CS 111
Operating Systems Principles
Section 1E Week 3

Tengyu Liu

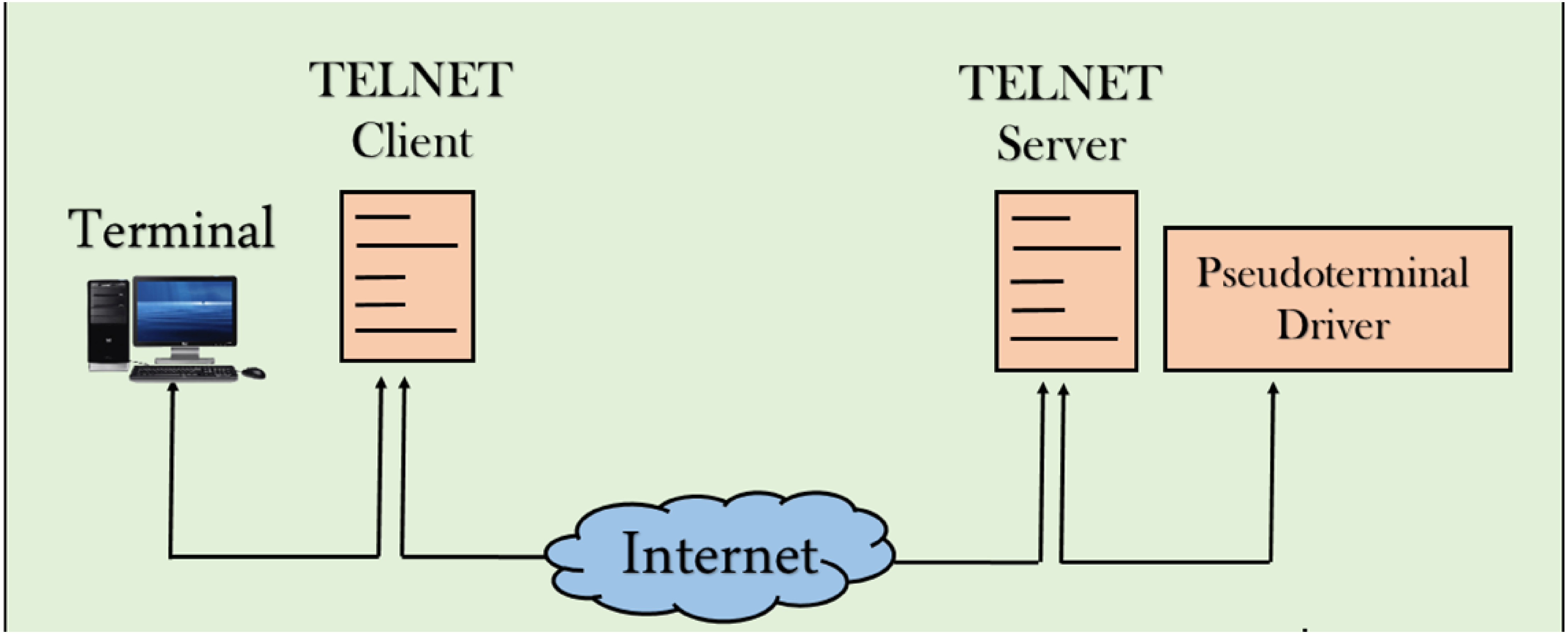
Changes

- Slides are now uploaded to CCLE every Wednesday
  - I will update the slides before each discussion section
- I will explain the overall picture of each project
- I will include more code examples

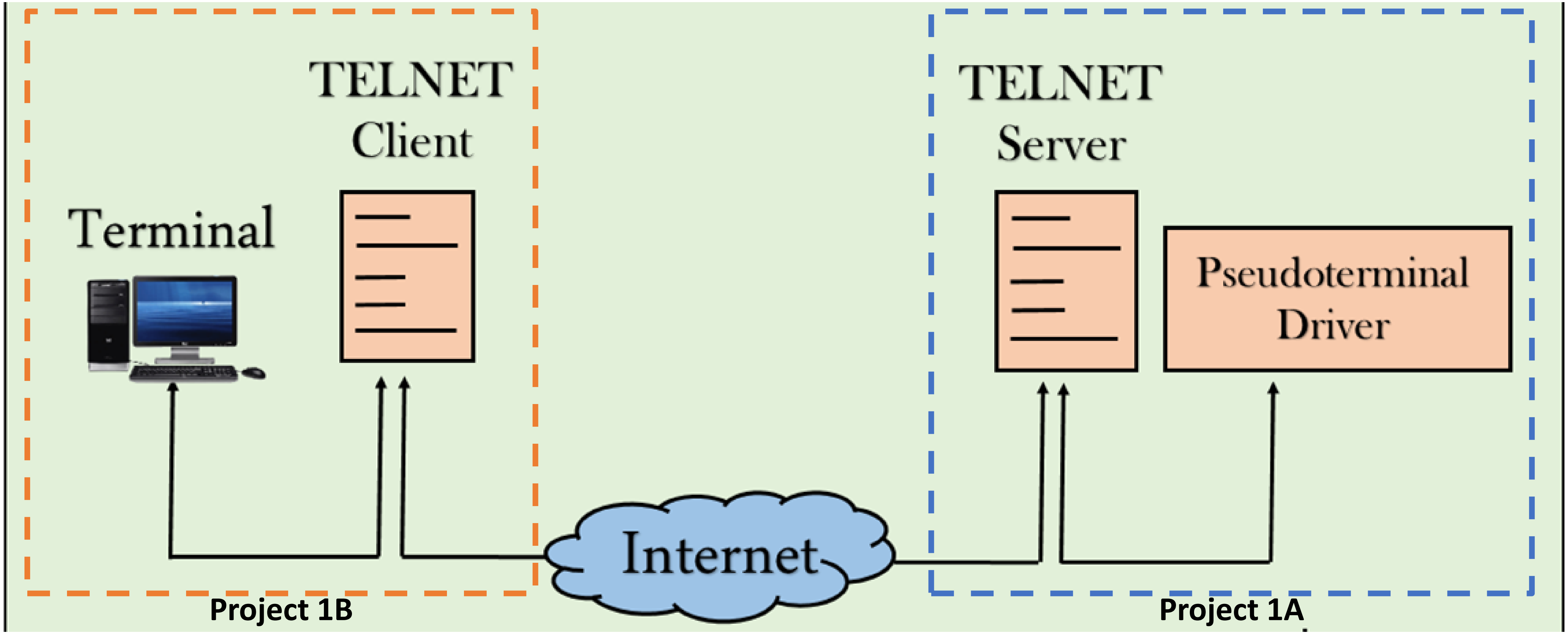
Overview

- Project 1B: Inter-Process Communication over Internet
- System calls:
  - Internet communication
    - `socket(2), connect(2), bind(2), listen(2), accept(2), shutdown(2)`
  - Compressed Communication
    - `deflateInit(), inflateInit(), deflate(), inflate(), deflateEnd(), inflateEnd()`

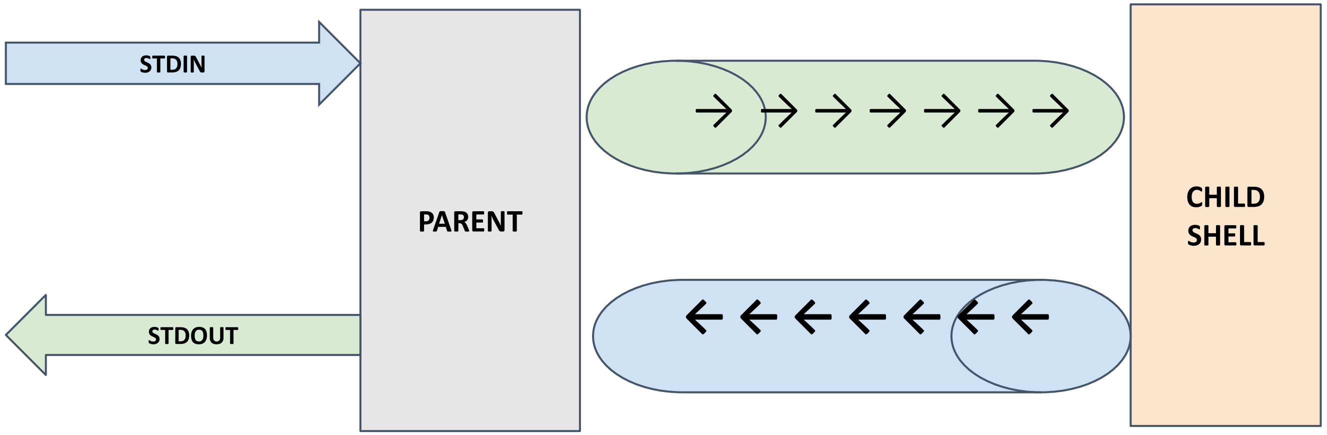
Project 1B


TELNET
Client
TELNET
Server
Terminal
Pseudoterminal
Driver
Internet

Project 1B


TELNET
Client
Terminal
TELNET
Server
Pseudoterminal
Driver
Internet
Project 1B
Project 1A

