UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

RETI DI CALCOLATORI 2021/2022

Francesco Crisci 1219620

Contents

Chapter 1

ClientWeb

1.1 Client Web in teoria

Quando si parla delle applicazioni in rete si f quasi sempre riferimento al modello client server. Nel nostro modello, il

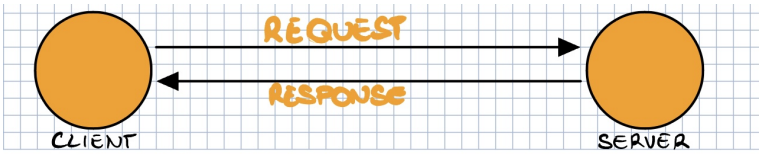


Figure 1.1: Modello Client Server

server ed il client sono due programmi, di fatto non parliamo dell'architettura hardware dei due sistemi.
Il client effettuerà una request al server, che risponderà con una response, proprio come visto nel modello osi.
Questo modello definisce che lo scambio di dati avviene secondo una certa sequenza. Il client prende l'iniziativa ed il server è in attesa di una richiesta, quella di un client. Come conseguenza, il server invia una risposta al client.A differenza dei nodi della rete, il modello client e server hanno il vincolo di richiesta e risposta. Vengono così definiti i vincoli del modello client server:

- il client prende iniziativa
- il server aspetta le richieste (è sempre attivo)
- la risposta segue ad una richiesta con response time il più piccolo possibile, ovvero minimo.

Esistono ovviamente delle alternative a questo modello, come il publish, subscribe e notify, riportato in figura 1.2
Ogni qual volta il publisher pubblica qualcosa, anche se il subscriber non ha fatto nulla, il notifier invia delle informazioni,

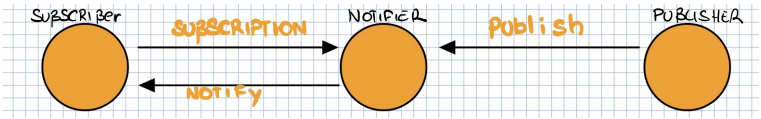


Figure 1.2: Modello Subcribe, Notify, Publish

a lui e a tutti gli altri subscribers. Questo modello non ha nulla a che fare con il client server.
Un ulteriore modello è la rete P2P (peer-to-peer), un modello che prevede che i nodi fungino sia da client che da server (figura 1.3).
Un nodo cerca una risorsa che potrebbe essere contenuta in un altro nodo, ovvero attraversa i suoi vicini e continua ad esplorare la rete fino a che non trova la risorsa cercata. Ogni nodo funge sia da client che da server, quindi a livello concettuale è come se si avesse un cammino diretto sino al nodo finale. L'esistenza di più cammini che congiungono un nodo alla risorsa non è esclusa.

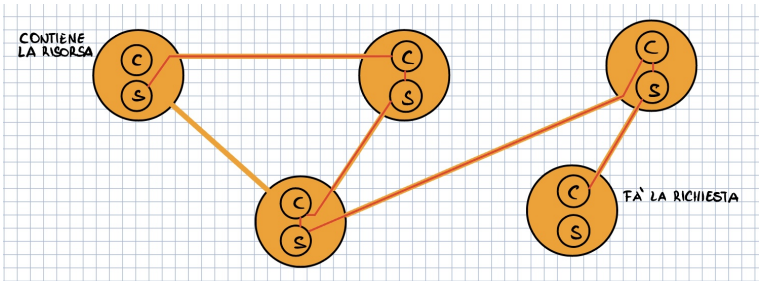


Figure 1.3: Modello Peer-to-Perr

1.2 Client Web 0.9

1.2.1 parte teorica

Nella sua forma originale prevedeva il caricamento di un contenuto di un file all'interno di un server remoto, di fatto la richiesta veniva implementata come una stringa del tipo "GET Filename". La risposta del server risultava essere il contenuto del file richiesto.
Usando i servizi del livello di trasporto, il livello 4 del modello OSI, possiamo fare una richiesta ad un server, nel nostro caso il server di google.

socket

I socket sono gli endpoint della comunicazione. Accettano 3 paramteri, ovvero dominio, tipo e protocollo. Come dominio usiamo l'AF_INET che riguarda i protocolli IPv4. Come tipologia usiamo solitamente SOCK_STREAM e SOCK_DGRAM. Il primo fornisce una connessione sequenziale, affidabile e bidirezionale di byte, mentre il secondo è un servizio di messaggi aventi una lunghezza massima.
I protocolli che implementano questi due servizi sono rispettivamente il tcp e l'udp. Il tcp è affidabile ma non fornisce feedback sul completamento o meno dell'operazione richiesta. L'udp è meno affidabile, ma fornisce riscontro sia sulla ricezione del messaggio che dell'azione che esso ha effettuato.
Il protocollo HTTP lavora su un servizio di tipo SOCK_STREAM, ovvero su un servizio tcp.
L'ultimo parametro che definisc il socket è il protocollo, che specifica le scelte fatte nei primi due parametri.
Una volta che il socket è inizializzato, viene creato un endpoint per la comunicazione e restituisce un descrittore file che riferisce a quell'endpoint. Il file descriptor deve armonizzarsi con tutto ciò che è presente all'interno del sistema operativo. Ovvero, risulta essere un indice in una tabella, indice che verrà utilizzato dal socket.

Struttura indirizzo

L'address, ovvero l'indirizzo, identifica l'host o più precisamente l'interfaccia ip dell'host. In un host possono girare tanti programmi che possono avere connessioni di rete indipendenti.
Un server può ospitare più servizi, per cui un indirizzo solo non è sufficienteper definire a quale server e a quale servizio io mi voglia collegare. Viene introdotta così un nuovo identificatore, il port, che permette di identificare il punto di accesso ad un socket presente i una macchina.
Attraverso l'uso di un port remoto è possibile far arrivare un pacchetto al server web desiderato. Il port deve dunque essere incluso all'interno del pacchetto stesso.
L'indirizzo di un socket viene determinato in maniera univoca a livello globale dalla combinazione indirizzo remoto e port remoto, AR e PR. Il socket deve quindi essere associato a un indirizzo e a un port.
Il port utilizzato all'interno del protocollo http è il port 80, mentre il port per il protocollo https è il port 443.

