TCP Socket

Programmazione di Reti

Ing. Chiara Contoli, PhD
chiara.contoli@unibo.it

Corso di Laurea Triennale in
Ingegneria e Scienze Informatiche

Outline

Socket Programming:

- Basic concepts
- TCP socket
- Client/Server interaction

C language Socket Programming

- Simple TCP Client application
- Simple TCP Server Application

TCP: Transmission Control Protocol (RFC 793)

TCP provides a **reliable**, **full-duplex** end-to-end communication between **one** source and **one** destination.

It is **connection-oriented**, **byte-stream oriented** and it implements **flow control** and **congestion control**.

Reliability is achieved through:

- packets sequential numbering;
- acknowledgment;
- and retransmission.

Flow control is about not overloading the destination. While congestion control is about (trying) not to overload the network.

TCP basic concepts

TCP requires **three-way-handshake** mechanism at the beginning in order to establish a connection and to initialize both sender and receiver *state*.

TCP state information concern:

- TCP connection state;
- TCP congestion control algorithm;
- TCP buffers
- ….

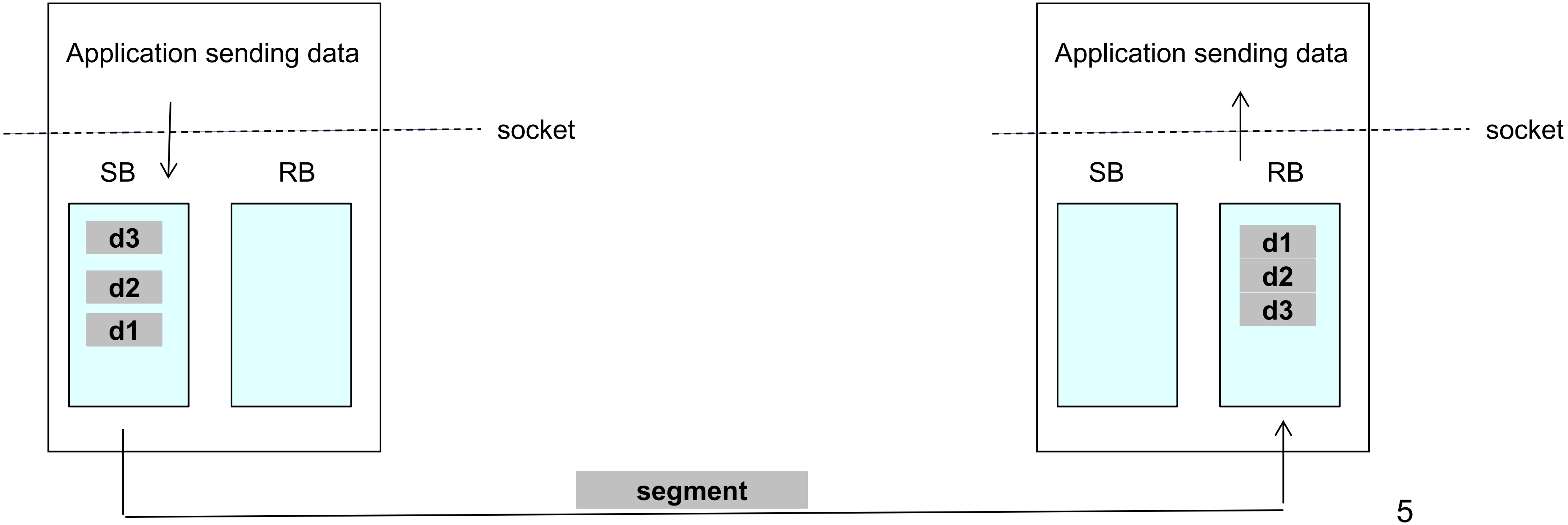
Buffers play an important role in TCP. Applications leverage both

- Sending buffer
- Receiving buffer

TCP buffers

Given that TCP must provide certain guarantees, sending and receiving buffers act as follow:

- All sent data are stored in sending buffer (SB) until a positive ACK is received;
- All received data are stored in receiving buffer (RB) until all pieces have been reordered before passing to application.