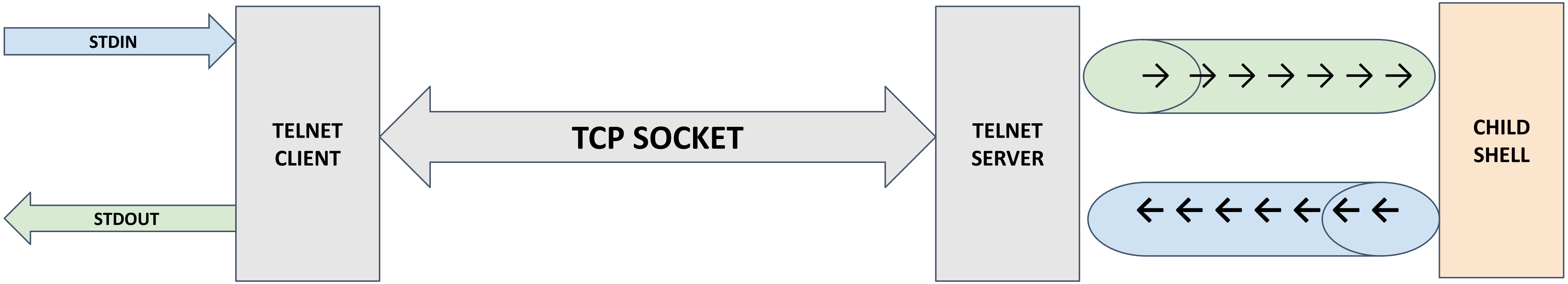
Project 1A



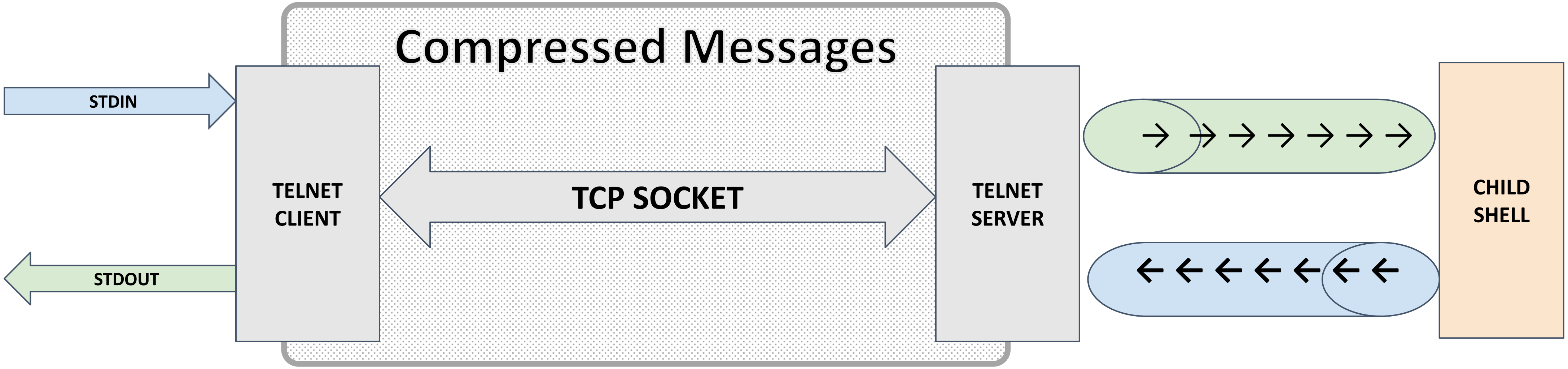
🟢 Output from parent

🔵 Input to parent

Project 1B


STDIN
STDOUT
TELNET CLIENT
TCP SOCKET
TELNET SERVER
CHILD SHELL

Project 1B `--compress`



TCP Socket

TCP Socket

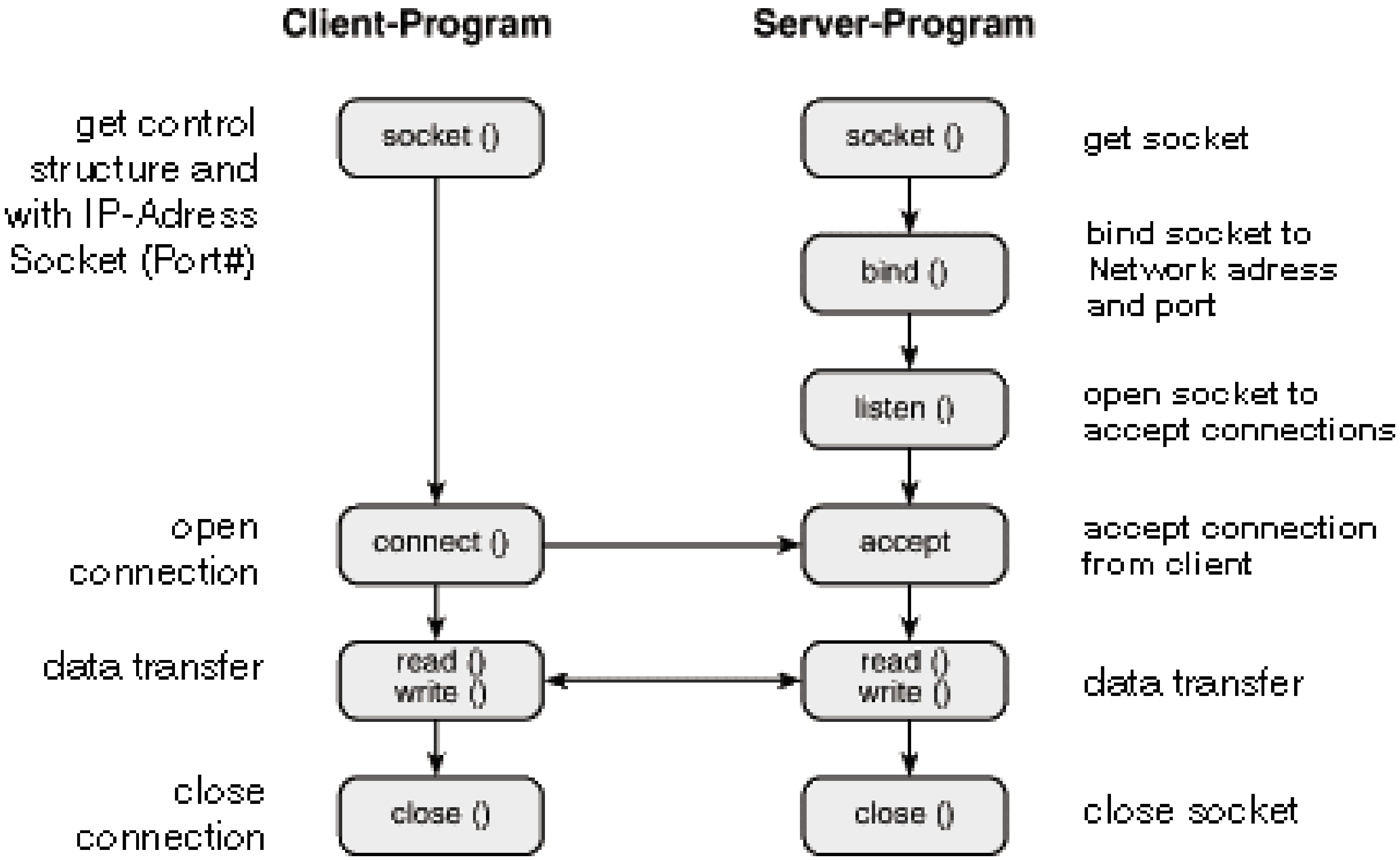
Pipe

- Both ends must live on the same machine
- Returns 2 file descriptors, one for reading, one for writing

Socket

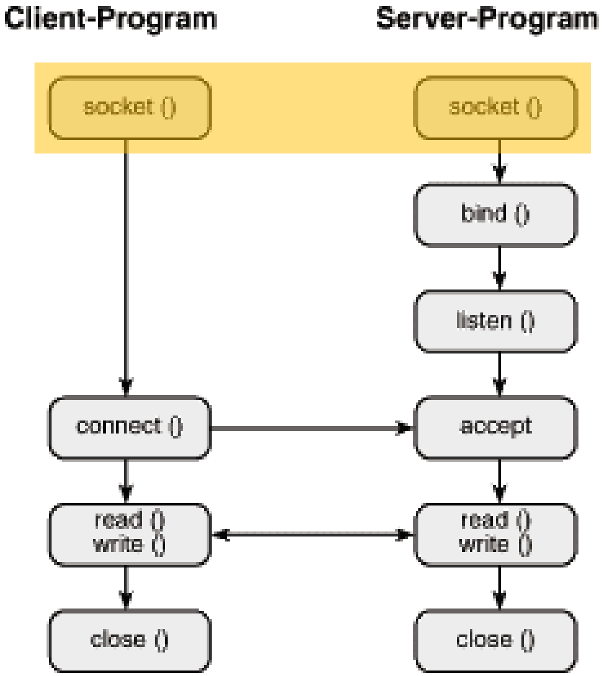
- Makes no assumption on process location
- Returns only 1 file descriptor, for both reading and writing
- Socket needs more information
  - Protocol
  - Target address

TCP Socket: Pipeline Overview



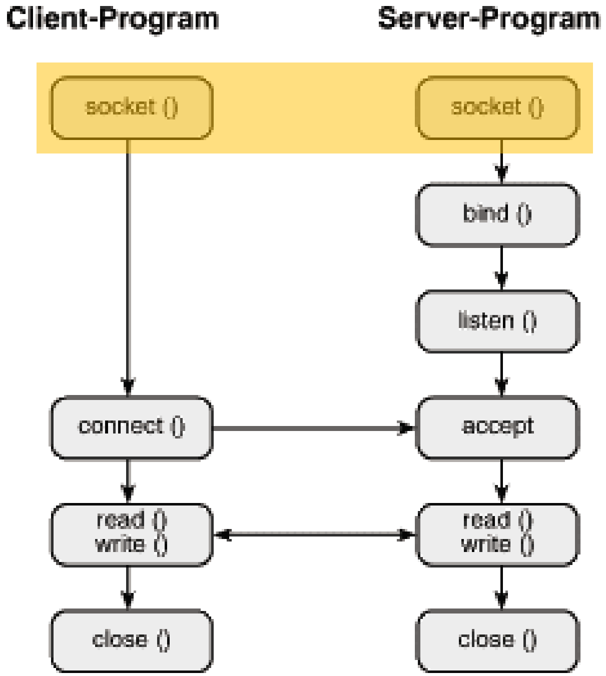
socket(2)

- Create an endpoint for communication
- `int socket(int domain, int type, int protocol)`



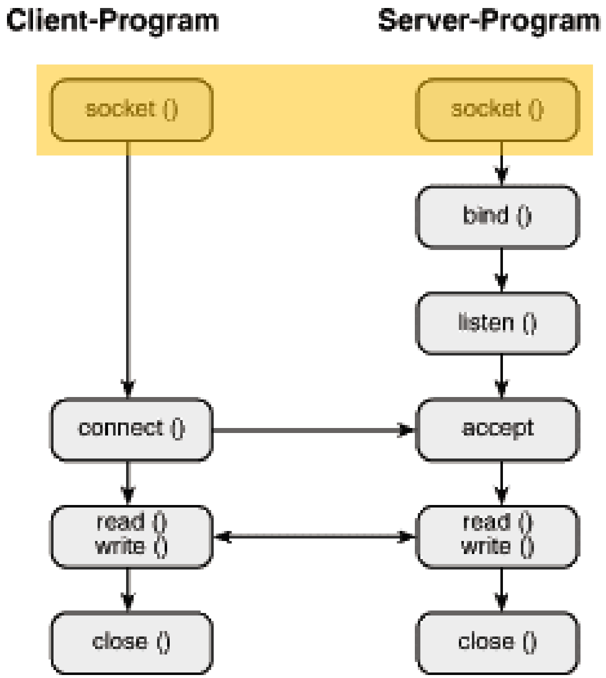
socket(2)

