

Sistemi Operativi e Reti di Calcolatori (SORECa)

Corso di Laurea in *Ingegneria Informatica e Automatica (BIAR)*

Terzo Anno | Primo Semestre

A.A. 2024/2025

Esercitazione [09] Semplice HTTP server “from scratch”

Riccardo Lazzeretti lazzeretti@diag.uniroma1.it

Paolo Ottolino paolo.ottolino@uniroma1.it

Alessio Izzillo izzillo@diag.uniroma1.it

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI



SAPIENZA
UNIVERSITÀ DI ROMA

Sommario

- Soluzioni precedente esercitazione
- Simple HTTP Server:
 - Caratteristiche
 - Recap su Socket TCP/IP: `socket()`, `bind()`, `listen()`, `accept()`
 - RFC HTTP/1: RFC 1945
- Lab09, es. 1: nano HTTP server
 - Ascolto: Elaborare Thread-based (Pooling), GET valida, nome file e sua codifica
 - Risposta: tipo MIME, Ricerca file e sua Esistenza, Messaggio HTTP

Simple HTTP server

Caratteristiche

Recap su Socket TCP/IP: `socket()`, `bind()`, `listen()`, `accept()`

RFC HTTP/1: RFC 1945

Simple HTTP Server

Caratteristiche

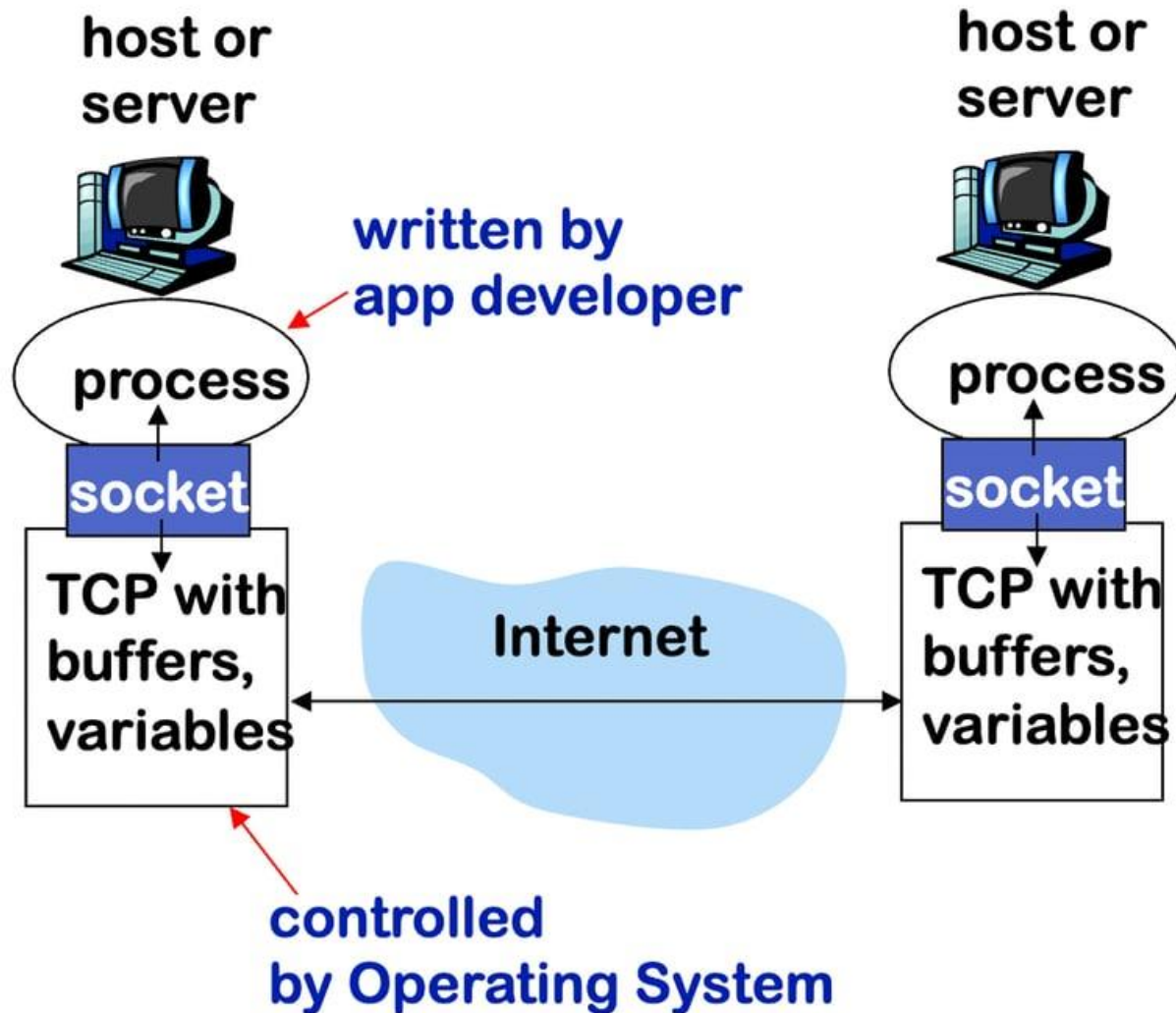
Il server HTTP deve effettuare le seguenti operazioni:

1. ascoltare le connessioni in arrivo su una **porta TCP 8080**. → `socket()`, `bind()`, `listen()`
2. elaborare le richieste dei client **in thread separati** (più connessioni simultanee. → `pthread_create()`)
3. accettare **solo richieste** dal client valide di tipo **GET**. → gestione GET non valida
4. estrarre il **nome file** richiesto (eliminare «.», evitare directory traversal)
5. decodificare il **nome file codificato** in URL (ATTENZIONE: i caratteri speciali sono codificati; es. « » diventa %20)
6. ricercare il file **senza distinzione** tra **maiuscole** e **minuscole** (ATTENZIONE: in Unix «A» e «a» sono caratteri distinti).
7. verificare se il **file** richiesto **esiste** nella directory corrente. → gestione «**404 Not Found**».
8. determinare l'**estensione** del **file** per identificare il **tipo MIME** appropriato per la risposta
9. creare una **risposta HTTP** con le intestazioni appropriate, incluso il tipo MIME determinato, e invia il contenuto del file al client.