1.2.2 parte pratica

```
1 #include<stdio.h>
2 #include<sys/types.h>
3 #include<sys/socket.h>
4 #include<errno.h>
5 #include<arpa/inet.h>
6 #include<stdint.h>
7 #include<string.h>
8 #include<unistd.h>
```

```c
struct sockaddr_in remote;

int main(){
        int s,n;
        char request[100]= "GET / \r\n";
        char response[1000000];
        unsigned char ipserver[4] = {142,250,180,3};//Server di google
        s = socket(AF_INET,SOCK_STREAM,0);          //IL mio socket
        if(s == -1){                                //Controllo che il socket exist
                printf("errno = %d\n",errno);       //Printo l'errore
                perror("socket fallita");           //Perror della socket fallita
                return -1;                          //Termino il programma
        }
        remote.sin_family = AF_INET;                //Inizializzo la struttra
        remote.sin_port = htons(80);                //Il port alla quale mi connect
        remote.sin_addr.s_addr = *((uint32_t*)ipserver);//L'ip per la connect
        if(connect(s,(struct sockaddr*)&remote, sizeof(remote)) == -1){//check the conn
                printf("errno = %d\n",errno);       //In case is an error print
                perror("connect fallita");          //stuff.
                return -1;
        }
        write(s,request,strlen(request));
        size_t len = 0;

        //Si puo usare questo ciclo
        while(n = read(s,response + len,1000000-len)){
                len += n;
        }
        //O in alternativa quest'altro
        for(len = 0;n = read(s,response+len, 1000000-len);
        //Fai attenzione ad usarne solamente uno!!!
        printf("%s\n",response);
}
```

La richiesta del client web 0.9 prevede una semplice GET con un singolo CRLF, il tutto viene memorizzato all'interno di una request, che nel nostro caso ha dimensione 100.
La response è un buffer di dimensione arbitraria molto grande, nel nostro caso un millione.
La variabile ipserver contiene un insieme di 4 interi che rappresentano l'indirizzo ip alla quale ci vogliamo collegare.
Il socket s rappresenta un numero intero che parte obbligatoriamente da 3 in quanto i valori 0,1,2 sono già in uso per standard input, output e errore. In caso la creazione del socket vada male, il valore restituito è -1.
La struttura remote è una struttura di tipo sockaddr_in che viene messa a disposizione da una delle librerie riportate in cima. è costituita da 3 campi fondamentali, ovvero:

- sin_family che indica la famiglia di protocolli che si vogliono usare. Dato che usiamo l'IPv4 useremo la famiglia AF_INET.
- sin_port che indica il numero della porta che vogliamo utilizzare, nel nostro caso il port 80, che deve essere obbligatoriamente espresso in big endian (a questo scopo si può utilizzare la funzione htons(int) che converte un intero in big endian).
- L'ultimo parametro che vogliamo configurare è il sin_addr.s_addr, che risulta essere l'indirizzo remoto alla quale ci vogliamo collegare, nel nostro caso è un cast un pò particolare della variabile ipserver. più nello specifico utilizziamo un cast ad un puntatore a uint32_t per poi accedere al valore completo.

Tutti e 3 i parametri riportati qui sopra sono membri accessibili dalla struttura remote.
Una volta inizializzata la struttura remote possiamo effettuare la connect al server remoto attraverso l'utilizzo della funzione connect().
La connect accetta 3 parametri e restituisce un numero intero. in caso l'intero restituito è -1 la connect è fallita. I parametri che la connect necessità sono:

- socket, che nel nostro caso è il socket s iniziallizato precedentemente
- un puntatore ad una struttura sockaddr che possiamo ottenere attraverso il seguente cast della struct remote: (struct sockaddr*)&remote.
- la dimensione della struct passata come secondo parametro, ottenibile attraverso l'uso del sizeof come segue: sizeof(remote).

Successivamente alla connect dobbiamo effettuare sul socket della request e leggere i dati che vengono mandati indietro dal server web. La write accetta 3 parametri che sono il socket, la stringa contenente la request (ovviamente una stringa di tipo C), nel nostro caso l'array request, e la lunghezza della request, ottenibile attraverso l'uso della funzione strlen(char[]).
Dopo la write, il nostro compito è quello di leggere i dati dal buffer. Si possono usare diversi approcci per ottenere lo stesso risultato. La metodologia più compatta è un ciclo for del seguente tipo: for(len = 0;n = read(s,response+len, 1000000-len);
La condizione che ci permette di iterare attraverso il ciclo è che la read ritorna 0 quando non ha più caratteri da leggere, quindi questo ciclo consuma tutti i caratteri che vengono mandati indietro dal server web.
Dato che vogliamo leggere tutta la response assieme, sovrascriviamo il buffer response shiftando la memorizzazione dei caratteri letti tramite il response + len, così facendo otteniamo la response completa senza perdere i dati nel buffer.
L'ultimo parametro che notiamo all'interno della read è 1000000-len, che ci indica che stiamo togliendo dalla dimensione del buffer i caratteri già letti dalla read. Questo ci aiuta ad evitare che si provi a scrivere più dati di quelli che possono essere memorrizati all'interno del buffer, evitando che si sovrascrivano celle in memoria non appartenenti al buffer.
Dopo il ciclo di read e la memorizzazione della risposta del server nel buffer response possiamo stampare in output il contenuto della response.