- Create an endpoint for communication
- `int socket(`**`int domain`**`, int type, int protocol)`
  - Communication domain
    - `IPv4: AF_INET`
    - `IPv6: AF_INET6`
    - `Local: AF_LOCAL`



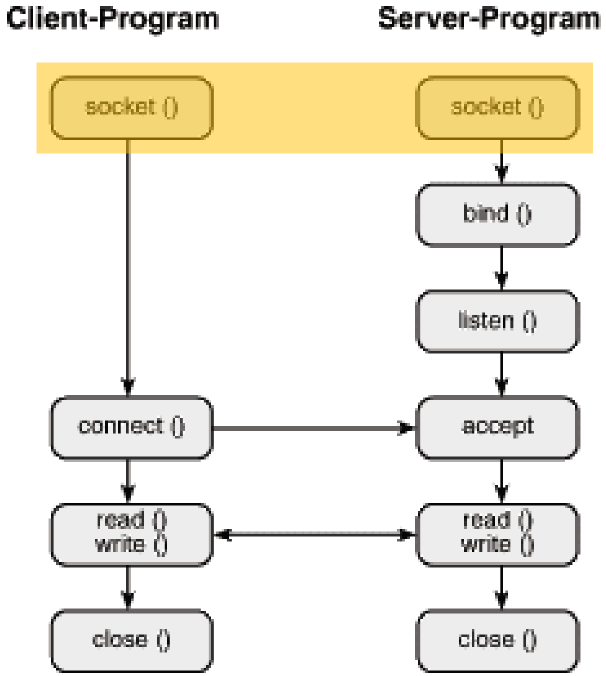
socket(2)

- Create an endpoint for communication
- `int socket(int domain, `**`int type`**`, int protocol)`
    - Communication type
        - `TCP: SOCK_STREAM`
        - `UDP: SOCK_DGRAM`



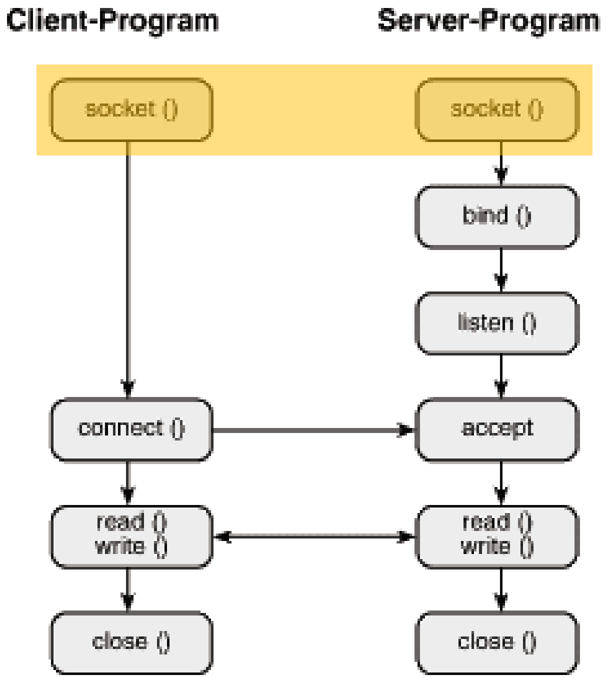
socket(2)

- Create an endpoint for communication
- `int socket(int domain, int type,` **`int protocol`**`)`
  - IP protocol value
    - Usually there is only one protocol available for each domain
    - Use 0 to use the default



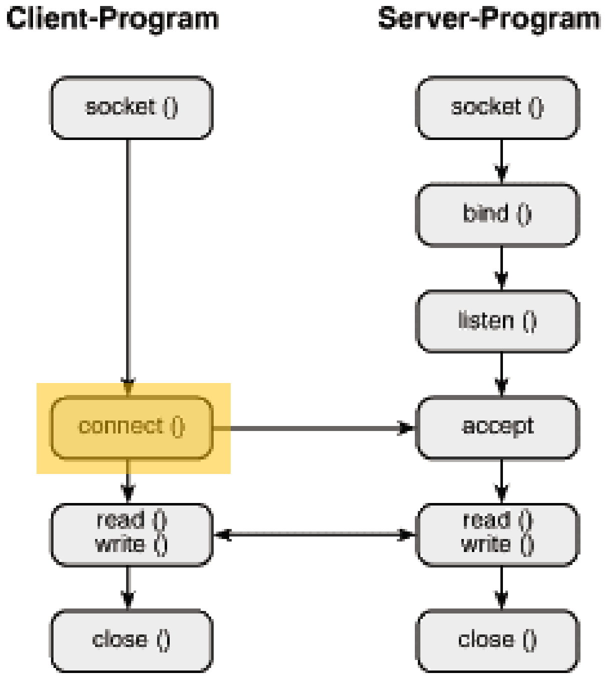
socket(2)

- Create an endpoint for communication
- **`int socket`**`(int domain, int type, int protocol)`
  - Return value: socket descriptor
  - On error: returns -1



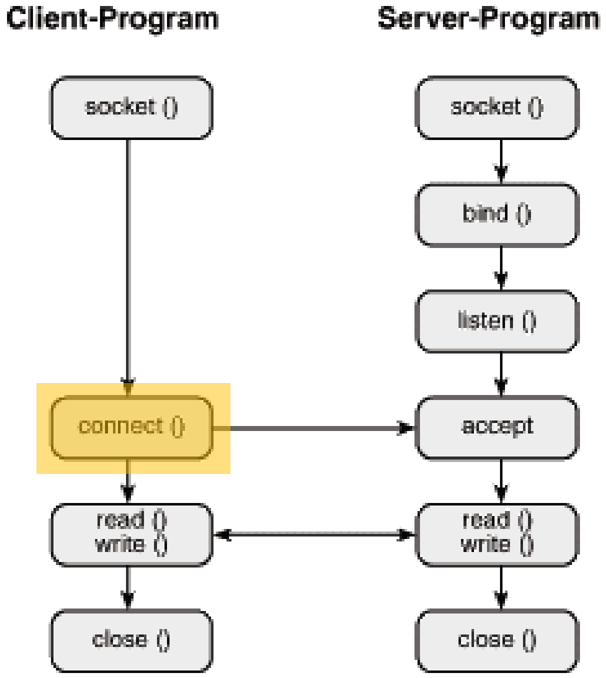
connect(2)

- Connect socket to a remote host
- `int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`



connect(2)

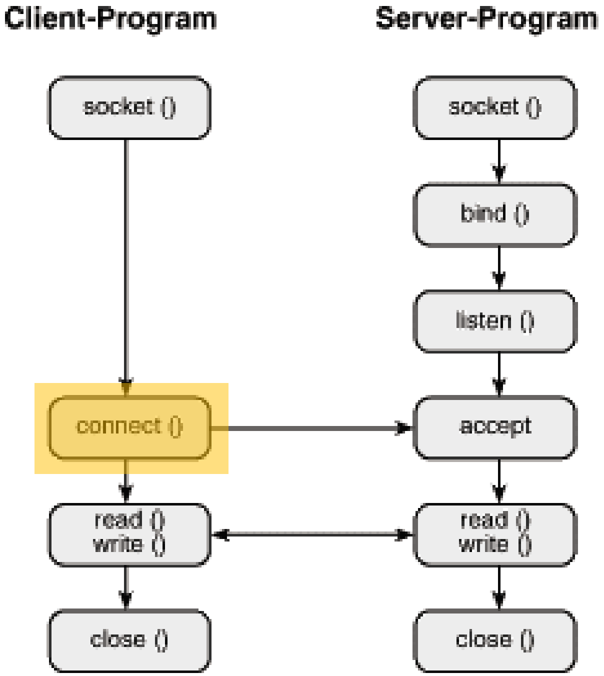
- Connect socket to a remote host
- `int connect(`**`int sockfd`**`, const struct sockaddr *addr, socklen_t addrlen);`
  - Socket for connect (returned by `socket(2)`)



connect(2)

- Connect socket to a remote host
- `int connect(int sockfd, `**`const struct sockaddr *addr`**`, socklen_t addrlen);`
  - Structure containing server's IP address and port number

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};
```

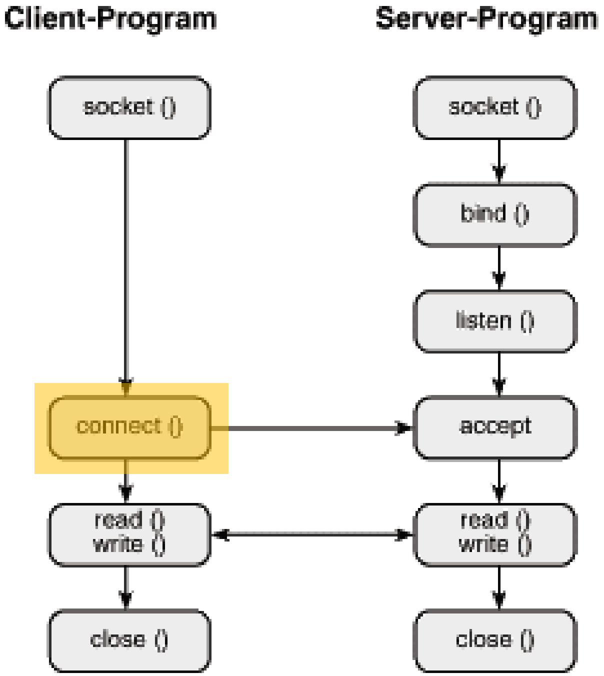


connect(2)

- Connect socket to a remote host
- int connect(int sockfd, **const struct sockaddr *addr**, socklen_t addrlen);
  - Structure containing server’s IP address and port number

AF_INET

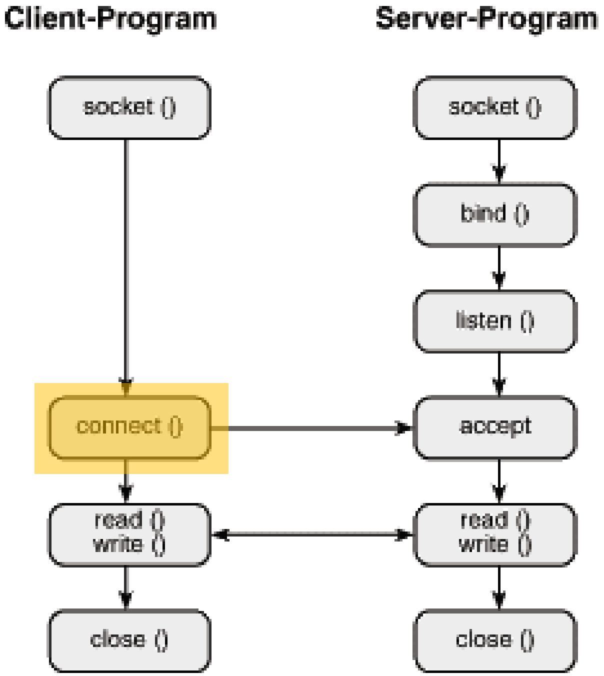
```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};
```



connect(2)

- Connect socket to a remote host
- int connect(int sockfd, **const struct sockaddr *addr**, socklen_t addrlen);
    - Structure containing server’s IP address and port number