Il server è un demone: continua ad accettare ed elaborare le connessioni in arrivo finché non viene terminato manualmente (Ctrl-C, kill -9 etc.).

Simple HTTP Server

1. Ascoltare le connessioni in arrivo – TCP Sockets in Unix 1/2

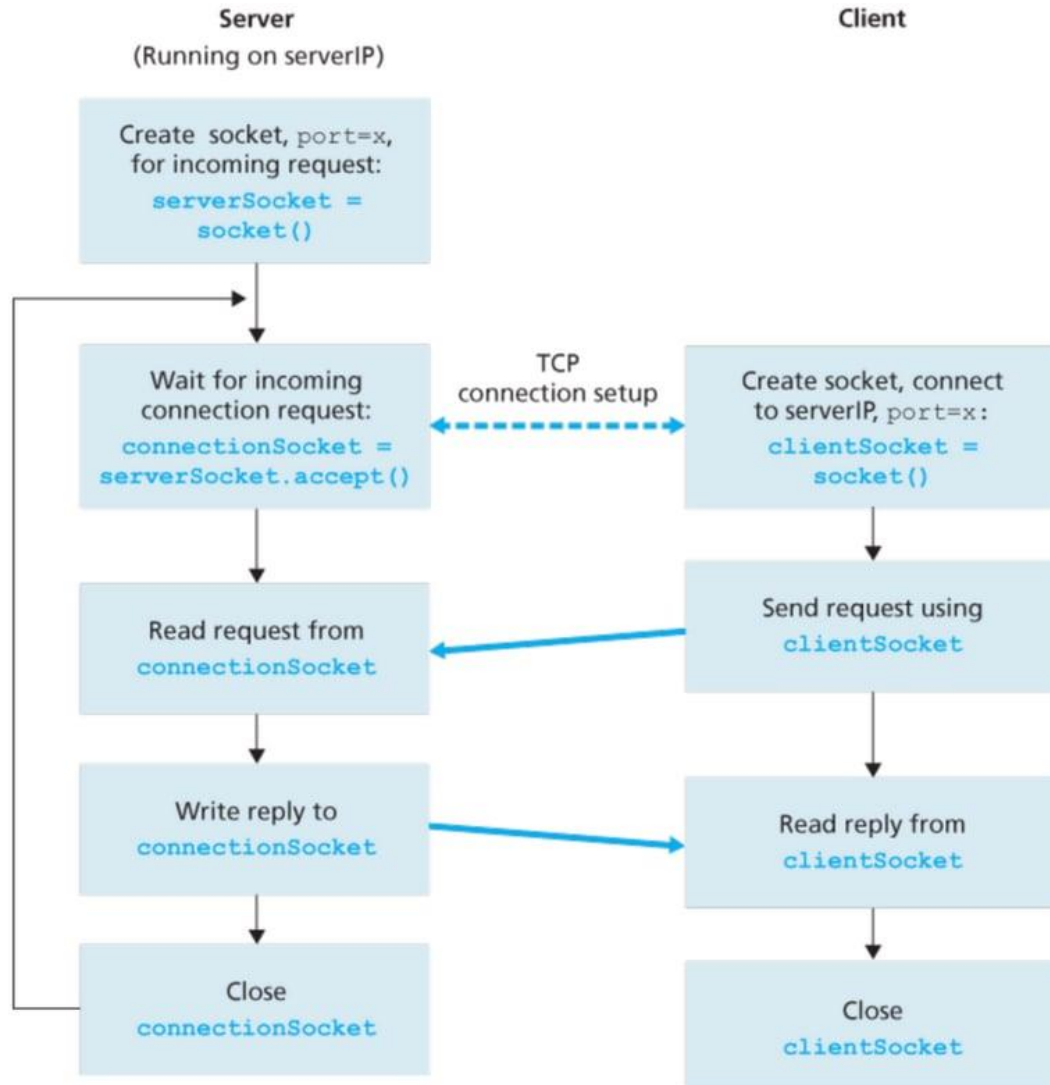


Se un processo in esecuzione su un host è come una casa, un socket è come una porta che lascia entrare e uscire la posta.

Quando una persona in casa riceve o consegna una lettera, può essere agnostica su come la posta viene consegnata nel mondo esterno.

Simple HTTP Server

1. Ascoltare le connessioni in arrivo – TCP Sockets in Unix 2/2



Utilizzando l'interfaccia di rete socket, è possibile redigere il codice del server web traendo vantaggio dal set di funzioni dal pacchetto standard C `<sys/socket.h>` che consentono al server di «dialogare con» i client tramite Internet. Anche i client utilizzano socket per parlare con il server web. Il socket è il meccanismo consensuale con cui comunicare tramite Internet.