1.3 Client Web 1.0

1.3.1 Parte teorica

L'http permette di utilizzare una serie i metodi espandibili da usare per indicare la finalità di una richiesta, ad esempio la GET è solo una delle funzioni che può essere eseguita.
I sistemi infromativi si costruiscono sopra la disciplina di riferimento fornito dall'URI, l'Uniform Resources Identifier. Come l'URL, si ha l'informazione di dove il documento si trova, l'indirizzo dell'host, e viene aggiunto il path all'interno della quale il documento si trova nel sistema. L'URL e l'URN sono due esempi di URI.
Questo protoollo viene utilizzato anche come generico protocollo di comunicazione tra utenti (users agent) e proxies/gateways.
Vediamo ora un pò di terminologia utile:

- Messaggio: unità base della comunicazione http, che consiste in una sequenza strutturata di byte che matchano la sintassi definita nella sezione 4 e trasmessi tramite connessione.
- Request: una richiesta http come quella del modello client server.
- Response: una risposta http.
- Resource: un network data object o un servizio che può essere identificato da un URI. Sequenza di byte che si usano per implementare quel servizio. Può essere inclusa sia in una richiesta che in una risposta.
- Entity: unaparticolare rappresentazione, o rendering, di una risorsa di dati. Sequenza di byte che si usano per implementare quel servizio. Può essere inclusa sia in una richiesta che in una risposta.
- Client: un programma applicativo che stabilisce connessioni per mandare richieste. Un esempio è il client web 0.9 scritto da noi.
- User agent: il client che inizia una richiesta. Spesso browser, editors, o altri tool dal lato utente.
- Server: un programma appplicativo che accetta connessioni per servire richieste mandando indietro risposte.
- Origin server: il server sulla quale una risorsa risiede o sta per essere creata.
- Proxy: un programma intermediario che si comporta sia da client che da server. Un filtro online.
- Gateway: server intermediario tra altri server.
- Tunnel: intermediario che si comporta come relay cieco tra due connessioni.
- Cache: lo store locale di un programma memorizza i messaggi e il sottosistema che controlla la la memorizzazione, il ritrovo e la eliminazione dei suoi messaggi.

Request

I tipi di richiesta sono divise in due macrocategorie, la simple request e la full request.
La simple request è la richiesta che corrisponde all'http 0.9, mentre la full request corrisponde all'http 1.0 e riesce a gestire ancora le richieste del tipo http 0.9.
La simple request è la request implementata nell'http 0.9, ovvero la semplice GET, mentre la full request ha un'implementazione del genere:

```
full-request = reuqest line
               *(general-header | request-Header | entity-header)
               CRLF
               Entity-body
```

Quindi risulta possibile fare una richiesta del genere:
request-line — general header — request header — entity hader — request — CRLF — Entity body.
La simple request non presenta nè header nè entity body, mentre l'http 1.0 e 1.1 si.

Request Line

Il raffinamento della request line è dato dalla struttura Method SP Request-URI SP HTTP-version CRLF. I method sono descritti all'interno della grammatica e includono get, head, post e extension methods.
Il request-URI include il path del file system del server.
L'http version è un nuovo elemento, ed è composto da un major number e da un minor number. La sua sintassi è la seguente: "HTTP" "/" 1*DIGIT "." 1*DIGIT.
La logica vorrebbe che il primo elemento del pacchetto fosse il tipo, così so come leggere gli altri byte, ma purtrobbo data la retro-compatibilità con l'http 0.9 essa viene messo in fondo alla request.

Status Line

la status line è composta grammaticalmente come segue: HTTP-Version SP Status-Code SP Reason-Phrase CRLF. In questo caso la versione viene specificata prima in quanto la status line non è presente all'interno dell'http 0.9 e quindi non comporta problemi di retro-compatibilità.
Gli status code sono invece identificati da un numero o da una frase, che mi specifica lo stato di quella richiesta, ovvero se c'è stato un errore o meno. Il numero specifica il tipo di errore e posso fare in modo che il software lo riesca a gestire. Lo status code è composto da 3 cifre la quale la prima definisce 5 famiglie:

- 1xx: informativo, non viene usato ma viene riservato al futuro.
- 2xx: successo, l'azione è stata ricevuta con successo, capita ed accettata.
- 3xx: redirection, altre azioni devono essere intraprese per completare la richiesta.
- 4xx: Client Error, la richiesta contiene sintassi sbagliata o non può essere soddisfatta.
- 5xx: Server Error, il server ha fallito a soddisfare una richiesta apparentemente valida.

Header

Vediamo ora la struttura di un header, visibile in figura 1.5. Come si può vedere, dopo l'http-version e lo status-code,

Figure 1.4: Struttura della status line

sono presenti una serie di header rappresentati dalla coppia "name" : "value", che verranno memorizzati all'interno una struttura header che vedremo meglio all'interno della parte pratica.

1.3.2 Parte pratica

```c
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<errno.h>
#include<arpa/inet.h>
#include<stdint.h>
#include<string.h>
#include<unistd.h>
struct sockaddr_in remote;

int main(){
        int s,n;
        char request[100]= "GET /eolomammolo:HTTP/1.0\r\n\r\n";
        char response[1000000];
        unsigned char ipserver[4] = {142,250,180,3};//Server di google
        s = socket(AF_INET,SOCK_STREAM,0);          //IL mio socket
        if(s == -1){                                //Controllo che il socket exist
                printf("errno = %d\n",errno);       //Printo l'errore
                perror("socket fallita");           //Perror della socket fallita
                return -1;                          //Termino il programma
        }
        remote.sin_family = AF_INET;                //Inizializzo la struttra
        remote.sin_port = htons(80);                //Il port alla quale mi connect
        remote.sin_addr.s_addr = *((uint32_t*)ipserver);//L'ip per la connect
        if(connect(s,(struct sockaddr*)&remote, sizeof(remote)) == -1){//check the conn
                printf("errno = %d\n",errno);       //In case is an error print
                perror("connect fallita");          //stuff.
                return -1;
        }
        write(s,request,strlen(request));
        size_t len = 0;
        for(len = 0;n = read(s,response+len, 1000000-len);
        printf("%s\n",response);
}
```

Come si può notare la struttura del client web 1.0 è molto simile alla struttura del client web 0.9.
La differenza più importante da notare, ed anche l'unica, è la struttura della request.
Come si può notare, la request presenta un CRLF in più alla fine della request, presenta una :HTTP/VERSION ed un nome di seguito allo / presente all'interno della GET.
Questa struttura è diversa anche dal tipo di request usata per effettuare richieste attraverso l'uso dell'http 1.1 che vedremo nel prossimo paragrafo.