```
             struct sockaddr_in {
                 sa_family_t    sin_family; /* address family: AF_INET */
Port number  →   in_port_t      sin_port;   /* port in network byte order */
                 struct in_addr sin_addr;   /* internet address */
             };

             /* Internet address. */
             struct in_addr {
                 uint32_t       s_addr;     /* address in network byte order */
             };
```

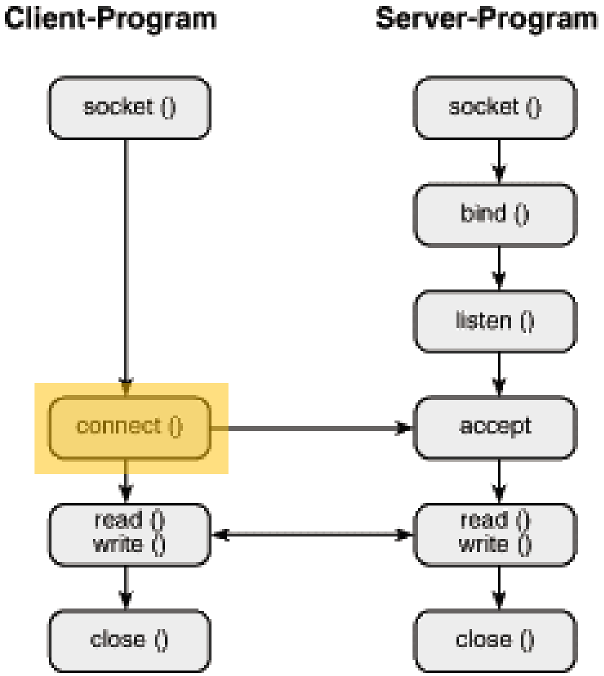


connect(2)

- Connect socket to a remote host
- int connect(int sockfd, **const struct sockaddr *addr**, socklen_t addrlen);
  - Structure containing server’s IP address and port number

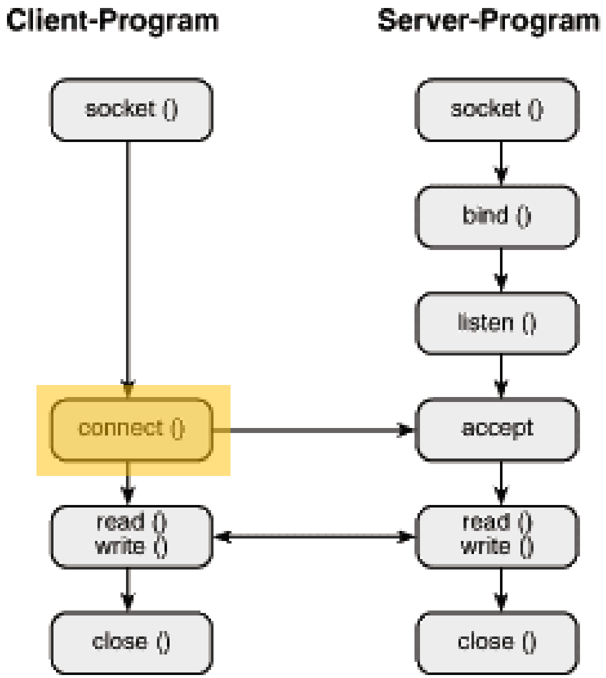
```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};
```

Obtained by gethostbyname(3)



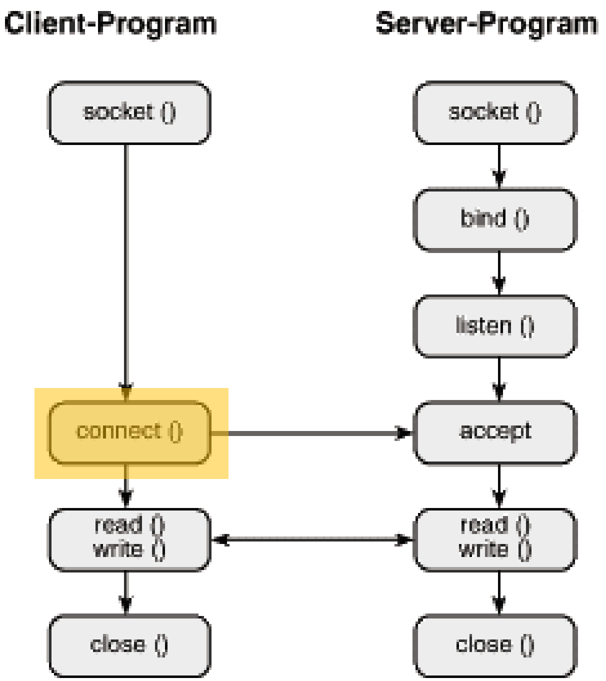
connect(2)

- Connect socket to a remote host
- `int connect(int sockfd, const struct sockaddr *addr,` **`socklen_t addrlen`**`);`
  - `sizeof(*addr)`



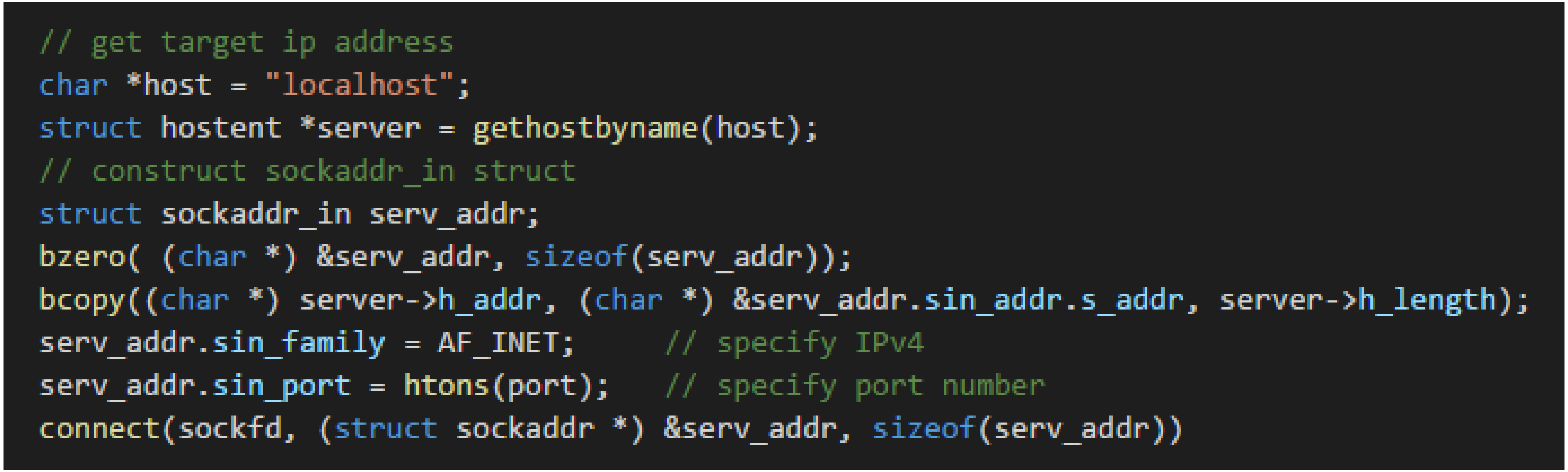
connect(2)

- Connect socket to a remote host
- **int connect**(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
    - Returns the socket descriptor
    - On error, return -1



connect(2)

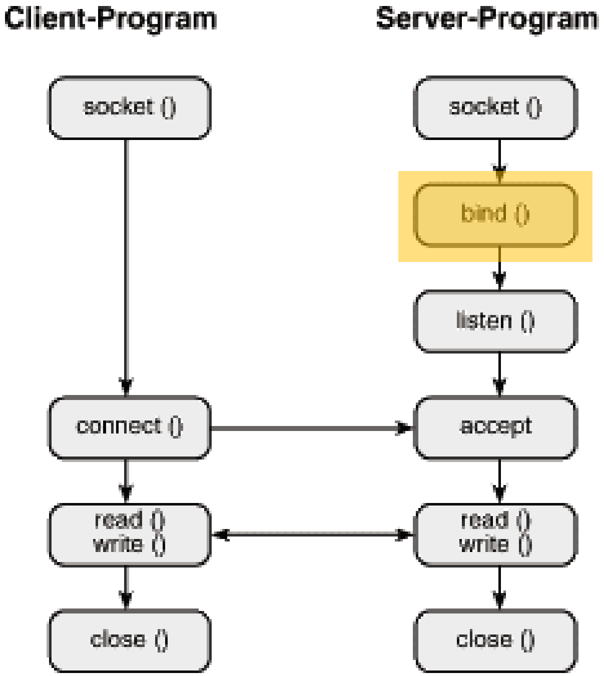
- `int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
- Example (error checks are omitted)