Simple HTTP Server

1. Ascoltare le connessioni in arrivo – UDP, TCP Sockets in C

1. creare una socket per esporre le funzionalità del server: `socket()`:
 1. configurare la famiglia di socket richiesta (`sin_family = AF_INET`)
 2. configurare l'indirizzo IPv4 (`sin_addr.sin_addr = INADDR_ANY`)
 3. configurare il numero di porta (`sin_port = htons(PORT)`).

Nel server HTTP, la configurazione del socket segue l'URL `http://localhost:8080` usando il numero di porta 8080.

2. associare il socket alla porta (`bind(server fd, (struct sockaddr *)&server_addr, sizeof(server_addr))`)
 - una volta associato, **altre applicazioni o processi** utente **non** potranno **acquisire quel socket** fino al rilascio da parte del processo server
 - Ovvero, dal momento del **`bind()`**, il processo server avrà il **pieno controllo** su quella **porta**.
 - Finché il **server** sarà **in esecuzione**, la **porta** sarà **riservata** a quel server.
 - Questo è il **meccanismo** con cui il **kernel** (che gestisce tutte le comunicazioni TCP/IP) **associa univocamente** tutte le **connessioni** su una determinata **porta** ad un determinato **processo**.
 - Anche **qualora non** sia in **ascolto** su **tutte le interfacce** di rete (cioè su tutti gli IP) dell'host, la **porta** è **riservata** ad esso **dappertutto**, anche dove non in ascolto.
 - Dopo il **`bind()`** avvenuto con **successo**, siamo sicuri che **tutte** le **comunicazioni** su quella **porta** siano veicolate verso il **server**, potendo ascoltare da quella porta le richieste utentem senza perderne alcuna.

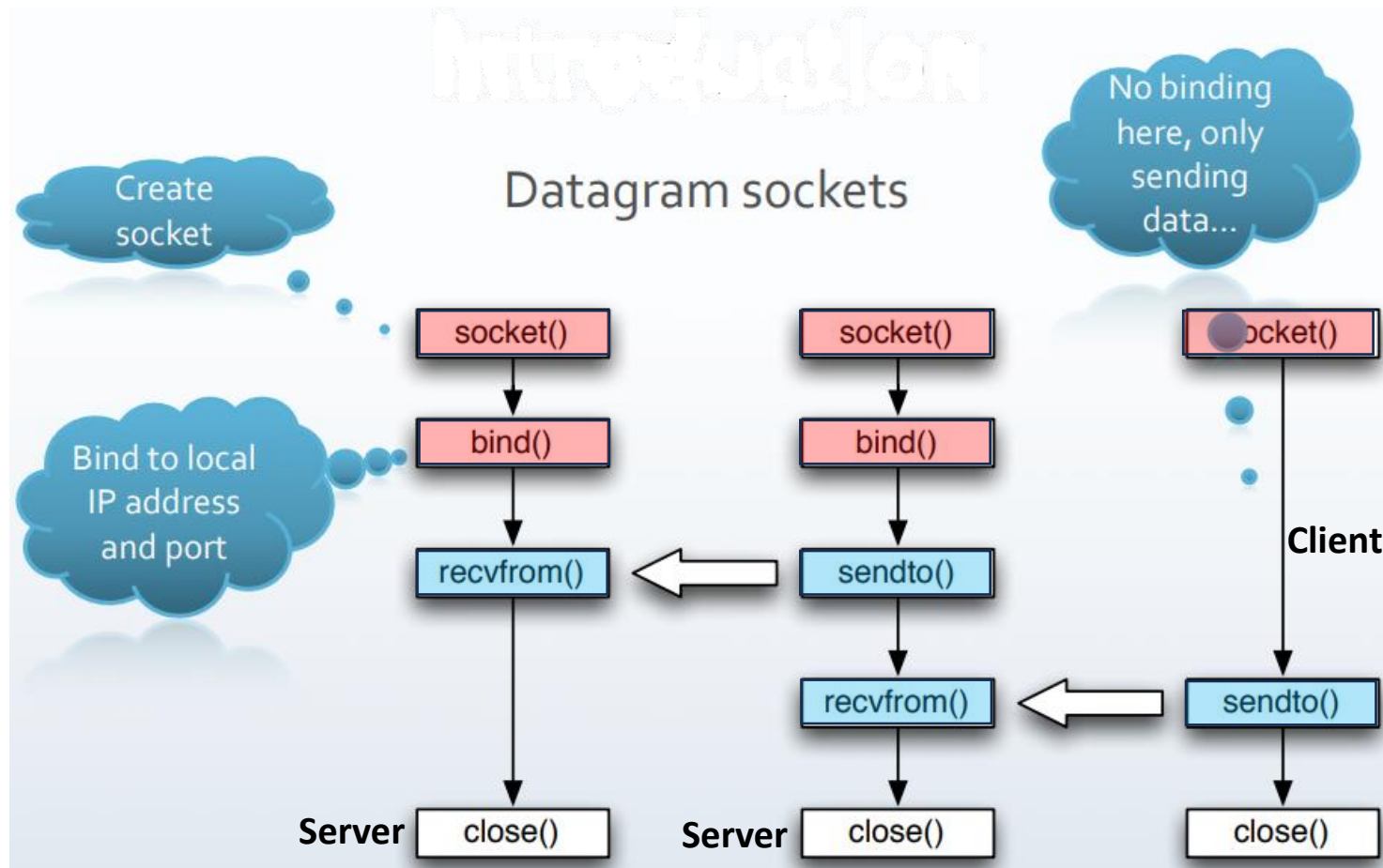
Simple HTTP Server

1. Ascoltare le connessioni in arrivo – UDP, TCP Sockets in C

3. Mettersi in ascolto sul socket (**solo TCP**): `listen()`
4. Accettare connessioni sulla socket (**solo TCP**): `accept()`
5. Ricevere ed inviare i dati sulla socket, per ogni connessione: `send()` `recv()`:
 - **UDP**: `recvfrom()`, `sendto()`
 - **TCP**: `recv()`, `send()`
6. Chiudere il socket del client (**solo TCP**): `close(client_sock)`
7. Chiudere il socket del client: `close(server_fd)`