TCP Socket Programming

TCP: **client** and **server** have to establish a connection

Handshake occurs between client and server: client has to explicitly *connect* to server *before* sending messages

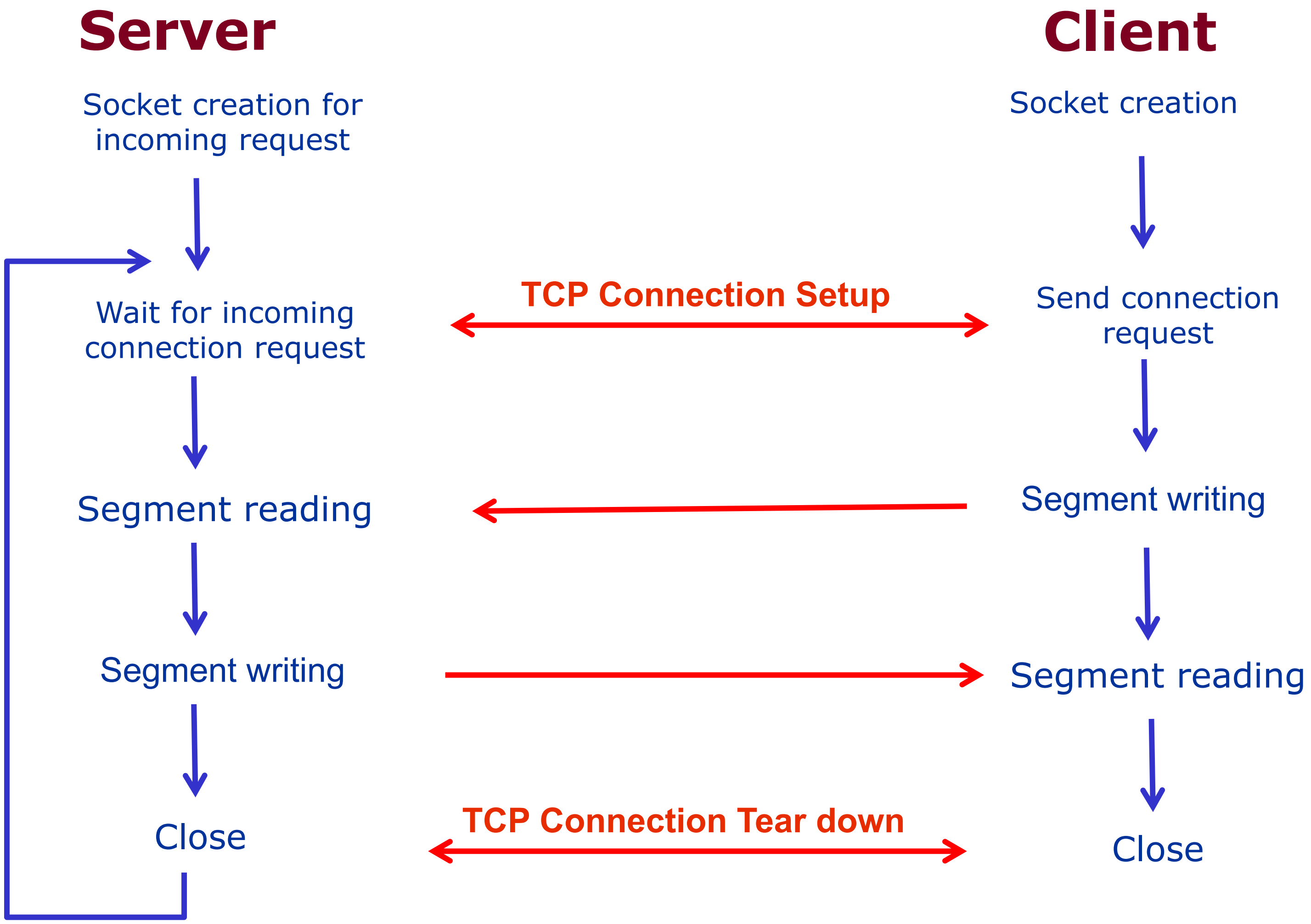
Multiple application level messages will likely be **contained in a single TCP segment**