```c
// get target ip address
char *host = "localhost";
struct hostent *server = gethostbyname(host);
// construct sockaddr_in struct
struct sockaddr_in serv_addr;
bzero( (char *) &serv_addr, sizeof(serv_addr));
bcopy((char *) server->h_addr, (char *) &serv_addr.sin_addr.s_addr, server->h_length);
serv_addr.sin_family = AF_INET;     // specify IPv4
serv_addr.sin_port = htons(port);   // specify port number
connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
```

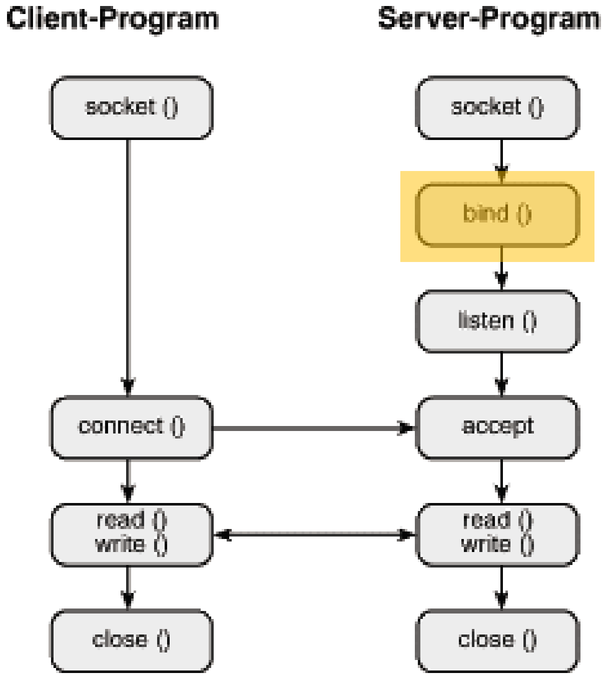
bind(2)

- Bind a name to a socket
- `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`



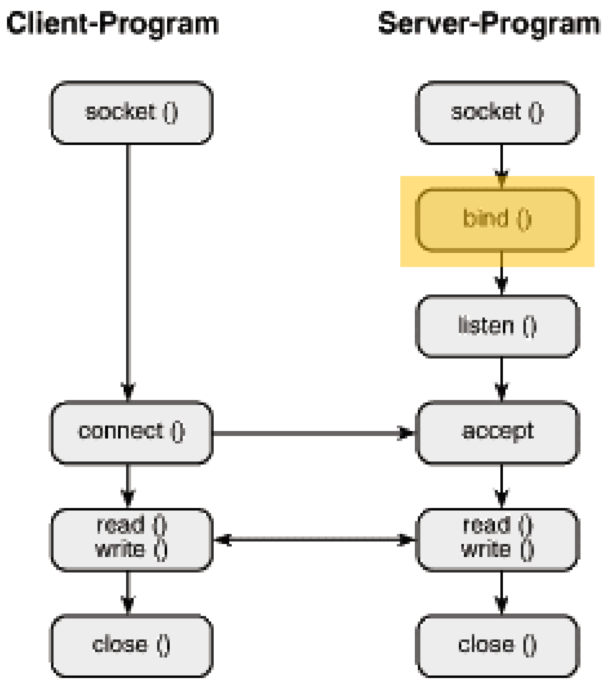
bind(2)

- Bind a name to a socket
- `int bind(`**`int sockfd`**`, const struct sockaddr *addr, socklen_t addrlen);`
  - Socket descriptor



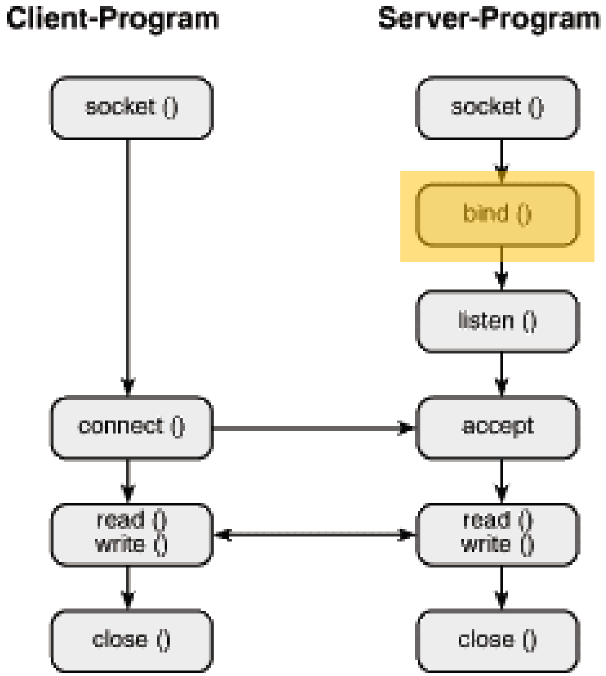
bind(2)

- Bind a name to a socket
- `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
  - IP Address and Port



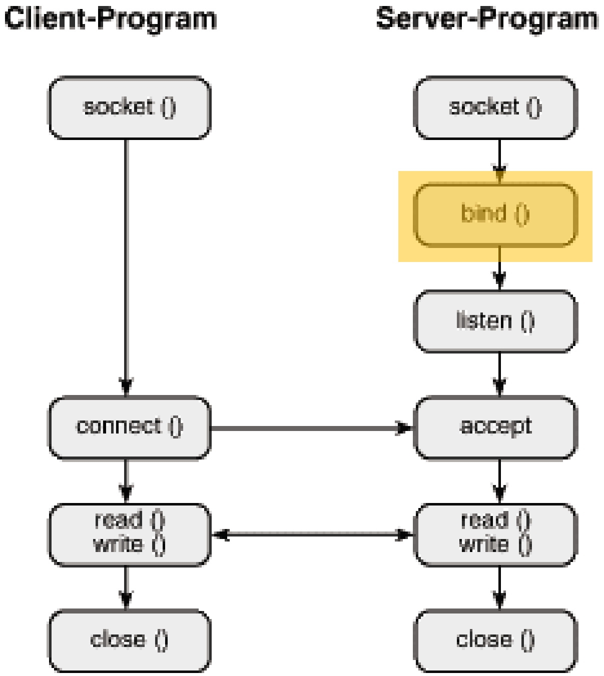
bind(2)

- Bind a name to a socket
- `int bind(int sockfd, const struct sockaddr *addr,` **`socklen_t addrlen`**`);`
  - `sizeof(*addr)`



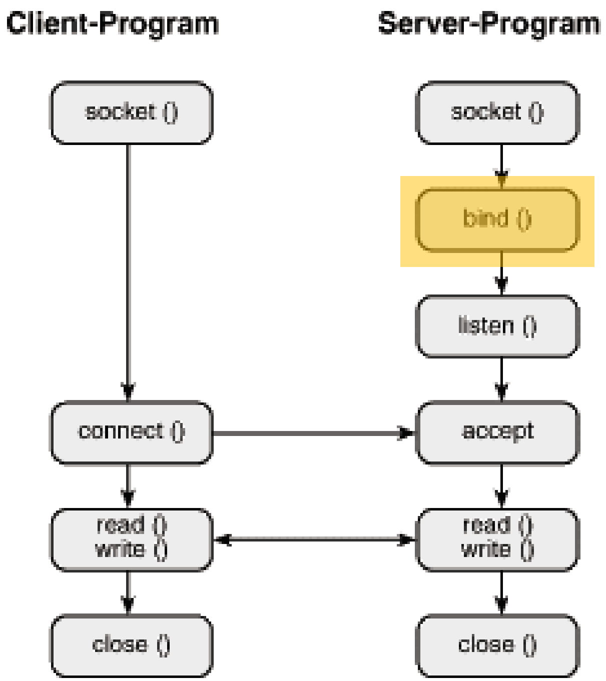
bind(2)

- Bind a name to a socket
- **`int bind`**`(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
  - Socket descriptor
  - On error, return -1

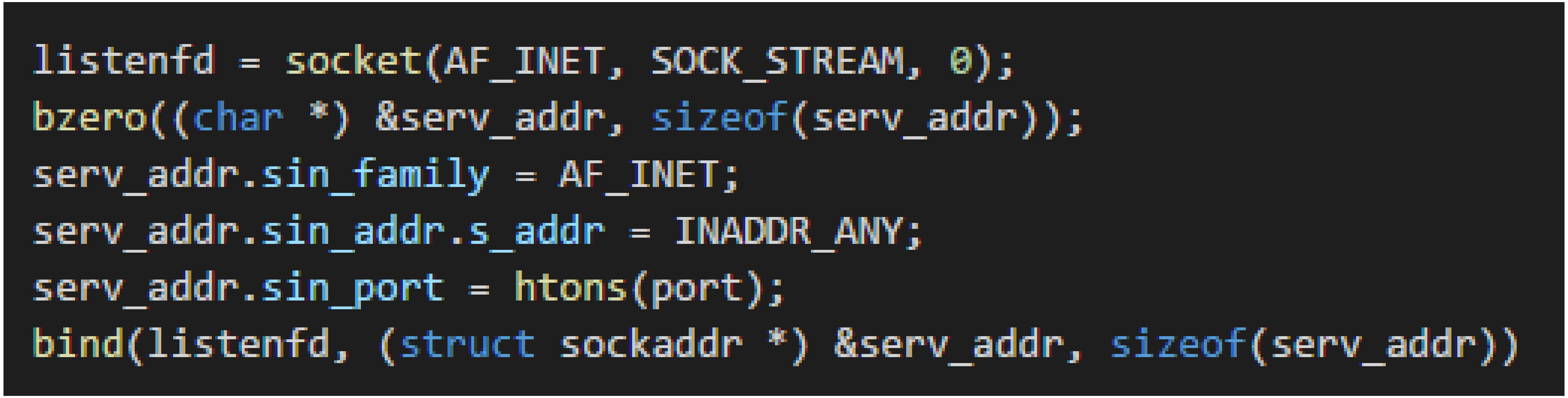


bind(2)

- Bind a name to a socket
- `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
  - This function effectively gives a port to the socket
  - IP address: address of the machine
  - Port number: address of the process
  - Without a port number, messages cannot reach your process



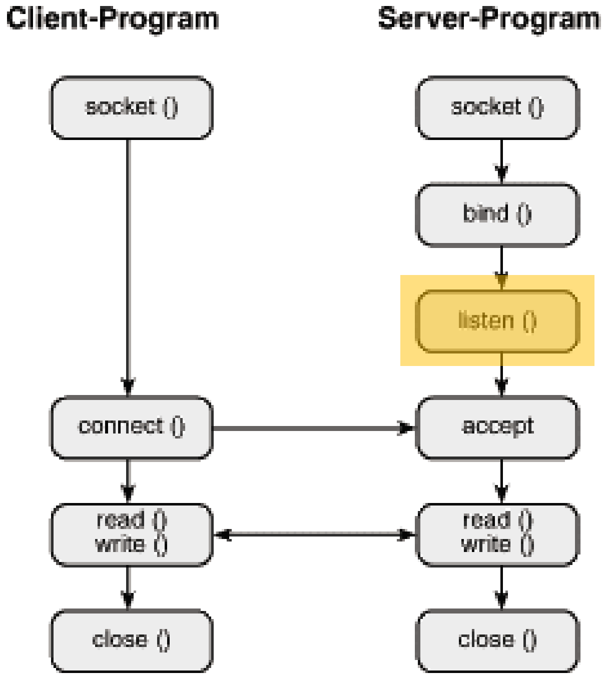
bind(2)