Simple HTTP Server

How Socket Works – Datagram 1/2



- No connection and unreliable the same socket can be used to send/receive
- Datagrams to/from multiple processes
- uniquely identified by the IP addresses and port numbers (remote and local) for:
 - client (`socket()`)
 - server (`socket()`, `bind()`)
- No syntax distinction between server and client, just:
 - `sendto()`
 - `recvfrom()`

Simple HTTP Server

Ascoltare le connessioni in arrivo - UDP Sockets in C

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main(void){
    int socket_desc;
    struct sockaddr_in server_addr, client_addr;
    char server_message[2000], client_message[2000];
    int client_struct_length = sizeof(client_addr);

    // Clean buffers:
    memset(server_message, \0 , sizeof(server_message));
    memset(client_message, \0 , sizeof(client_message));

    // Create UDP socket:
    socket_desc = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if(socket_desc < 0){/* AF_INET: use IPv4 (no IPv6); SOCK_DGRAM: UDP */
        printf("Error while creating socket\n");
        return -1;    }
    printf("Socket created successfully\n");

    // Set port and IP:
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(2000);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    // Bind to the set port and IP:
    if(bind(socket_desc, (struct sockaddr*)&server_addr, sizeof(server_addr))
    < 0){
        printf("Couldn't bind to the port\n");
        return -1;    }
    printf("Done with binding\n");

    printf("Listening for incoming messages...\n\n");
```

```
// Receive clients message:
if (recvfrom(socket_desc, client_message, sizeof(client_message), 0,
    (struct sockaddr*)&client_addr, &client_struct_length) < 0){
    printf("Couldn't receive\n");
    return -1;    }
printf("Received message from IP: %s and port: %i\n",
    inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));

printf("Msg from client: %s\n", client_message);

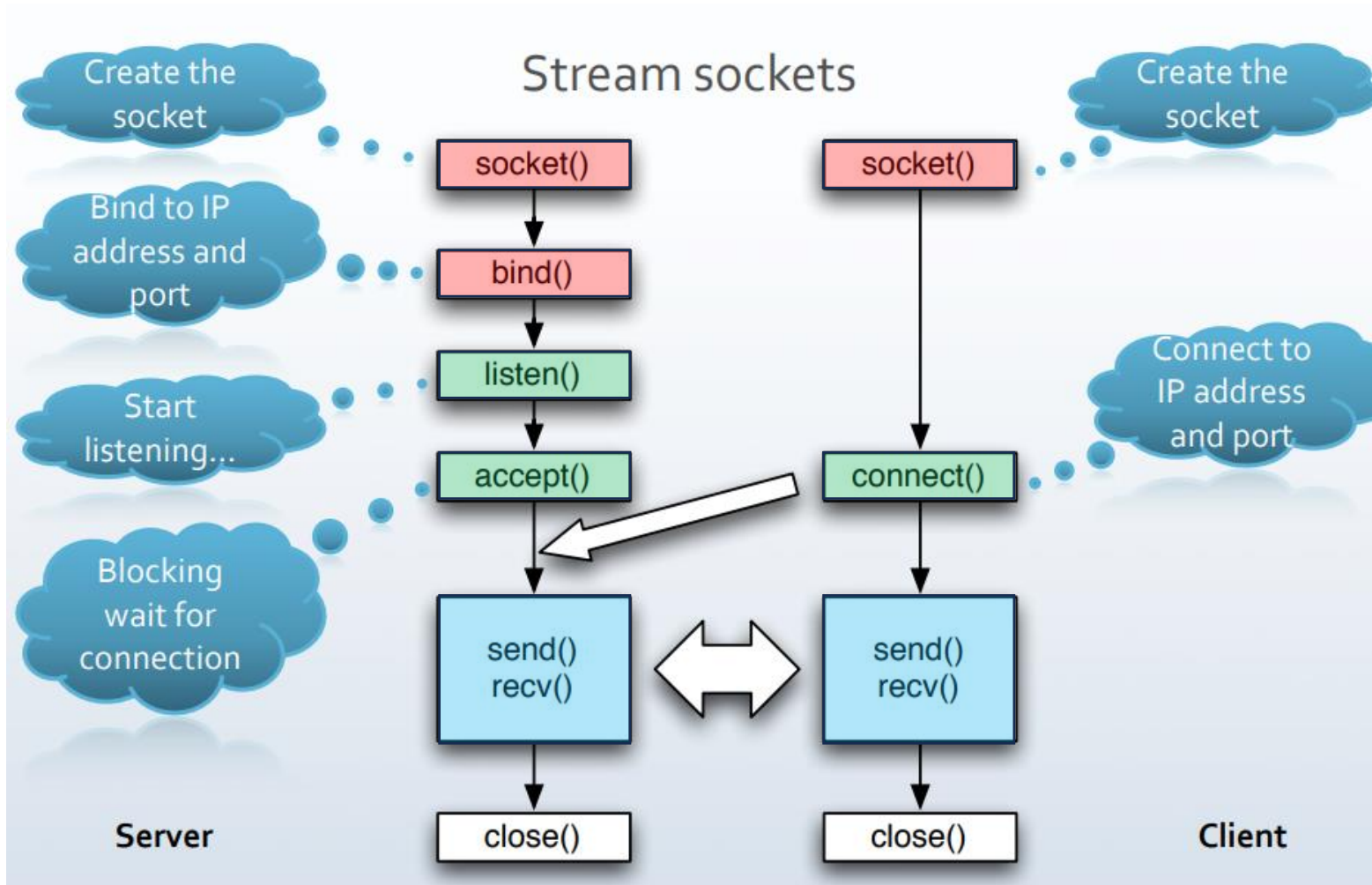
// Respond to client:
strcpy(server_message, client_message);
if (sendto(socket_desc, server_message, strlen(server_message), 0,
    (struct sockaddr*)&client_addr, client_struct_length) < 0){
    printf("Can't send\n");
    return -1;    }