1.4 Client Web 1.1

```c
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<errno.h>
#include<arpa/inet.h>
#include<stdint.h>
#include<string.h>
#include<unistd.h>
#include<stdlib.h>
struct sockaddr_in remote;
struct header{
        char* n;
        char* v;
}h[100];                //buffer that will contain our headers
char response[1000001];
int main(){
        int s,n,i,j,k,bodylen;
        size_t len = 0;
        char* statusline;
        char hbuffer[10000];
        char* request = "GET / HTTP/1.1\r\nHost:www.google.com\r\n\r\n";

        unsigned char ipserver[4] = {142,250,180,3};//Server di google
        s = socket(AF_INET,SOCK_STREAM,0);          //IL mio socket
        if(s == -1){                                //Controllo che il socket exist
                printf("errno = %d\n",errno);       //Printo l'errore
                perror("socket fallita");           //Perror della socket fallita
```

```
28                  return -1;                          //Termino il programma
29          }
30          remote.sin_family = AF_INET;                //Inizializzo la struttra
31          remote.sin_port = htons(80);                //Il port alla quale mi connect
32          remote.sin_addr.s_addr = *((uint32_t*)ipserver);//L'ip per la connect
33          if(connect(s,(struct sockaddr*)&remote, sizeof(remote)) == -1){//check the conn
34                  printf("errno = %d\n",errno);       //In case is an error print
35                  perror("connect fallita");          //stuff.
36                  return -1;
37          }
38          for(k = 0;k<1;k++){
39                  write(s,request,strlen(request));
40                  bzero(hbuffer,10000);
41                  statusline = h[0].n = hbuffer;
42                  for(i=0,j=0;read(s,hbuffer+i,1);i++){
43                          if((hbuffer[i]=='\n') && (hbuffer[i-1]=='\r')){
44                                  hbuffer[i-1] = 0;
45                                  if(!h[j].n[0])break;
46                                  h[++j].n=hbuffer+i+1;
47                          }
48                          if(hbuffer[i]==':' && !h[i].v){
49                                  hbuffer[i]=0;
50                                  h[j].v=hbuffer+i+1;
51                          }
52                  }
53                  bodylen=1000000;
54                  for(i=1;i<j;i++){//Ricordati che il primo header e' particolare
55                          printf("%s------>%s\n",h[i].n,h[i].v);
56                          if(!strcmp("Content-Length",h[i].n))bodylen=atoi(h[i].v);
57                  }
58                  for(len = 0;len<bodylen &&(n = read(s,response + len,1000000-len))>0;len+=n);
59                  if(n==-1){perror("read fallita");return-1;}
60                  response[len]=0;
61                  printf("%s\n",response);
62          }
63  }
```

Vediamo ora di aggiungere un po' di commenti al codice, così da poter analizzare al meglio il suo funzionamento.
La prima differenza che notiamo è ancora una volta all'interno della request, come è possibile notare l'HTTP/VERSION si trova subito dopo lo / successivo alla GET, ed è seguito da un CRLF. Dopo il CRLF troviamo la categoria Host: seguita dal nome dell'host alla quale ci vogliamo connettere, nel nostro caso www.google.come. Alla fine della request sono presenti 2 CRLF così come che nel client web 1.0.
Un'altra differenza è presente dopo che è stata effettuata la connect, infatti possiamo notare un ciclo for. Questo ciclo serve per poter permettere al client web di effettuare più richieste successive. All'interno di questo ciclo troviamo la solita write, seguita dall'inizializzazione del buffer per gli header a zero (viene utilizzato la funzione bzero(array,size)).
Successivamente viene assegnato al puntatore statusline la locazione della struttura h (struttura che contiene la coppia nome-valore di tutti gli header) che punta a sua volta all'inizio di hbuffer, ovvero il buffer sulla quale verranno memorizzati temporaneamente tutti gli header mandati in risposta dal server web. La riga di codice che ci permette di effettuare questo è la seguente:
statusline = h[0].n = hbuffer;
Dopo l'inizializzazione delle variabili riportate qui sopra, avviene il parsing degli header, ovvero viene scandito l'hbuffer per permettere di memorizzare le coppie nome-valore di tutti gli header.
Ogni header viene mandato come segue: "name":"value" CRLF, ad eccezione dell'ultimo header che ha una struttura del tipo: CRLFCRLF, ovvero corrisponde ad un header nullo.
Il parsing ci permette di suddividere i vari header, associando ad ognuno di essi il rispettivo nome con il corrispondente valore, rimuovendo i ':' all'interno della loro denominazione.
Questo passaggio risulta essere molto importante, in quanto permette di riconoscere header specifici come il Content-Length che indica la dimensione totale della pagina o il Transfer-Encoding che potrebbe permetterci di notare che la codifica mandata risulta essere di tipo chunked e che quindi la lettura della response del server deve essere interpretata in maniera differente.
Come è possibile notare, successivamente al parsing viene effettuata una scansione su tutti gli header, ad eccezione del primo, per vedere se viene trovata una Content-Length. La Content-Length ci permette di sapere quanti byte sono stati mandati dal server, ovvero ci indica quanti caratteri devono essere letti dopo gli header, o più precisamente quanto grande risulta essere l'entity body della response.
Di fatto, successivamente alla scansione degli header, viene effetttuato il classico ciclo for per riempire il buffer di risposta ponendo particolare attenzione a che i caratteri letti non sorpassino la grandeza dell'entity body, ovvero del valore memorizzato all'interno della variabile bodylen.