To send a response to the client, server does not need to extract any information from received segment

TCP: data transmitted are guaranteed to **arrive** at the destination **in order** compared to the sending sequence

Application point of view: TCP provides a **reliable** byte transfer service of stream of bytes (known as segments) between client and server

TCP: client – server interaction



Server

Socket creation for incoming request

Wait for incoming connection request

Segment reading

Segment writing

Close

Client

Socket creation

Send connection request

Segment writing

Segment reading

Close

TCP Connection Setup

TCP Connection Tear down

Socket Libraries

Several libraries are need: <*sys/socket.h*>, <*arpa/inet.h*>, <*netinet/in.h*>

From <*sys/socket.h*>

- int **socket**(int *domain*, int *type*, int *protocol*);
  - Returns (socket) file descriptor on success, -1 on error
  - Parameters: i) *domain*: communication domain, ii) *type*: socket type, iii) *protocol*: protocol to be used

- int **bind**(int *sockfd*, const struct sockaddr **addr*, socklen_t *addrlen*);
  - Returns 0 on success, -1 on error
  - Parameters: i) *sockfd*: socket file descriptor, ii) *addr*: address to be bound to the socket; *addrlen*: size of the ii)

Stream Socket *server side* system calls

From <*sys/socket.h*>

- int **listen**(int *sockfd*, int *backlog*);
  - Returns 0 on success, -1 on error
  - Parameters: i) *sockfd*: socket file descriptor, ii) *backlog*: maximum number of connections that can be queued

- int **accept**(int *sockfd*, struct sockaddr **addr*, socklen_t **addrlen*);
  - Returns non-negative integer on success (a new socket), -1 on error
  - Parameters: i) *sockfd*: socket file descriptor, ii) *addr*: filled with *peer socket address*, iii) *addrlen*: size of peer address

Stream Socket *server side: note*

How many types of sockets exist in such Connection Oriented context?

- *Association* server socket (connection):
  - Associated to a port
  - Supported operations: *bind*, *listen*, *accept*
  - (also known as “welcome socket”)
- *Communication* server socket:
  - Associated to an already established connection
  - Supported operations: *read*, *write*
- *Association* client socket (connection) and communication:
  - Associated to a port (in case port is disconnected) or to a connection (in case port is connected)
  - Supported operations: *bind* (optional), *connect* (in case port is not already connected), *read* and *write* (in case port is connected)

Stream Socket *client side* system call

From <*sys/socket.h*>

- int **connect**(int *sockfd*, const struct sockaddr **addr*, socklen_t *addrlen*);
  - Returns 0 on success, -1 on error
  - Parameters: i) *sockfd*: socket file descriptor, ii) *addr*: peer socket address you want to connect to, iii) *addrlen*: size of *addr*

This system call can be also invoked by connectionless socket. Be aware that, if socket type is:

- ○ SOCK_STREAM: *connect()* results in an attempt to establish a connection (i.e., three-way-handshake mechanism)
- ○ SOCK_DGRAM: *connect()* results in sending/receiving packets *only* to/from specified *addr* (note: it **does not trigger** the three-way-handshake mechanism)

Stream Socket I/O

From <*sys/socket.h*>

- ssize_t **recv**(int *sockfd*, void **buffer*, size_t *length*, int *flags*);
  - Returns number of bytes received, 0 on stream socket peer “polite” shutdown, -1 on error
  - Parameters: i) *sockfd*: socket file descriptor, ii) *buffer*: store received data, iii) *length*: size of the buffer vi) *flags*: bitwise operation