// Close the socket: un solo messaggio e poi muore
close(socket_desc);
return 0;
}
```

Ma il protocollo HTTP prevede una connessione TCP, quindi un socket TCP, non UDP...

Simple HTTP Server

How Socket Works – Stream 2/2



- stream socket = virtual circuit between two processes
- the connection is:
 - Sequenced
 - Reliable
 - Bi-directional (`send()`, `recv()`)
- uniquely identified by the IP addresses and port numbers (remote and local) for:
 - client (`socket()`)
 - server (`socket()`, `bind()`)
- setup by:
 - a server that awaits (`listen()`) for connections (`accept()`)
 - a client that connects to a server (`connect()`)

Simple HTTP Server

Ascoltare le connessioni in arrivo - TCP Sockets in C

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main(void){
    int socket_desc, client_sock, client_size;
    struct sockaddr_in server_addr, client_addr;
    char server_message[2000], client_message[2000];

    // Clean buffers:
    memset(server_message, \0 , sizeof(server_message));
    memset(client_message, \0 , sizeof(client_message));
    // Create socket:
    socket_desc = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if(socket_desc < 0){/* AF_INET: use IPv4 (no IPv6);
    SOCK_STREAM: TCP (no UDP) */
        printf("Error while creating socket\n");
        return -1;    }
    printf("Socket created successfully\n");

    // Set port and IP:
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(2000);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    // Bind to the set port and IP:
    if(bind(socket_desc, (struct sockaddr*)&server_addr,
    sizeof(server_addr))<0){
        printf("Couldn't bind to the port\n");
        return -1;    }
    printf("Done with binding\n");

    // Listen for clients:
    if(listen(socket_desc, 1) < 0){
        printf("Error while listening\n");
        return -1;    }
    printf("\nListening for incoming connections.....\n");

    // Accept an incoming connection:
    client_size = sizeof(client_addr);
    client_sock = accept(socket_desc, (struct sockaddr*)&client_addr,
    &client_size);
    if (client_sock < 0){
        printf("Can't accept\n");
        return -1;    }
    printf("Client connected at IP: %s and port: %i\n",
    inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));

    // Receive client's message:
    if (recv(client_sock, client_message, sizeof(client_message), 0) < 0){
        printf("Couldn't receive\n");
        return -1;    }
    printf("Msg from client: %s\n", client_message);

    // Respond to client:
    strcpy(server_message, "This is the server's message.");
    if (send(client_sock, server_message, strlen(server_message), 0) <
    0){
        printf("Can't send\n");
        return -1;    }

    // Closing the socket:
    close(client_sock);
    close(socket_desc);

    return 0;
}
```



Simple HTTP Server

HTTP/1 – [RFC 1945](#) (1996): Metodi del Protocollo

Il client deve inserire nel messaggio di richiesta quale tipo di servizio vuole richiedere al server.

Il simple HTTP server accetta solo il metodo GET

SAFE METHODS NO ACTION ON SERVER	{	GET	HTTP/1.1 MUST IMPLEMENT THIS METHOD	
		HEAD	INSPECT RESOURCE HEADERS	
MESSAGE WITH BODY SEND DATA TO SERVER	{	PUT	DEPOSIT DATA ON SERVER – INVERSE OF GET	
		POST	SEND INPUT DATA FOR PROCESSING	
		PATCH	PARTIALLY MODIFY A RESOURCE	
		TRACE	ECHO BACK RECEIVED MESSAGE	
		OPTIONS	SERVER CAPABILITIES	
		DELETE	DELETE A RESOURCE – NOT GUARANTEED	

La versione 1.0 del protocollo HTTP prevede solo i metodi: GET, HEAD, POST.

Prevede come addizionali: PUT, DELETE, LINK, UNLINK

Gli altri sono stati aggiunti con HTTP/1.1 ([RFC 2616](#)), nel 1999

Simple HTTP Server

HTTP/1 – [RFC 1945](#) (1996): Codici di Stato

HTTP Status Codes

Level 200

200: OK
201: Created
202: Accepted
203: Non-Authoritative Information
204: No content

Level 400

400: Bad Request
401: Unauthorized
403: Forbidden
404: Not Found
409: Conflict

Level 500

500: Internal Server Error
501: Not Implemented
502: Bad Gateway
503: Service Unavailable
504: Gateway Timeout
599: Network Timeout

Il server deve inserire il codice opportuno nel messaggio di risposta al client.