1.5 Client Web 1.1 e Transfer-Encoding: Chuncked

```c
#include<stdio.h>
#include <string.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <errno.h>
#include <arpa/inet.h>
#include <stdint.h>
#include <unistd.h>
#include <stdlib.h>

struct sockaddr_in remote;
char response[1000001];
struct header {
        char * n;
        char * v;
} h[100];

int main()
{
    size_t len = 0;
    int i,j,k,n,offset,chunklen,bodylen=0;
    char * request = "GET / HTTP/1.1\r\nHost:www.google.it\r\n\r\n";
    char * statusline;
    char hbuffer[10000], ckbuffer[100];
    unsigned char ipserver[4] = { 142,250,180,3};
    int s;
    if (( s = socket(AF_INET, SOCK_STREAM, 0 )) == -1){
        printf("errno = %d\n",errno); perror("Socket Fallita"); return -1;
    }
    remote.sin_family = AF_INET;
    remote.sin_port = htons(80);
    remote.sin_addr.s_addr = *((uint32_t *) ipserver);
    if ( -1 == connect(s, (struct sockaddr *)&remote,sizeof(struct sockaddr_in)))
    {
        perror("Connect Fallita"); return -1;
    }
    for(k=0; k < 1; k++){
            write(s,request,strlen(request));
            bzero(hbuffer,10000);
            statusline = h[0].n = hbuffer;
            for (i=0,j=0; read(s,hbuffer+i,1); i++) {
                if(hbuffer[i]=='\n' && hbuffer[i-1]=='\r'){
                    hbuffer[i-1]=0; // Termino il token attuale
                    if (!h[j].n[0]) break;
                    h[++j].n=hbuffer+i+1;
                }
                if (hbuffer[i]==':' && !h[j].v){
                        hbuffer[i]=0;
                        h[j].v = hbuffer + i + 1;
                }
            }
            bodylen=0;
            for(i=1;i<j;i++){
                printf("%s ----> %s\n",h[i].n,h[i].v);
                if(!strcmp("Content-Length",h[i].n))
                        bodylen=atoi(h[i].v);
                if(!strcmp("Transfer-Encoding",h[i].n))
                    if (!strcmp(" chunked",h[i].v)) bodylen = -1;
            }
            if(bodylen == -1){
                offset = 0;
                chunklen = 1;
                while(chunklen){
                    chunklen = 0;
                    for(j=0; (n = read(s, ckbuffer + j, 1)) > 0 && ckbuffer[j] != '\n' && ckbuffer[j-1]
    != '\r'; j++){
                        if(ckbuffer[j] >= 'A' && ckbuffer[j] <='F')
                        chunklen =chunklen*16+(ckbuffer[j]-'A'+10);
                        if(ckbuffer[j] >= 'a' && ckbuffer[j] <='f')
                        chunklen =chunklen*16+(ckbuffer[j]-'a'+10);
                        if(ckbuffer[j] >= '0' && ckbuffer[j] <='9')
                        chunklen = chunklen*16 + ckbuffer[j]-'0';
```

```
72                      }
73                      if(n==-1){perror("Read fallita"); return -1;}
74                      ckbuffer[j-1] = 0;
75                      printf("chunklen ---> %d\n", chunklen);
76                      printf("ckbuffer ---> %s\n", ckbuffer);
77                  for(len=0; len<chunklen && (n = read(s,response + offset, chunklen - len))>0; len+=n,
    offset+=n);
78                  if (n==-1) { perror("Read fallita"); return -1;}
79                  read(s, ckbuffer, 1);
80                  if (n==-1) { perror("Read fallita"); return -1;}
81                  read(s, ckbuffer +1,1);
82                  if (n==-1) { perror("Read fallita"); return -1;}
83                  if(ckbuffer[0]!= '\r' || ckbuffer[1] != '\n'){
84                      printf("Errore nel chunk"); return -1;
85                  }
86                  }
87              response[offset] = 0;
88              printf("%s\n", response);
89          }
90      }}
```

Riportiamo di seguito il passaggio dell'RFC dedicato al chunked encoding per poi parlare del codice scritto:

3.6.1 Chunked Transfer Coding

The chunked encoding modifies the body of a message in order to transfer it as a series of chunks, each with its own size indicator, followed by an OPTIONAL trailer containing entity-header fields. This allows dynamically produced content to be transferred along with the information necessary for the recipient to verify that it has received the full message.

```
Chunked-Body   = *chunk
                 last-chunk
                 trailer
                 CRLF

chunk          = chunk-size [ chunk-extension ] CRLF
                 chunk-data CRLF
chunk-size     = 1*HEX
last-chunk     = 1*("0") [ chunk-extension ] CRLF

chunk-extension= *( ";" chunk-ext-name [ "=" chunk-ext-val ] )
chunk-ext-name = token
chunk-ext-val  = token | quoted-string
chunk-data     = chunk-size(OCTET)
trailer        = *(entity-header CRLF)
```

The chunk-size field is a string of hex digits indicating the size of the chunk. The chunked encoding is ended by any chunk whose size is zero, followed by the trailer, which is terminated by an empty line.

La struttura del client web che accetta anche Transfer-Encode chuncked è molto simile alla struttura del client web 1.1. Di fatto, viene aggiunto un controllo ulteriore al momento della scansione della struct degli header. Infatti si controlla che il nome di un header corrisponda a Transfer-Encoding e che il corrispettivo valore sia " chuncked" (lo spazio all'interno delle virgolette è voluto). Se questa condizione è soddisfatta, il bodylen viene impostato a -1.
Come possibile vedere dall'RFC, il chuncked-body, ovvero la struttura dell'entity body, è suddivida in un numero arbitrario di chunck, seguiti dall'ultimo chunk, dal trailer e da un CRLF. Ma come è fatto un chunk?
Un chunck è composto da una chunck-size, espressa in esadecimale, e viene seguita da un CRLF, seguito da i dati che lo compongono, ovvero un chunck-size espresso in ottetti, seguito in fine da un altro CRLF. C'è la possibilità che il chunk-size venga seguito da un parametro opzionale, ovvero la il nome dell'estensione del chunk, il chunk-ext-name.
Spiegata la struttura teorica del chunk passiamo ora a vedere la parte finale del codice:
Prima di tutto verifichiamo che il bodylen risulti -1, questo in quanto è il valore che abbiamo scelto per leggere lo stream dei dati con codifica chunked.