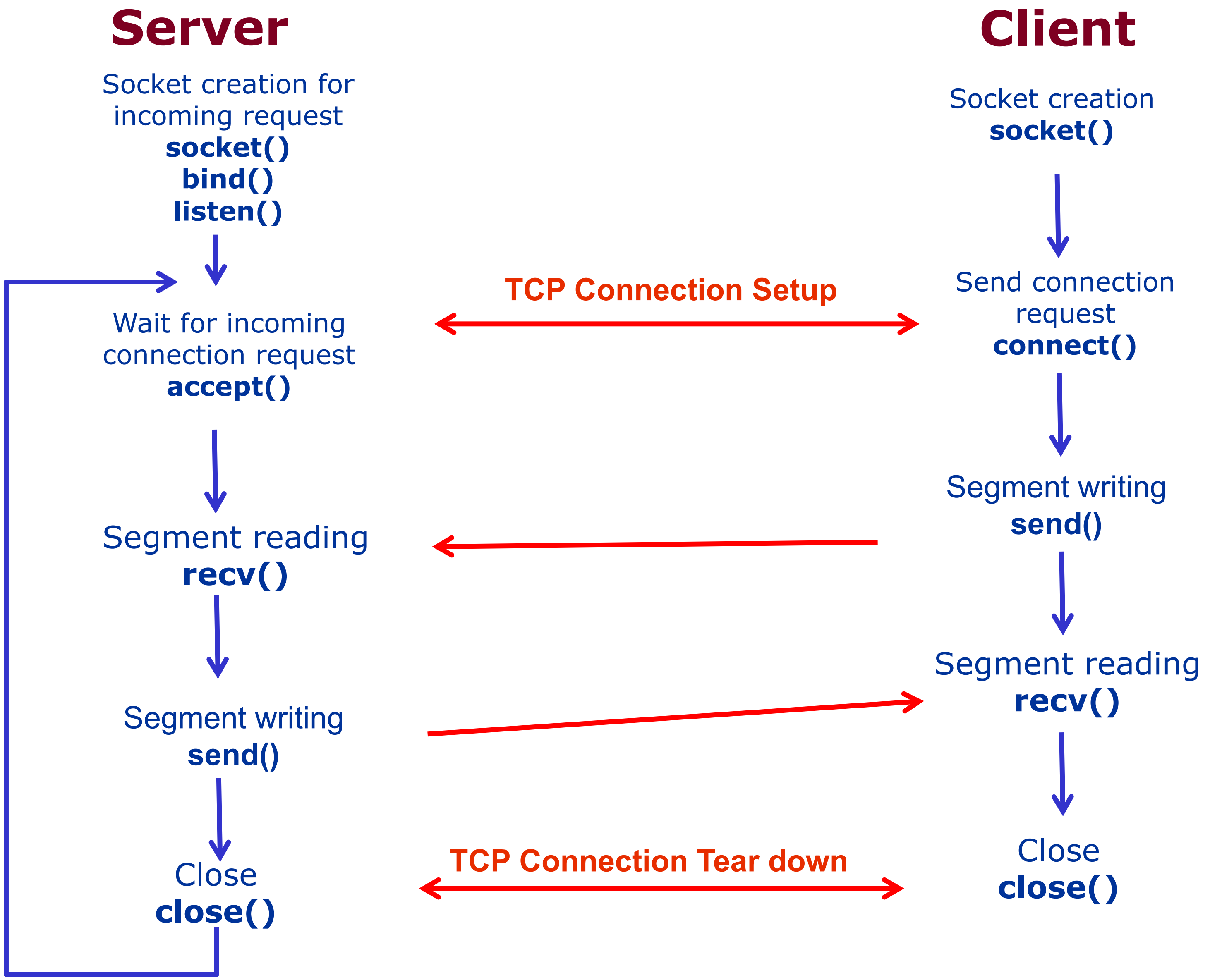
- ssize_t **send**(int *sockfd*, const void **buffer*, size_t *length*, int *flags*);
  - Returns number of bytes sent, -1 on error
  - Parameters: i) *sockfd*: socket file descriptor, ii) *buffer*: store message to be sent, iii) *length*: length of the message