```c
listenfd = socket(AF_INET, SOCK_STREAM, 0);
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(port);
bind(listenfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
```

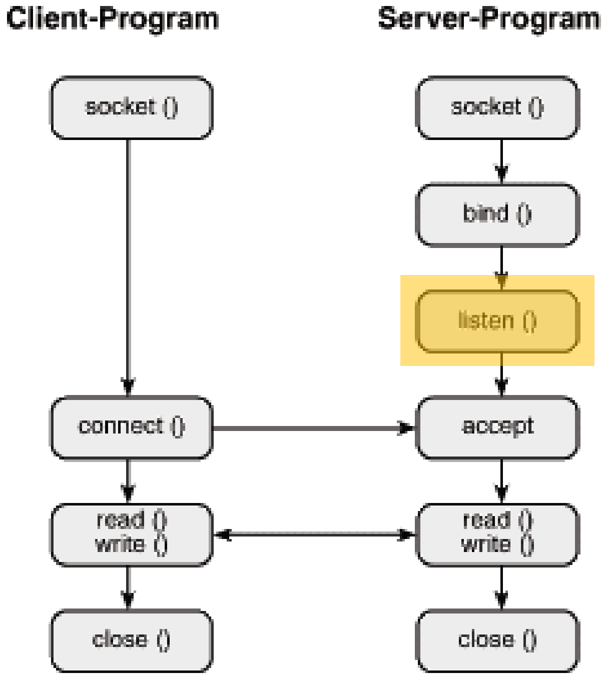
listen(2)

- Start listening for incoming connections
- `int listen(int sockfd, int backlog)`



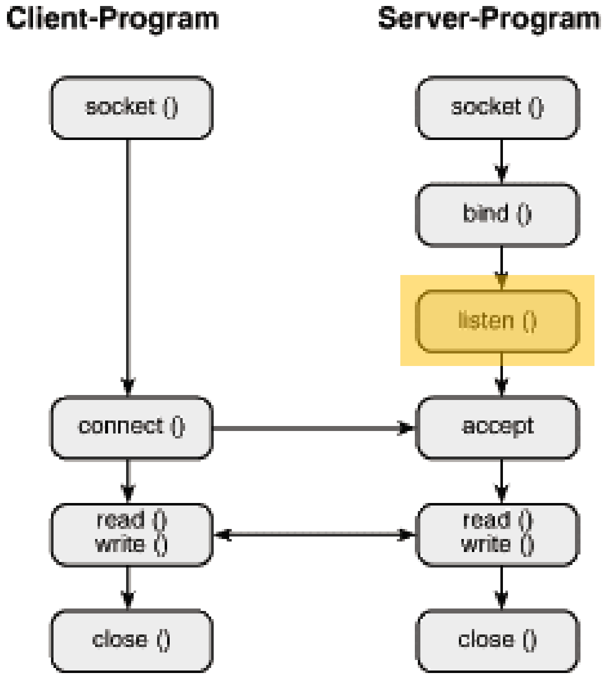
listen(2)

- Start listening for incoming connections
- `int listen(`**`int sockfd`**`, int backlog)`
  - Socket file descriptor



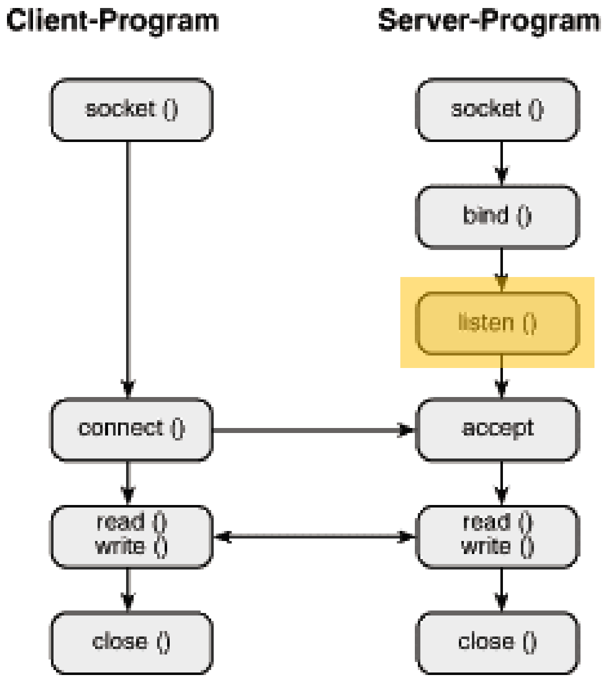
listen(2)

- Start listening for incoming connections
- `int listen(int sockfd, int backlog)`
  - Number of connections allowed in the incoming queue
  - On most systems, max allowed is 5
  - Use 5



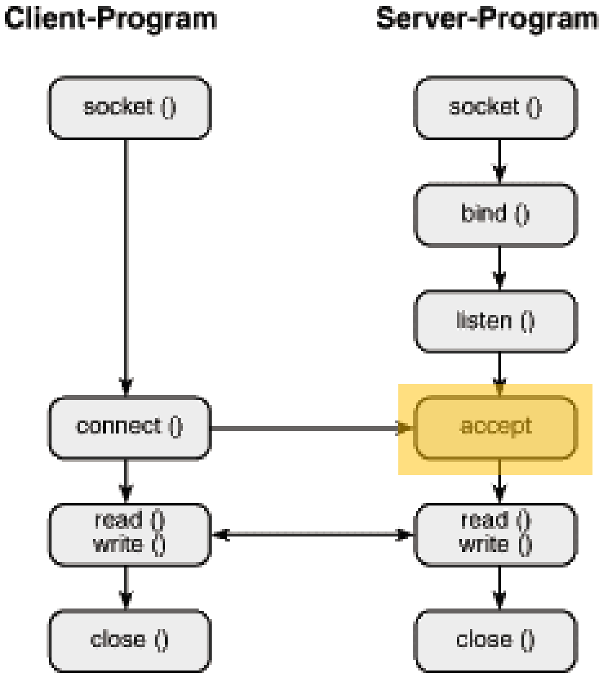
listen(2)

- Start listening for incoming connections
- **`int listen`**`(int sockfd, int backlog)`
  - Success: `0`
  - Error: `1`



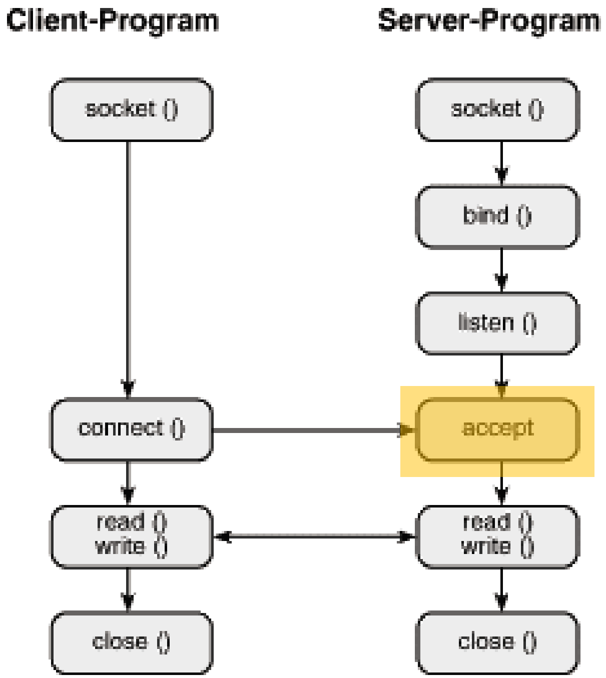
accept(2)

- Accept a connection on a socket (blocking until a connection is accepted)
- `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)`



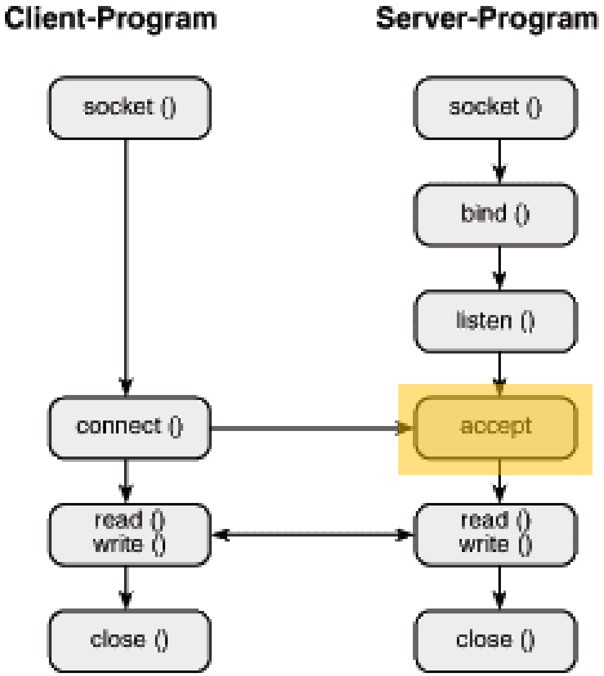
accept(2)

- Accept a connection on a socket (blocking until a connection is accepted)
- `int accept(`**`int sockfd`**`, struct sockaddr *addr, socklen_t *addrlen)`
  - Socket descriptor that we listened on



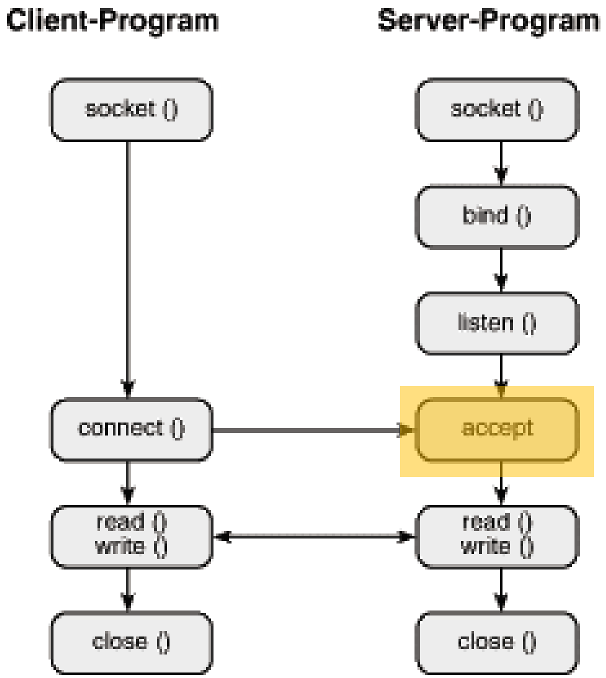
accept(2)

- Accept a connection on a socket (blocking until a connection is accepted)
- `int accept(int sockfd, `**`struct sockaddr *addr`**`, socklen_t *addrlen)`
  - Output parameter
  - Contains information about the incoming connection



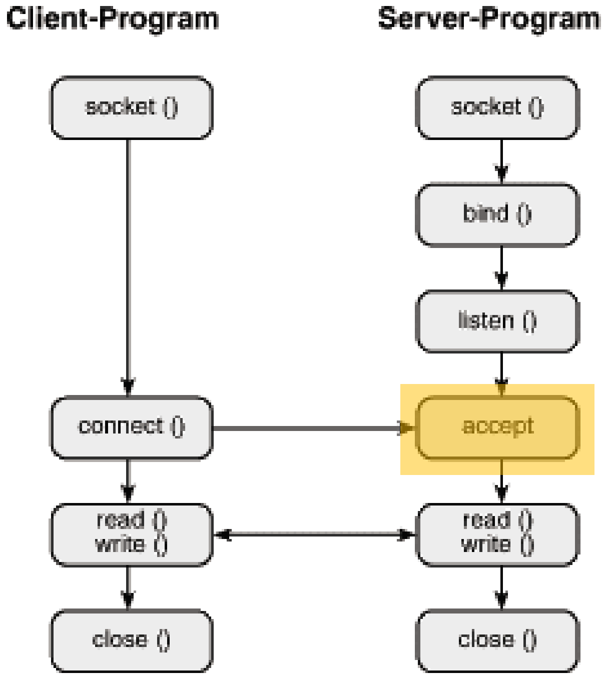
accept(2)

- Accept a connection on a socket (blocking until a connection is accepted)
- `int accept(int sockfd, struct sockaddr *addr,` **`socklen_t *addrlen`**`)`
  - Length of the incoming `sockaddr struct`



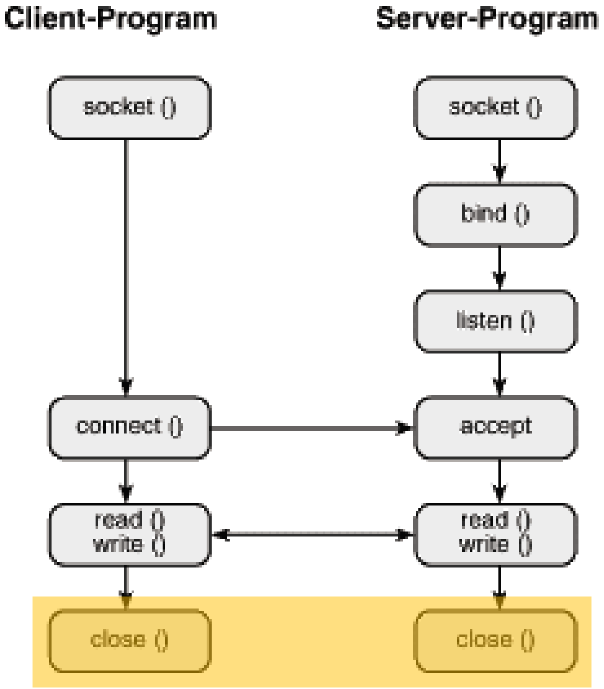
accept(2)

- Accept a connection on a socket (blocking until a connection is accepted)
- **`int accept`**`(int sockfd, struct sockaddr *addr, socklen_t *addrlen)`
  - Success