Imposto una variabile offset a 0 e la lunghezza del chunk, ovvero la chunklen as 1 momentaneamente.
Fintantoché non si trova un valore di chunklen nullo, si itera per leggere tutti i chunk.
Come primo passaggio, impostiamo la chunklen a 0, e scandiamo successivamente la risposta da parte del server. Ovvero, effettuiamo una read dal socket e salviamo il tutto su un buffer, il ckbuffer, iterando per singolo carattere. L'iterazione controlla anche che gli ultimi 2 caratteri letti non siano il CRLF, in quanto se così fosse, il ciclo si ferma. Visto che la lunghezza del chunk viene espressa in esadecimale, ogni carattere che viene letto aggiorna il valore della size dopo essere stato trasformato in decimale. Finito il ciclo abbiamo trovato la dimensione del chunk che stiamo leggendo.
Successivamente, controlliamo che il valore restituito dalla read non sia -1 così da essere sicuri che tutto sia proceduto correttamente.
Effettuato il controllo, impostiamo a 0 il penultimo carattere del ckbuffer a 0, ovvero terminiamo la stringa. Stampiamo ora la grandezza del chunk e il suo contenuto.
Possiamo ora inserire all'interno della response tutto quello che viene letto dai chunk, più precisamente controlliamo che la len sia sempre minore della chunklen e che il valore restituito dalla read sia maggiore di 0. Aggiorniamo ora la len e l'offset, che ci permettono rispettivamente di togliere dalla chunklen il numero di caratteri letti e di aggiornare il punto di scrittura sulla response.
Effetttuiamo ora che il valore di n, ovvero di caratteri letti non sia negativo, ovvero controlliamo che la read sia stata effettuata correttamente.
Leggiamo in fine gli ultimi due caratteri che ci vengono spediti e controlliamo successivamente che non siano i terminatori CRLF.
Terminati tutti i chunk, impostiamo ora l'ultimo valore di response a 0, ovvero terminiamo la stringa. Possiamo ora stampare in output il valore completo della response.

1.6 CLient web CGI

1.6.1 CGI.c

```c
#include<stdlib.h>
#include<stdio.h>
#include <string.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <errno.h>
#include <arpa/inet.h>
#include <stdint.h>
#include <unistd.h>

struct sockaddr_in local, remote;
char request[1000001];
char response[1000];

int main()
{
char * method, *url, *ver;
char * filename;
char command[100];
FILE * fin;
int c;
int n;
int i,t, s,s2;
int yes = 1;
int len;
if (( s = socket(AF_INET, SOCK_STREAM, 0 )) == -1)
        { printf("errno = %d\n",errno); perror("Socket Fallita"); return -1; }
local.sin_family = AF_INET;
local.sin_port = htons(17999);
local.sin_addr.s_addr = 0;

t= setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
if (t==-1){perror("setsockopt fallita"); return 1;}

if ( -1 == bind(s, (struct sockaddr *)&local,sizeof(struct sockaddr_in)))
{ perror("Bind Fallita"); return -1;}

if ( -1 == listen(s,10)) { perror("Listen Fallita"); return -1;}
remote.sin_family = AF_INET;
remote.sin_port = htons(0);
remote.sin_addr.s_addr = 0;
```

```
len = sizeof(struct sockaddr_in);
while ( 1 ){
        s2=accept(s,(struct sockaddr *)&remote,&len);
        len = read(s2,request,1000);
        request[len]=0;
        printf("%s",request);
        if(len == -1) { perror("Read Fallita"); return -1;}
        method = request;
        for(i=0;i<len && request[i]!=' ';i++); request[i++]=0;
        url=request+i;
        for(;i<len && request[i]!=' ';i++); request[i++]=0;
        ver=request+i;
        for(;i<len && request[i]!='\r';i++); request[i++]=0;
        if ( !strcmp(method,"GET")){
                filename = url+1;
                if(!strncmp(url,"/cgi-bin/",strlen("/cgi-bin/"))){
                                sprintf(command,"%s > tmpfile",(url+strlen("/cgi-bin/")));
                                printf("Eseguo comando %s\n", command);
                                system(command);
                                strcpy(filename,"tmpfile");
                                }
                fin=fopen(filename,"rt");
                if (fin == NULL){
                        sprintf(response,"HTTP/1.1 404 Not Found\r\n\r\n");
                        write(s2,response,strlen(response));
                        }
                else{
                        sprintf(response,"HTTP/1.1 200 OK\r\n\r\n");
                        write(s2,response,strlen(response));

                        while ( (c = fgetc(fin))!=EOF) write(s2,&c,1);
                        fclose(fin);
                        }
        }
        else {
                        sprintf(response,"HTTP/1.1 501 Not Implemented\r\n\r\n");
        write(s2,response,strlen(response));
        }
        close(s2);
}
close(s);
}
```

1.6.2 CGI-exe.c

```
#include<unistd.h>
#include<string.h>
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char * argv[], char * env[]){
        int i,j,t;
        int length;
        char line[500],*key,*value,*buffer;
        //fgets(line,500,stdin);

        //printf("Io sono il cgiexe e ho letto questa riga da stdin: %s",line);
        printf("E ho questo environment:\n");
        for(i=0;env[i];i++){
                        key = env[i];
                        for(j=0;env[i][j]!='=';j++);
                                env[i][j]=0;
                        value = env[i]+j+1;
                        printf("key:%s, value:%s\n",key,value);
        if(!strcmp(key,"CONTENT_LENGTH"))
                        length = atoi(value);
        }
        buffer=malloc(length);
        for(i=0;i<length && (t=read(0,buffer+i,length-i));i+=t);
  printf("Body received\n");
        for(i=0;i<length && (t=write(1,buffer+i,length-i));i+=t);

        printf("Ciao, muoio\n");
}
```