Stream Socket *client/server side* system call

From <*unistd.h*>

- int **close**(int *sockfd*);
  - Returns 0 on success, -1 on error
  - Parameter: *sockfd*: socket file descriptor

TCP: client – server interaction



Server

Socket creation for incoming request
socket()
bind()
listen()

Wait for incoming connection request
accept()

Segment reading
recv()

Segment writing
send()

Close
close()

TCP Connection Setup

TCP Connection Tear down

Client

Socket creation
socket()

Send connection request
connect()

Segment writing
send()

Segment reading
recv()

Close
close()

Example: a simple client / server application

Client:

- User leverage keyboard (standard input) to write a string
- Client send this string to server

Server:

- Server receive the string sent by the client
- Convert the string to upper case letter
- Send modified string to the client

Client:

- Receive modified string line
- Print string on the screen (standard output)

Example: TCP Server

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include "myfunction.h”

#define MAX_BUF_SIZE 1024
#define SERVER_PORT 9876 // Server port
#define BACK_LOG 2 // Maximum queued requests

int main(int argc, char *argv[]){
  struct sockaddr_in server_addr; // struct containing server address information
  struct sockaddr_in client_addr; // struct containing client address information
  int sfd; // Server socket filed descriptor
  int newsfd; // Client communication socket - Accept result
  int br; // Bind result
  int lr; // Listen result
```

Example: TCP Server

```
int i;
int stop = 0;
ssize_t byteRecv; // Number of bytes received
ssize_t byteSent; // Number of bytes to be sent

socklen_t cli_size;
char receivedData [MAX_BUF_SIZE]; // Data to be received
char sendData [MAX_BUF_SIZE]; // Data to be sent

sfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

if (sfd < 0){
  perror("socket"); // Print error message
  exit(EXIT_FAILURE);
}

// Initialize server address information
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SERVER_PORT); // Convert to network byte order
server_addr.sin_addr.s_addr = INADDR_ANY; // Bind to any address
```

Example: TCP Server

```
br = bind(sfd, (struct sockaddr *) &server_addr, sizeof(server_addr));

if (br < 0){
  perror(”bind"); // Print error message
  exit(EXIT_FAILURE);
}
cli_size = sizeof(client_addr);

// Listen for incoming requests
lr = listen(sfd, BACK_LOG);
if (lr < 0){
  perror(”listen"); // Print error message
  exit(EXIT_FAILURE);
}

for(;;){
  // Wait for incoming requests
  newsfd = accept(sfd, (struct sockaddr *) &client_addr, &cli_size);
  if (newsfd < 0){
    perror(”accept"); // Print error message
    exit(EXIT_FAILURE);
  }
```

Example: TCP Server

```
while(1){
  byteRecv = recv(newsfd, receivedData, sizeof(receivedData), 0);
  if (byteRecv < 0){
    perror("recv");
    exit(EXIT_FAILURE);
  }

  printf("Received data: ");
  printData(receivedData, byteRecv);
  if(strncmp(receivedData, "exit", byteRecv) == 0){
    printf("Command to stop server received\n");
    close(newsfd);
    break;
  }
  convertToUpperCase(receivedData, byteRecv);
  printf("Response to be sent back to client: ");
  printData(receivedData, byteRecv);

  byteSent = send(newsfd, receivedData, byteRecv, 0);
```

Example: TCP Server

```
      if(byteSent != byteRecv){
        perror("send");
        exit(EXIT_FAILURE);
      }
    }
  } // End of for(;;)

  close(sfd);
  return 0;
}
```

Example: TCP Client

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include "myfunction.h”

#define MAX_BUF_SIZE 1024
#define SERVER_PORT 9876 // Server port

int main(int argc, char *argv[]){
  struct sockaddr_in server_addr; // struct containing server address information
  struct sockaddr_in client_addr; // struct containing client address information
  int sfd; // Server socket filed descriptor
  int br; // Bind result
```