    - New socket file descriptor to use for this connection
    - The old descriptor is only for accepting incoming connections
    - A new descriptor is allocated for every connection established
  - Error
    - -1



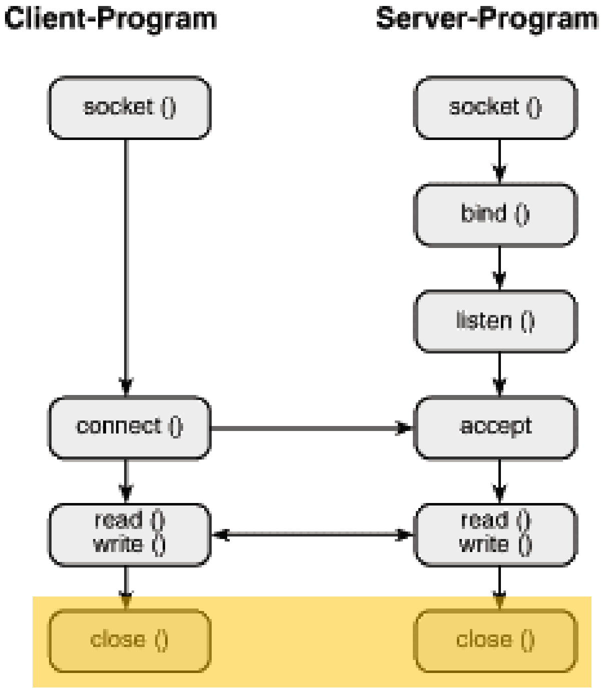
shutdown(2)

- Shut down socket
- `int shutdown(int socket, int how)`



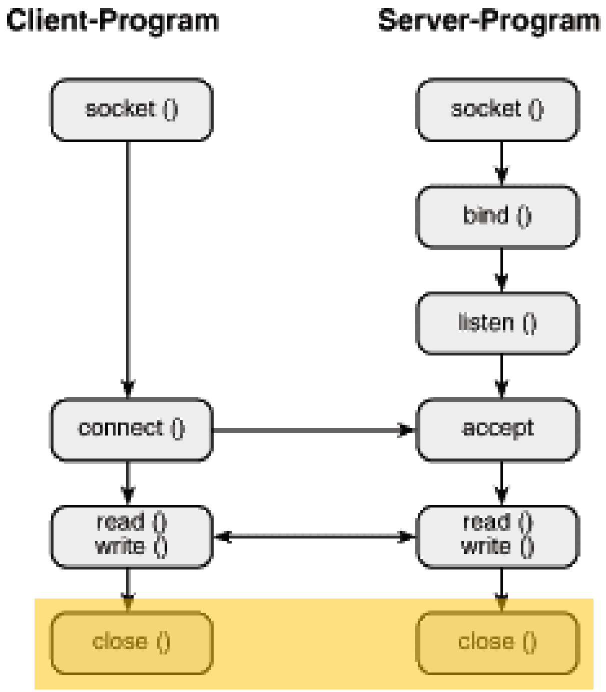
shutdown(2)

- Shut down socket
- `int shutdown(`**`int socket`**`, int how)`
  - Socket descriptor



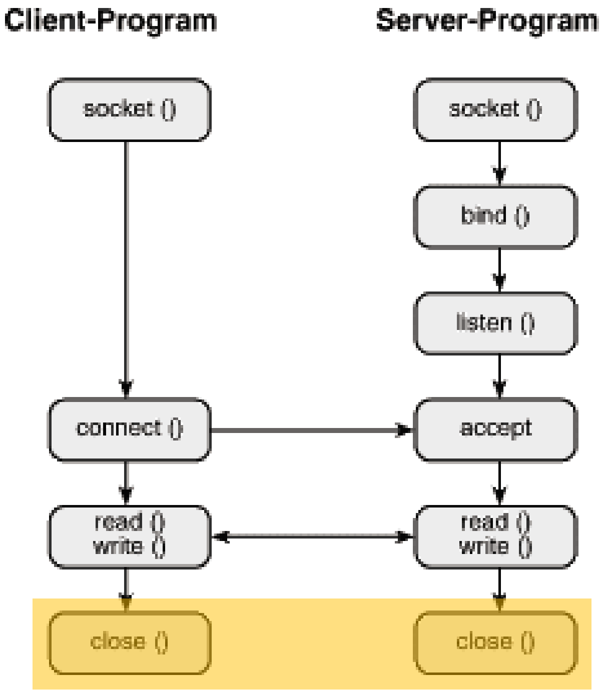
shutdown(2)

- Shut down socket
- `int shutdown(int socket, **int how**)`
  - `SHUT_RD:` shut down receive ops
  - `SHUT_WR:` shut down sending ops
  - `SHUT_RDWR:` shut down obth ops



shutdown(2)

- Shut down socket
- **`int shutdown`**`(int socket, int how)`
  - Success: `0`
  - Error: `-1`



Compressed Communication

Advantage?

Disadvantage?

Compressed Communication

Advantage

- Reduce communication length
- Therefore uses less bandwidth

Disadvantage

- Requires processing at both ends of the communication channel

Compressed Communication

Advantage

- Reduce communication length
- Therefore uses less bandwidth

Disadvantage

- Requires processing at both ends of the communication channel

Is compressed message always smaller than the uncompressed one?

Can compression provide privacy?

Compressed communication

- Pipeline
  - To compress
    - `deflateInit(), deflate(), deflateEnd()`
  - To decompress
    - `inflateInit(), inflate(), inflateEnd()`
- You will need zlib
  - `include <zlib.h>`
  - Compile with `–lz` flag

deflateInit()

- Initialize internal stream state for compression
- `ZEXTERN int ZEXPORT deflateInit OF((z_streamp strm, int level))`

deflateInit()

- Initialize internal stream state for compression
- **`ZEXTERN`** `int` **`ZEXPORT`** `deflateInit` **`OF`**`((z_streamp strm, int level))`
  - zlib macros for cross-platform compatibility
  - You can think of it as
    - `extern int deflateInit(z_streamp strm, int level)`

deflateInit()

- Initialize internal stream state for compression
- ZEXTERN int ZEXPORT deflateInit OF((**z_streamp strm**, int level))
  - Pointer to z_stream