Simple HTTP Server

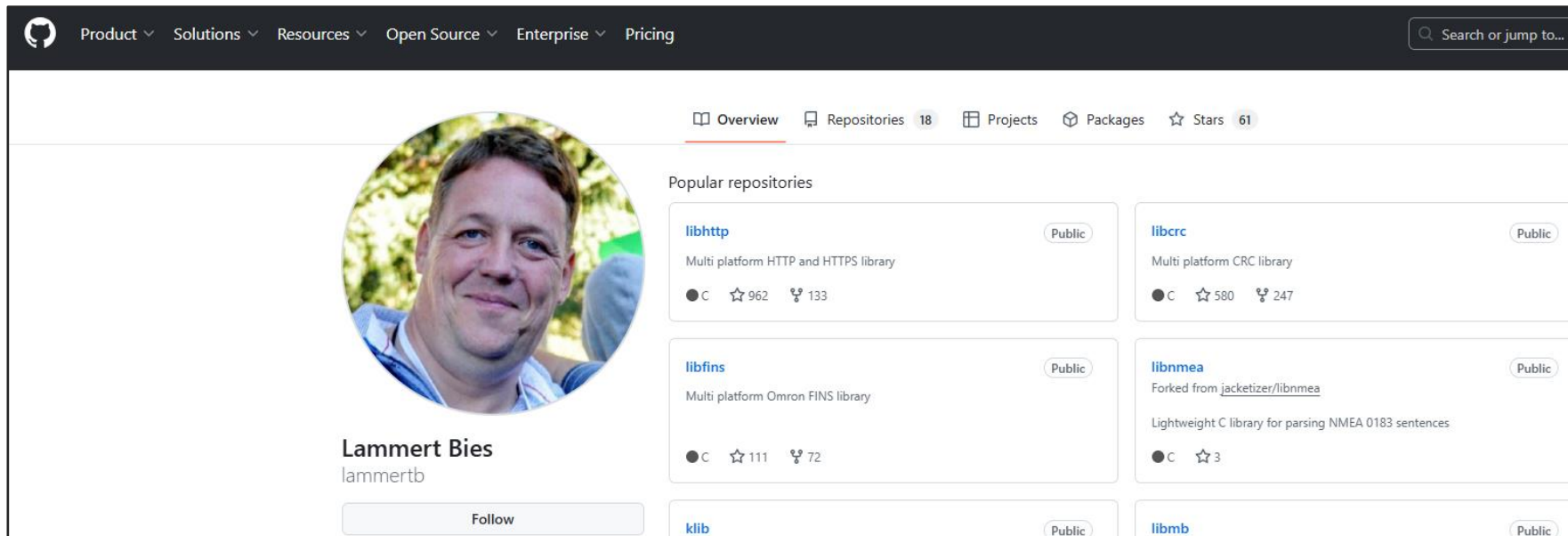
LibHTTP – libreria C

Questa esercitazione è relativa ad un semplice server HTTP, scritto «from scratch», in C. Valido come PoC (Proof of Concept), in modo da poter comprenderne il funzionamento.

Per esigenze operative reali, esistono ausili per non dover riscrivere tutto da capo. Fra questi: LibHTTP: <https://www.libhttp.org/>

LibHTTP è una libreria con licenza MIT scritta in C che implementa un server HTTP/HTTPS con capacità websocket.


- include anche funzionalità per connessioni client ad altri server.
- è basata sulla famiglia di server HTTP Mongoose(MIT)/[Civetweb](#).
- condivide il codice con questi, sebbene la compatibilità tra chiamate di funzione non sia garantita.
- comprende la [documentazione online](#) su come utilizzare la libreria.



Product ▾ Solutions ▾ Resources ▾ Open Source ▾ Enterprise ▾ Pricing

Search or jump to...

Overview Repositories 18 Projects Packages Stars 61



Lammert Bies
lammertb

Follow

Popular repositories

Repository	Language	Stars	Forks	Public
libhttp	C	962	133	Public
libcrc	C	580	247	Public
libfins	C	111	72	Public
libnmea	C	3	0	Public
klib	C	0	0	Public
libmb	C	0	0	Public