Example: TCP Client

```
int cr; // Connect result
int stop = 0;

ssize_t byteRecv; // Number of bytes received
ssize_t byteSent; // Number of bytes to be sent
size_t msgLen;
socklen_t serv_size;
char receivedData [MAX_BUF_SIZE]; // Data to be received
char sendData [MAX_BUF_SIZE]; // Data to be sent

sfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

if (sfd < 0){
  perror("socket"); // Print error message
  exit(EXIT_FAILURE);
}

server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SERVER_PORT);
server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

Example: TCP Client

```
 serv_size = sizeof(server_addr);

 cr = connect(sfd, (struct sockaddr *) &server_addr, sizeof(server_addr));

 if (cr< 0){
   perror(”connect"); // Print error message
   exit(EXIT_FAILURE);
 }

 while(!stop){
   printf("Insert message:\n");
   scanf("%s", sendData);
   printf("String going to be sent to server: %s\n", sendData);

   if(strcmp(sendData, "exit") == 0){
     stop = 1;
   }

   msgLen = countStrLen(sendData);
   byteSent = sendto(sfd, sendData, msgLen, 0, (struct sockaddr *)
&server_addr, sizeof(server_addr));
   printf("Bytes sent to server: %zd\n", byteSent);
```

Example: TCP Client

```
    if(!stop){
      byteRecv = recv(sfd, receivedData, MAX_BUF_SIZE, 0);
      printf("Received from server: ");
      printData(receivedData, byteRecv);
    }
  } // End of while
  close(sfd);
  return 0;
}
```

“myfunction.h”

```
#include <ctype.h>

size_t countStrLen(char *str){
  size_t c = 0;

  while(*str != ‘\0’){
    c += 1;
    str++;
  }
  return c;
}

void printData(char *str, size_t numBytes){
  for(int i = 0; i < numBytes; i++){
    printf("%c", str[i]);
  }
  printf("\n");
}
```

“myfunction.h”