```
typedef struct z_stream_s {
    z_const Bytef *next_in;     /* next input byte */
    uInt     avail_in;  /* number of bytes available at next_in */
    uLong    total_in;  /* total number of input bytes read so far */

    Bytef    *next_out; /* next output byte will go here */
    uInt     avail_out; /* remaining free space at next_out */
    uLong    total_out; /* total number of bytes output so far */

    z_const char *msg;  /* last error message, NULL if no error */
    struct internal_state FAR *state; /* not visible by applications */

    alloc_func zalloc;  /* used to allocate the internal state */
    free_func  zfree;   /* used to free the internal state */
    voidpf     opaque;  /* private data object passed to zalloc and zfree */

    int     data_type;  /* best guess about the data type: binary or text
                           for deflate, or the decoding state for inflate */
    uLong   adler;      /* Adler-32 or CRC-32 value of the uncompressed data */
    uLong   reserved;   /* reserved for future use */
} z_stream;
```

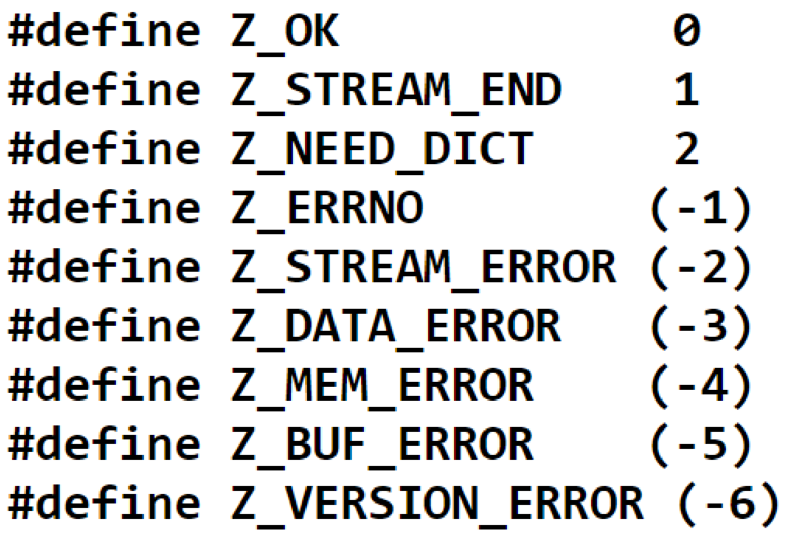
initialize to Z_NULL

deflateInit()

- Initialize internal stream state for compression
- `ZEXTERN int ZEXPORT deflateInit OF((z_streamp strm, ` **`int level`**`))`
  - Between 0 (no compression) and 9 (most compression)
  - Can set to `Z_DEFAULT_COMPRESSION`

deflateInit()

- Initialize internal stream state for compression
- ZEXTERN **int** ZEXPORT **deflateInit** OF((z_streamp strm, int level))



```
#define Z_OK            0
#define Z_STREAM_END    1
#define Z_NEED_DICT     2
#define Z_ERRNO        (-1)
#define Z_STREAM_ERROR (-2)
#define Z_DATA_ERROR   (-3)
#define Z_MEM_ERROR    (-4)
#define Z_BUF_ERROR    (-5)
#define Z_VERSION_ERROR (-6)
```

inflateInit()

- Initialize streams for decompression
- `ZEXTERN int ZEXPORT inflateInit OF((z_streamp strm))`

inflateInit()

- Initialize streams for decompression
- `ZEXTERN int ZEXPORT inflateInit OF((`**`z_streamp strm`**`))`
  - Pointer to `z_stream`
  - Same as `deflateInit()`

inflateInit()

- Initialize streams for decompression
- `ZEXTERN` **`int`** `ZEXPORT` **`inflateInit`** `OF((z_streamp strm))`
  - Same as `deflalteInit()`

deflate()

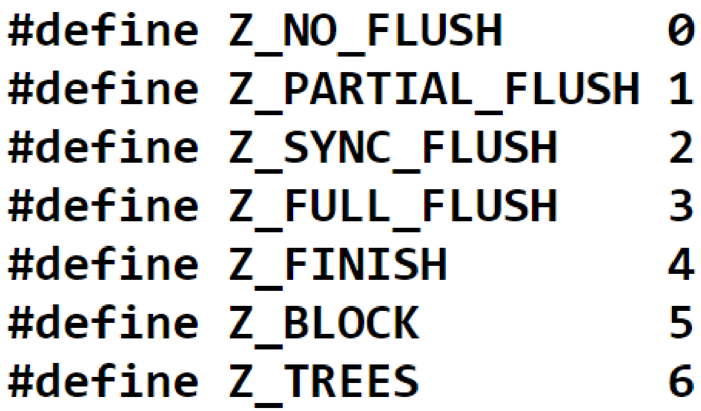
- Compress data until input buffer is empty of output buffer is full
- `ZEXTERN int ZEXPORT deflate OF((z_streamp strm, int flush))`

deflate()

- Compress data until input buffer is empty of output buffer is full
- `ZEXTERN int ZEXPORT deflate OF((`**`z_streamp strm`**`, int flush))`
  - Same as before
  - `strm.next_in:` next input byte
  - `strm.avail_in:` number of bytes available in next_in
  - `strm.total_in:` total number of bytes read so far
  - Similarly,
    - `strm.next_out, strm.avail_out, strm.total_out`

deflate()

- Compress data until input buffer is empty of output buffer is full
- ZEXTERN int ZEXPORT deflate OF((z_streamp strm, **int flush**))
  - How do you want to force flush
  - Forcing flush frequently degrades compression ratio
  - Z_NO_FLUSH gives best compression ratio



```
#define Z_NO_FLUSH      0
#define Z_PARTIAL_FLUSH 1
#define Z_SYNC_FLUSH    2
#define Z_FULL_FLUSH    3
#define Z_FINISH        4
#define Z_BLOCK         5
#define Z_TREES         6
```

deflate()

- Compress data until input buffer is empty of output buffer is full
- `ZEXTERN` **`int`** `ZEXPORT` **`deflate`** `OF((z_streamp strm, int flush))`
  - `Z_OK` on success

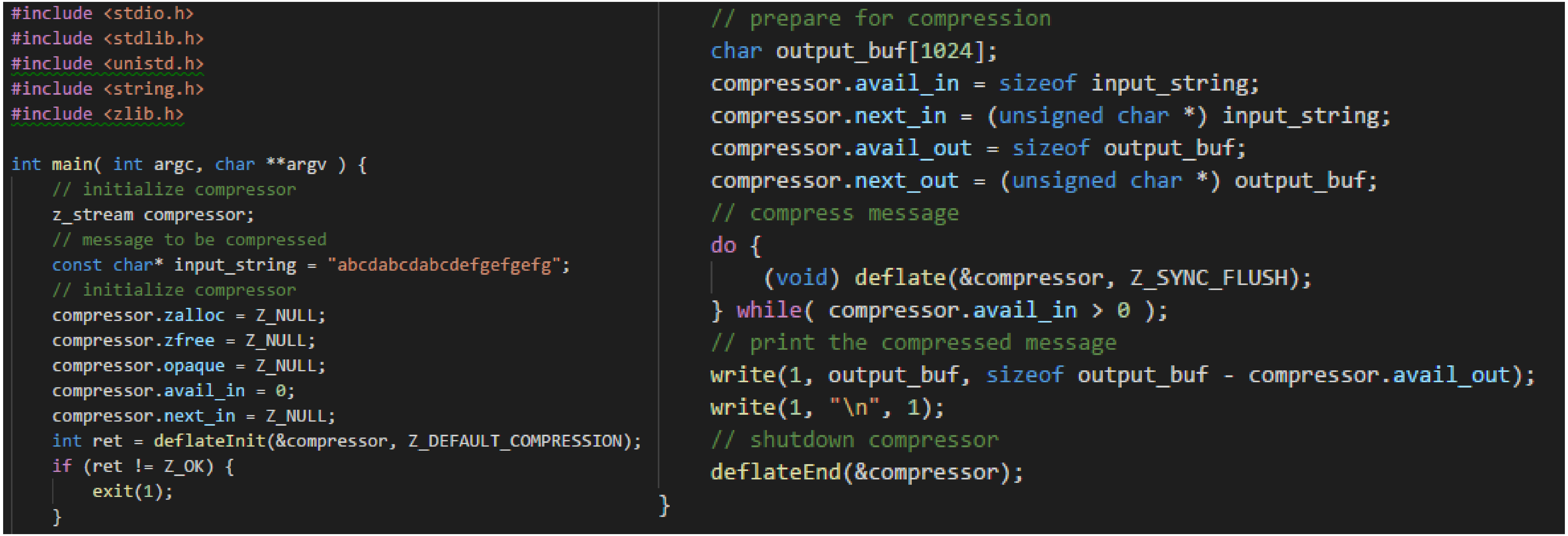
inflate()

- Decompress data until input buffer is empty or output buffer is full
- `ZEXTERN int ZEXPORT inflate OF((z_streamp strm, int flush))`
  - Similar to `deflate()`

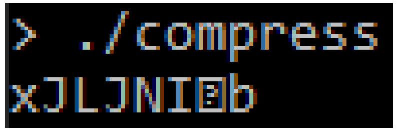
deflateEnd() / inflateEnd()

- `ZEXTERN int ZEXPORT deflateEnd OF((z_streamp strm))`
- `ZEXTERN int ZEXPORT inflateEnd OF((z_streamp strm))`

An example code for compressing



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <zlib.h>

int main( int argc, char **argv ) {
    // initialize compressor
    z_stream compressor;
    // message to be compressed
    const char* input_string = "abcdabcdabcdefgefgefg";
    // initialize compressor
    compressor.zalloc = Z_NULL;
    compressor.zfree = Z_NULL;
    compressor.opaque = Z_NULL;
    compressor.avail_in = 0;
    compressor.next_in = Z_NULL;
    int ret = deflateInit(&compressor, Z_DEFAULT_COMPRESSION);
    if (ret != Z_OK) {
        exit(1);
    }
    // prepare for compression
    char output_buf[1024];
    compressor.avail_in = sizeof input_string;
    compressor.next_in = (unsigned char *) input_string;
    compressor.avail_out = sizeof output_buf;
    compressor.next_out = (unsigned char *) output_buf;
    // compress message
    do {
        (void) deflate(&compressor, Z_SYNC_FLUSH);
    } while( compressor.avail_in > 0 );
    // print the compressed message
    write(1, output_buf, sizeof output_buf - compressor.avail_out);
    write(1, "\n", 1);
    // shutdown compressor
    deflateEnd(&compressor);
}
```



```
> ./compress
xJLJNI□b
```

Workflow

- How your code should work

Workflow: client

- Initialize zlib streams
- Create a socket
  - `socket(2)`
- Identify server
  - `gethostbyname(3)`
- `connect(2)` to server
- Wait for input
  - `poll(2)` on keyboard and socket
  - Compress and decompress if necessary
- `shutdown(2)` socket
- Restore terminal modes

Workflow: server

- Initialize zlib streams
- Create a socket
  - `socket(2)`
- `bind(2)` socket to name
  - Fill server’s sockaddr_in struct INADDR_ANY
- Establish connection with client
  - `listen(2), accept(2)`
- Input/output forwarding
  - `poll(2)` on socket and shell, decompress if necessary

Recommendation

- Test individual modules before putting everything together
    1. Write a server and a client that simply sends “hello” to each other
    2. Write a program and compresses and decompresses messages
        - Test if you compress and decompress, you get the same message back
    3. Add --compress support to your server and client program
    4. Add project 1A to your program

Socket FAQ

- Socket-in-use error
  - `listen(2)` remains active for some time (usually seconds) after the server exits. Wait for a few seconds and try again. Or you can switch to another port.
- Connection refused error
  - Nothing is listening to the port you are trying to connect.
  - On linux servers, monitor listening ports with
    - `netstat -tln`