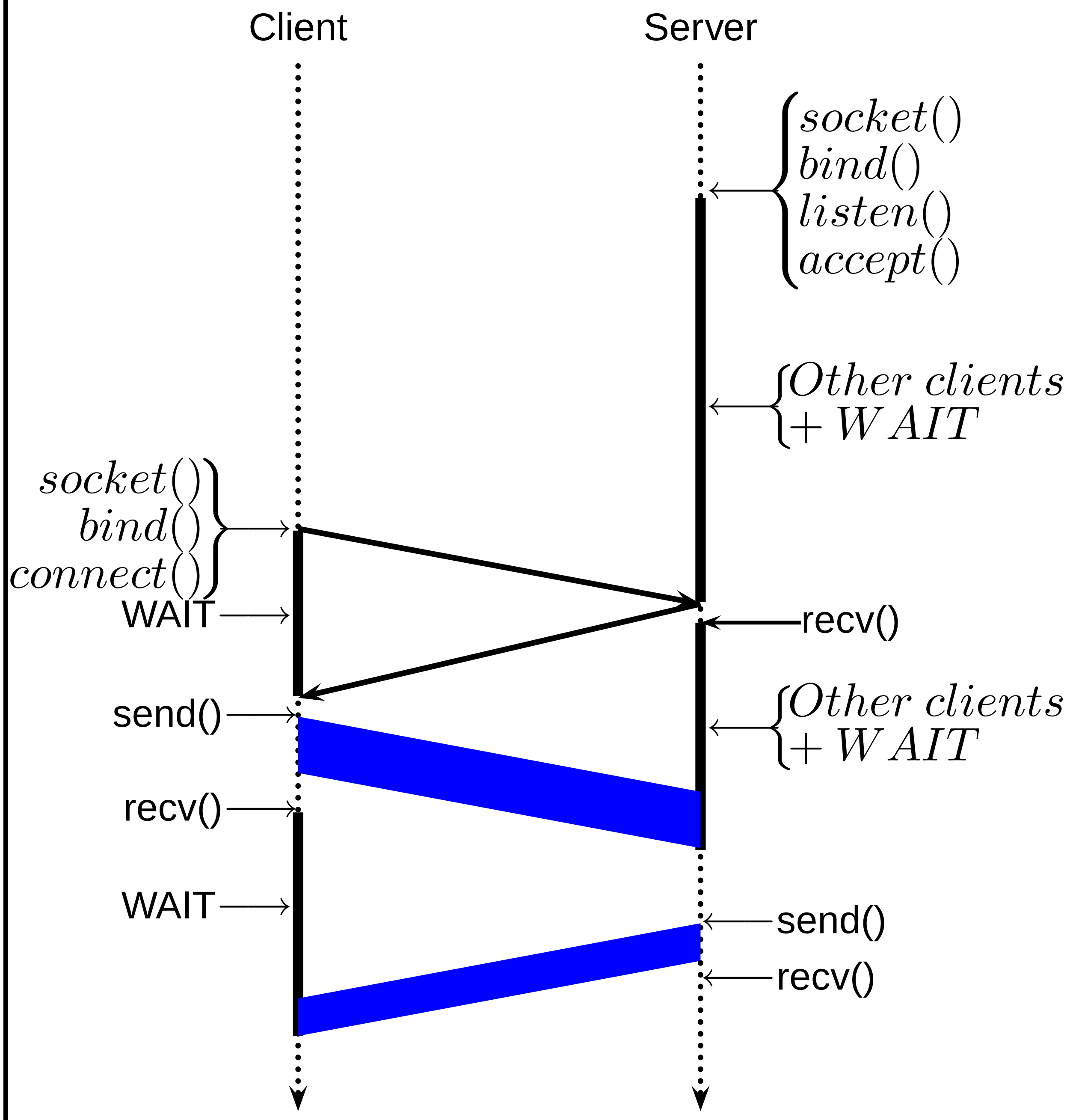
Typically, the API is a client–server interface. Its library covers the functionality of both of them.

TCP

Client	Server
socket()	socket()
bind() (?)	bind()
	listen()
connect()	accept()
[send() + recv()]*	[send() + 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()
send()
$Other\ clients$
$+WAIT$
recv()
WAIT
send()
recv()

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.

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 ) ;
.........
```

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) ;