```
void convertToUpperCase(char *str, size_t numBytes){
  for(int i = 0; i < numBytes; i++){
    str[i] = toupper(str[i]);
  }
}
```

Server operative mode: Sequential VS Concurrent

Programmazione di Reti

Ing. Chiara Contoli, PhD
chiara.contoli@unibo.it

Corso di Laurea Triennale in
Ingegneria e Scienze Informatiche

Sequential mode VS Concurrent mode

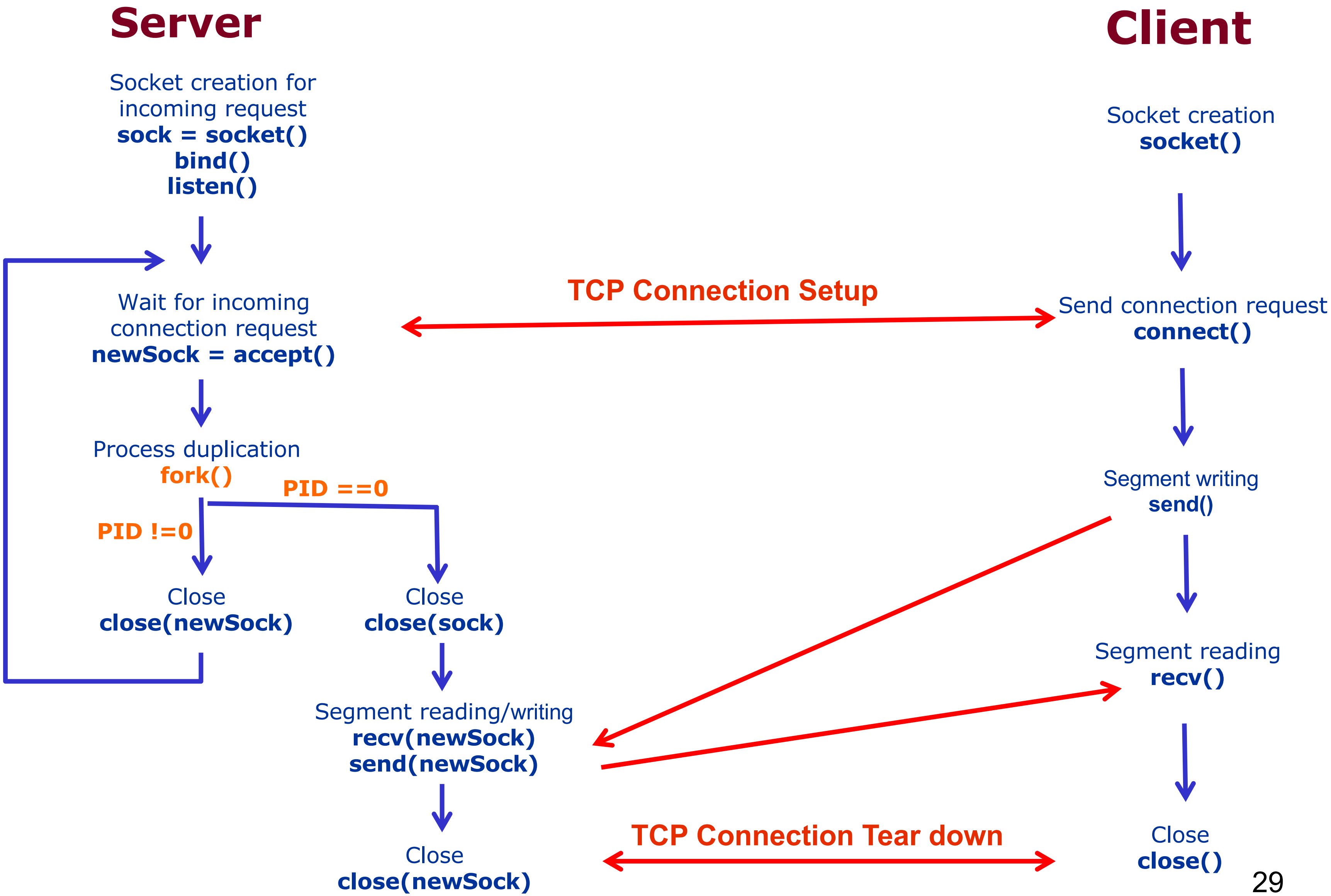
Sequential mode:

- When server S1 undertakes a dialog with a client C1, it fulfil such request (by accomplishing a task, typically) and terminates the connection with it before explicitly accepting (fulfill) a possibly new connection request with another client C2.

Concurrent mode:

- When server S1 undertakes a dialog with a client C1, S1 is duplicated:
  - One of the copy will continue to serve C1 till the end of the dialog, it will terminate the connection and it will terminate itself (i.e., this server copy dies).
  - The other copy will go back to wait for new incoming connection requests from other clients.

TCP: client – server concurrent interaction



Server

Socket creation for incoming request
sock = socket()
bind()
listen()

Wait for incoming connection request
newSock = accept()

Process duplication
fork()

PID !=0

Close
close(newSock)

PID ==0

Close
close(sock)

Segment reading/writing
recv(newSock)
send(newSock)

Close
close(newSock)

Client

Socket creation
socket()

Send connection request
connect()

Segment writing
send()

Segment reading
recv()

Close
close()

TCP Connection Setup

TCP Connection Tear down

TCP concurrent server, pseudo code example