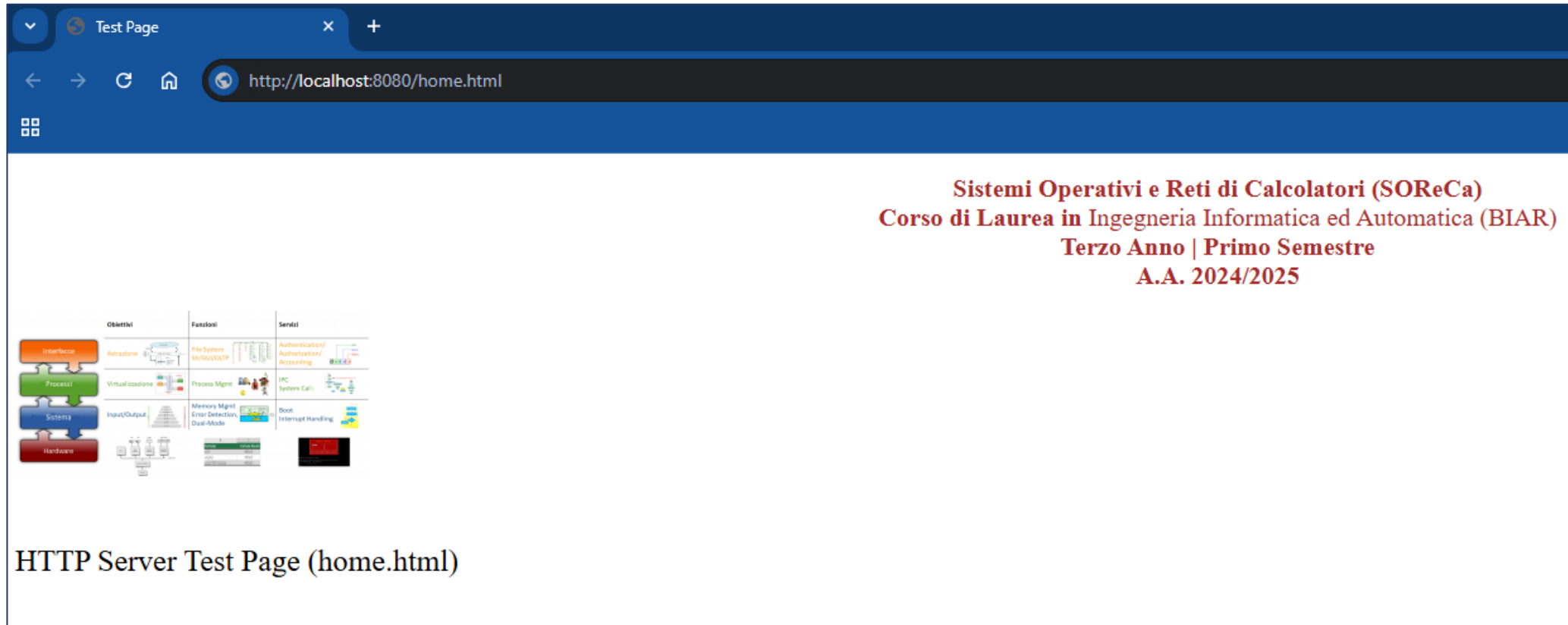
Simple HTTP server

Lab09, es.1 nano HTTP server

- Ascolto: Elaborare Thread-based (Pooling), GET valida, nome file e sua codifica
- Risposta: tipo MIME, Ricerca file e sua Esistenza, Messaggio HTTP

Simple HTTP Server

Lab09, es1 - Server HTTP client: normale browser



The screenshot shows a web browser window with a single tab titled "Test Page". The address bar displays "http://localhost:8080/home.html". The page content is as follows:

Sistemi Operativi e Reti di Calcolatori (SORECa)
Corso di Laurea in Ingegneria Informatica ed Automatica (BIAR)
Terzo Anno | Primo Semestre
A.A. 2024/2025

	Obiettivi	Funzioni	Servizi
Interfaccia	Adreazione	File System SU/NU/VOIP	Authentication/ Authorization/ Accounting
Processi	Virtualizzazione	Process Mgmt	IPC System Calls
Sistema	Input/Output	Memory Mgmt Error Detection Dual-Mode	Boot Interrupt Handling
Hardware			

HTTP Server Test Page (home.html)

Simple HTTP Server

Lab09, es1 – gestione stdout: [snprintf\(\)](#)

Il server HTTP ha un valore diverso per stdout.

stdout: per essere visibile al browser → deve essere reindirizzato sul socket di rete, previo passaggio per un buffer di memoria.

Per questo motivo, viene impiegata la funzione `snprintf()`, contenuta in `<stdio.h>`:

```
#include <stdio.h>
int snprintf(char *restrict s, size_t n, const char * format,
...);
```

s: puntatore al buffer

n: numero massimo di byte che saranno scritti nel buffer

format: stringa C che contiene una stringa di formato che segue le stesse specifiche di format in `printf()`.

...: argomenti facoltativi (formati stringa come: “%d”, myint, come per `printf()`).

Nella soluzione, sono usati due buffer:

1. header: per costruire l’header della risposta positiva
2. response: che contiene il payload (html o img o altro) o la risposta negativa

Simple HTTP Server

Lab09, es1 – gestione `stderr`: [`perror\(\)`](#)

Il server HTTP indirizza le informazioni di errore su `stderr`.

`stderr`: per visualizzare anche le informazioni di errore: `errno` settato. Viene impiegata la funzione `perror()`.

Come al solito, è necessario assicurarsi che la funzione `perror()` venga richiamata immediatamente dopo che una funzione della libreria ha restituito un errore; altrimenti, le chiamate successive potrebbero modificare il valore `errno`.

```
#include <stdio.h>
#include <errno.h>
void perror(const char *s);
s: puntatore alla stringa da stampare (es. "Could
not open data file")
Stampa
s + ": " + <messaggio associato al valore in
errno> + \n
```

<code>errno</code>	Descrizione
EADDRINUSE	Address already in use
EALREADY	Connection already in progress
ENOENT	No such file or directory
EOWNERDEAD	Owner died
EPERM	Operation not permitted

Per una lista completa dei codici di errore disponibili sul Sistema: `$ errno -l`

Simple HTTP Server

Lab09, es1 - 1 Ascolto:- Elaborare le richieste dei client (thread separati)

Continuare ad ascoltare le richieste degli utenti creando un ciclo continuo (`while (1) { [...] }`).

Qui sono accettate le richieste dei client.

Utilizzare le funzioni `pthread` per invocare le esecuzioni multithread.

Creare una funzione per gestire le richieste dei client: `void *handle_client(void *arg)`

Più processi parallelamente. Quindi più utenti possono utilizzare le risorse del server senza attendere tempi di inattività.