```
…
int sock, newSock;
sock = socket(…);
bind(sock, …);
listen(sock, 5);
for(;;){
  newSock = accept(sock, …);
  if (fork() == 0) {  // clone process
    close(sock);
    doTask(sock);
    exit(0);
  } else {
    close(newSock);  // father process
  }
}
```

What if an interaction with multiple sockets is needed?

Examples: a remote terminal application, at client side, consists of 2 input source

  - The local terminal
  - Connection with the remote server

Possible solutions:

- Polling
- Asynchronous interaction
- Any other solutions?

A system call, which works not only on sockets, exists:

From <*sys/types.h*> <*sys/time.h*>

- int **select**(int *maxfdpl*, fd_set **readfds*, fd_set **wirtefds*, fd_set **exceptfds*, struct timeval **timeout);*
  - Returns the number of sockets ready for the required I/O operation (it returns how many and which file are ready to be accessed without being blocked), -1 on error, 0 if no socket is ready for one of the required I/O operations

See http://man7.org/linux/man-pages/man2/select.2.html

I/O multiplexing: *select* in detail

The data type *fd_set* represent a set of file/socket descriptor. It is concretely implemented as an array of bits, each (positionally) associated to a file descriptor

Parameters:

- i) *maxfdpl*: highest numbered files descriptor in any of the three sets, plus 1 (e.g.: if descriptors 1, 4 and 7 has been set, maxfdpl is 8=7+1), ii) *readfds*: files ready to be read, iii) *writefds*: file ready to be written (in the socket buffer), iv) *exceptfds*: files experienced an exception, v) *timeval*: interval time, in seconds, spent (blocked!) waiting for a file descriptor to become ready

In case you are not interested in one of the I/O operation, NULL can be set in the corresponding fd_set.

I/O multiplexing: *select* in detail

How can you understand which sockets are ready for the required I/O operation? The application needs to scan all the sets of socket descriptors and tests each bit with *FD_ISSET()* operation.

Operation on fd_set:

- void **FD_ZERO**(fd_set **fdset);*
  - *Empty fd_set*
- void **FD_SET**(int fd, fd_set **fdset);*
  - Insert the numbered file descriptor *fd* into *fdset* set
- void **FD_CLR**(int fd, fd_set **fdset);*
  - Remove the numbered file descriptor *fd* from *fdset* set
- int **FD_ISSET**(int fd, fd_set **fdset);*
  - Check if the numbered file descriptor *fd* is present (returned value != 0) or not (returned value == 0) in the *fdset*

Few notes about *timeval*

The *select()* function blocks until:

- A file descriptor gets ready;
- The call is interrupted by a signal handler;
- The timeout specified by *timeval* expires

Also, if

- *timeval* != NULL and all 3 sets are empty, *select()* is non-blocking, returns immediately after socket status check
- *timeval* == NULL, *select()* is blocked indefinitely, returns only after at least one socket is ready for required I/O operation
- *timeval* != NULL and at least one of the set is not empty, *select()* blocks until one of the socket is ready for required I/O operation, but *only* for the quantity specified by *timeval*

select() usage pseudo code example

Considering only *fd* in reading set

```
int sock, newSock, readyFdNum, maxFd, temp;
fd_set readSet;
struct timeval tWait;
…
sock = socket(…);
bind(sock, …);
listen(sock, 5);
for(;;){
  FD_ZERO(&readSet);
  FD_SET(sock, &readSet);
  tWait.tv_sec = 5;
  tWait.tv_usec = 0;

  if ((temp = select(sock+1, &readSet, NULL, NULL, &tWait)) < 0) {
    printf(”Select error\n");
  }
```

select() usage pseudo code example

```
  if (temp == 0) {
    printf(”Timeout expired, no pending connection on socket\n");
  } else {
    if(!FD_ISSET(sock, &readSet)){
      printf(”Error\n");
    }
    printf(”At least one connection pending on socket\n");
    newSock = accept(sock, …);
    doTask(newSock );
    close(newSock);
  }
} // Close for
```