Per ogni nuovo utente, creiamo un nuovo thread per eseguire i suoi processi in isolamento.

Simple HTTP Server

Lab09, es1 - 2 Ascolto: Verificare se GET valida

Occorre ricevere dati dal lato client, che in genere sono messaggi che richiedono una risorsa specifica dal server.

Quindi, se tale richiesta viene ricevuta correttamente, possiamo controllare il metodo di richiesta per determinare se si tratta di una richiesta GET, PUT, POST, DELETE. Nota che, in questa implementazione, ci concentreremo solo sulle richieste GET.

La generica GET ben-formata è descrivibile dalla RegEx:

```
"^GET / ([^ ]*) HTTP/1«
```

È possibile effettuare la comparazione tramite la funzione:

```
#include <regex.h>  
int regcomp(regex_t *preg, const char *pattern, int cflags);
```

già implementata. 😊

Simple HTTP Server

Lab09, es1 - 3 Ascolto: Codifica ed effettivo nome del file

La funzione chiamata `url_decode()` gestisce le decodificazioni delle richieste URL.

Cioé, se l'URL richiesto dall'utente contiene valori esadecimali, decodifica tali valori in formato UNICODE.

Ad esempio `%20` significa spazio e `%23` significa #.

Funzione `char *url_decode(const char *src)` già definita. 😊

Simple HTTP Server

Lab09, es1 - 4 Risposta: Estensione del file e tipo MIME

Creare un'altra funzione chiamata `get_file_extension()` per leggere ed estrarre il tipo di file.

Ciò è necessario perché nel contesto HTML codificheremo i nostri payload in formato MIME.

Se il nome del file segue `FILE_NAME.file_type`, allora restituisce `file_type` come risultato, altrimenti restituisce una stringa vuota `""`.

```
const char *get_file_extension(const char *file_name) {  
    const char *dot = strrchr(file_name, '.');  
    if (!dot || dot == file_name) {  
        return "";  
    } // elimina anche i directory traversal (es. ../../..)   
    return dot + 1;  
}
```

```
const char *get_mime_type(const char *file_ext);
```

già definita. 😊

Simple HTTP Server

Lab09, es1 - 5 Risposta: Ricerca del file

Poiché il server risponderà con pagine HTML, innanzitutto occorre leggere i file dal disco rigido.

Creare una funzione chiamata `get_file_case_insensitive(const char *file_name)` per reperire il file. **già definita.** 😊

Questa funzione sostanzialmente prende la directory e il nome del file dai parametri e prova a creare il a directory completa e ad aprire il file effettivo.

Attenzione Maiuscole/minuscole

Simple HTTP Server

Lab09, es1 - 6 Risposta: Esistenza del file

provare ad aprire la risorsa richiesta (file) e se il file esiste allora possiamo continuare con gli altri passaggi, ma se non esiste allora rispondiamo all'utente come

404 Not Found
che significa che la risorsa richiesta non è stata trovata sul server.

Se la risorsa richiesta esiste, allora dobbiamo specificare alcuni metadati associati alla risorsa come Content-Length.

```
int file_fd = open(file_name, O_RDONLY);
if (file_fd == -1) {
    snprintf(response, BUFFER_SIZE,
             "HTTP/1.1 404 Not Found\r\n"
             "Content-Type: text/plain\r\n"
             "\r\n"
             "404 Not Found");
    *response_len = strlen(response);
    return;
}
```

Simple HTTP Server

Lab09, es1 - 7 Risposta: Creazione messaggio HTTP

Considerando che `build_http_response()` è una delle funzioni più importanti in questo server web. Perché come suggerisce il nome, questa funzione restituisce le risposte all'utente leggendo i file HTML effettivi che risiedono nel disco rigido.

Come primo passaggio, quando l'utente richiede una risorsa dalla dimensione del client, dobbiamo restituirla all'utente allegando i file di intestazione necessari. Quindi il renderer della dimensione del client, che in genere è un browser web, può facilmente rendere quel contenuto.

Qui stiamo allegando alcuni dei file di intestazione HTTP più importanti, come HTTP/1.1, che determina che utilizzeremo HTTP sulla connessione con lo stato predefinito 200 OK.

```
void build_http_response(const char *file_name,
                        const char *file_ext,
                        char *response,
                        size_t *response_len) {
    // build HTTP header
    const char *mime_type =
get_mime_type(file_ext);
    char *header = (char *)malloc(BUFFER_SIZE *
sizeof(char));
    snprintf(header, BUFFER_SIZE,
             "HTTP/1.1 200 OK\r\n"
             "Content-Type: %s\r\n"
             "\r\n",
             mime_type);
```

Simple HTTP Server

Lab09, es1 - 8 Risposta: Invio messaggio HTTP

Inviare la risposta sulla connessione HTTP verso il client.

```
send(client_fd, response, response_len, 0);
```

Simple HTTP Server

Lab09, es1

Completare il codice dell'HTTP Server in modalità multi-thread

Sorgenti

- Makefile
- Client: <none> usare il browser
- Server: `server.c`
- Pagina HTML: `home.html`
- Immagine nella pagina HTML: `osmatrix3x3_0.png`

Suggerimento: seguire i blocchi di commenti inseriti nel codice

Funzionamento:

- Avviare il server: `./server`
- Aprire nel browser l'URL `http://localhost:8080/home.html`
- Per vedere i thread avviati, da riga di comando:

```
ps -e -T | head -1 ; ps -e -T | grep multithread
```