Chapter 2

Server Web

2.1 Il nostro primo server web

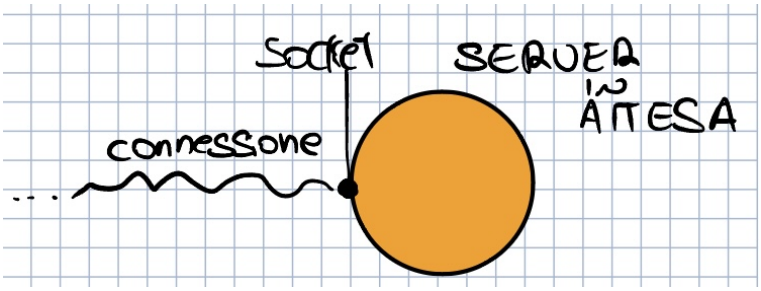


Figure 2.1: Rappresentazione del server web

Come possibile vedere in figura 2.1, anche il server ha bisogno di un socket e si trova in uno stato di attesa fintantoché un nodo non si connetta ad esso ed effettui una richiesta.
I passi che bisogna implementare in un socket sono i seguenti:

- dobbiamo aprire un socket.
- dobbiamo associare il port e l'indirizzo.
- dobbiamo rendere il socket passivo.
- dobbiamo accettare le connessioni.

Il port non può essere deciso a caso, come nel caso del client e del port usa e getta. Una chiamata al sistema rende il socket passivo.
I punti critici risiedono nel fatto che quando un utente si connette ad un web server, il port server è il port 80. In quel momento, l'utente crea una connessione con il server. Ci possono essere ovviamente più connessioni al port 80 da parte di altri client, ma come si possono amministrare tante connessioni aperte? Il server presenta socket diversi che hanno lo stesso port. Ci possono essere piu socket associati ad uno stesso port. In generale in un server, un port ha più socket associati ad esso. Più socket passivi possono avere lo stesso port.
L'operazione di associazione del port e dell'indirizzo prende il nome di binding. Vuol dire collegare il socket a un certo port e indirizzo.
Successivamente dobbiamo rendere il socket passivo, ovvero dobbiamo renderlo in grado di essere aperto da remoto. Questa operazione prende il nome di listen. L'ultima fase prende invece il nome di accept, ovvero la connessione viene accettata.
Il prototipo della bind assomiglia alla connect, a differenza che i dati utilizzati sono dati locali e non remoti. Per tutto il resto, vengono usati gli stessi parametri ma il contenuto è diverso. L'indirizzo del socket locale viene dat dalla macchina, è un bind implicito. Viene utilizzato un port usa e getta.
Il bind risulta essere un obbligo in caso di apertura di una connessione passiva. Noi useremo nuovamente la struttura sockaddr_in, ovvero la struttura specifica già usata dal client.
La listen marca il socket riferito da scokfd come passivo, ovvero in grado di accettare connessioni. La lunghezza della coda delle connessioni pending deve essere specificato attraverso il parametro int backlog.
La differenza tra essere passivi e attivi risiede nel controllo, non nelle nostre mani, di avvenuta apertura di un socket. Le

richieste che arrivano possono farlo con una densità temporale elevatissima.
Data la possibilità che due eventi arrivino molto vicini, il server deve essere in grado di bufferizzare queste richieste. Una richiesta di apertura causa un'attesa prima che venga accettata, e mi evita chdi avere eventi che possano essere rifiutati. Questo è il motivo per la quale è stato introdotto il backlog.
Le connessioni si accodano nel buffer e vengono man mano accettate dal serer. Una volta che una connessione è stata accettata tramite il metodo accept che ritorna un valore intero, molto significativo.
Il valore ritorno è un nuovo socket che presenta le seguenti caratteristiche:

- ha un nuovo numero.
- ha le stesse caratteristiche del socket creato precedentemente.
- è un socket aperto già connesso. Questa è la differenza.

A differenza della connect che torna valore 0 se tutto va bene, è equivalente al ricevere un nuovo socket tramite accept.
Arrivati a questo punto dobbiamo servire la richiesta fatta dal client.
L'int sockfd, ovvero il socket pasivo che usiamo, serve solo per accettare richieste, mentre l'accept ci fornisce il socket per comunicare.
Questa volta, la lunghezza viene passata come putatore in quanto la struttura può essere riscritta.
C'è un lasso di tempo nella quale il socket rimane in uno stato di time-out nel caso in cui dovrebbero arrivare dei pezzi di dati ancora dal client, C'è un tempo di time-out dopo la quale il socket risulta essere nuovamente disponibile.

2.1.1 Caching

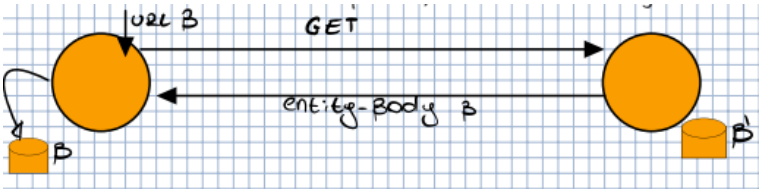


Figure 2.2: Concetto di caching

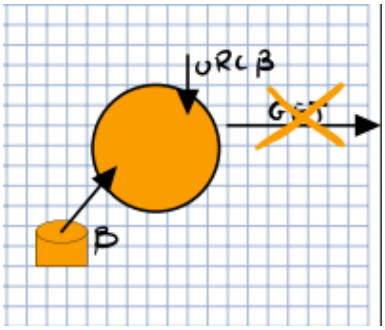


Figure 2.3: Concetto di caching 2

Come faccio a sapere che la copia sul client sia uguale a quella presente nel server? Qui è dove nasce il problema del caching. Chi mi dice che B sia uguale a B'? Vediamo ora una serie di metodologie utilizzate per ovviare a ciò:

- EXPIRE: il primo metodo utilizzato dall'http è che il client fa una richiesta ed il server risponde ed aggiunge un header particolare, lo EXPIRE: "http_date". In questo caso noi sappiamo dal server quando la cache dovrà essere ritenuta invalida. Ovviamente all'header expire viene dato anche l'header DATE: "http_date. La data di expiration del file risulta essere: expiration: 14:12 - 15:25 - 14:10, ovvero realtime() + EXPIRE-DATE. In caso i dati forniti siano già scaduti, i dati non vengono immessi nella cache. Se i dati sono forniti dinamicamente, i dati devono avere ua data di scadenza appropriata.
- If-modified-since:
  Il file viene sostituito solo se il file richiesto è stato modificato nuovamente dopo l'ultima richiesta fatta. Se ciò non accadesse, viene comunicato 304 Not Modified, ovvero non riceveremo alcun file e ci verrà comunicato che il file non è stato modificato dall'ultimo accesso al server.

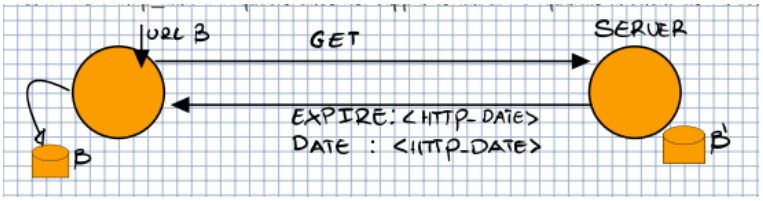


Figure 2.4: Cache con expiration

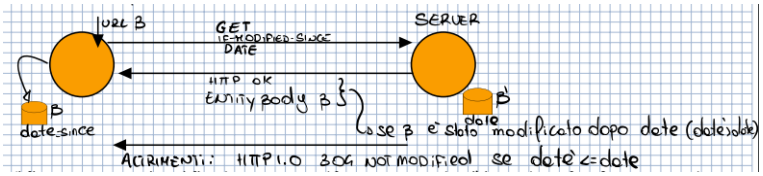


Figure 2.5: Cache con if-modified-since

- HEAD: il client richiederà un file attraverso il metodo HEAD, che impone solo di scaricare gli header. La HEAD è ua GET priva di entity body. Sia head che last-modified sono header. Nella casistica che il file richiesto mi serva

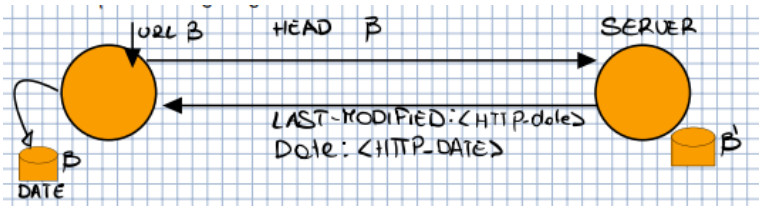


Figure 2.6: Cache con HEAD

effettivamente, si succede questa richiesta con una di tipo GET per ricevere il file in questione.

Il secondo scenario può risultare tedioso per il server, ma se il file è modificato, ho il vantaggio di avere il file già modificato. La if-modified-since è vantaggiosa se la risorsa si aggiorna spesso, mentre la HEAD è vantaggiosa se la risorsa si aggiorna raramente.
Se l'aggiornamento della risorsa è deterministico, io creo l'expire.
In caso di aggiornamento stocastico uso o if-modified-since o HEAD.
Esiste un ulteriore meccanismo che prende il nome di pragma no cache, ovvero il passaggio di direttive che vengono date dal server al client che impongono che non venga effettuata il caching.
Un ulteriore meccanismo di etag(?) ella cache, ella quale viene associato un hash del file. Il fatto che il file risulti modificato si verifica e si nota dall'hash che cambia.

2.1.2 Parte pratica

```c
#include<stdio.h>
#include <string.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <errno.h>
#include <arpa/inet.h>
#include <stdint.h>
#include <unistd.h>

struct sockaddr_in local, remote;
char request[1000001];
char response[1000];

struct header {
  char * n;
  char * v;
} h[100];

```

```
19
20 int main()
21 {
22 char hbuffer[10000];
23 char * reqline;
24 char * method, *url, *ver;
25 char * filename;
26 FILE * fin;
27 int c;
28 int n;
29 int i,j,t, s,s2;
30 int yes = 1;
31 int len;
32 if (( s = socket(AF_INET, SOCK_STREAM, 0 )) == -1)
33         { printf("errno = %d\n",errno); perror("Socket Fallita"); return -1; }
34 local.sin_family = AF_INET;
35 local.sin_port = htons(17999);
36 local.sin_addr.s_addr = 0;
37
38 t= setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
39 if (t==-1){perror("setsockopt fallita"); return 1;}
40
41 if ( -1 == bind(s, (struct sockaddr *)&local,sizeof(struct sockaddr_in)))
42 { perror("Bind Fallita"); return -1;}
43 if ( -1 == listen(s,10)) { perror("Listen Fallita"); return -1;}
44 remote.sin_family = AF_INET;
45 remote.sin_port = htons(0);
46 remote.sin_addr.s_addr = 0;
47 len = sizeof(struct sockaddr_in);
48 while ( 1 ){
49         s2=accept(s,(struct sockaddr *)&remote,&len);
50 bzero(hbuffer,10000);
51 bzero(h,sizeof(struct header)*100);
52 reqline = h[0].n = hbuffer;
53 for (i=0,j=0; read(s2,hbuffer+i,1); i++) {
54   if(hbuffer[i]=='\n' && hbuffer[i-1]=='\r'){
55     hbuffer[i-1]=0; // Termino il token attuale
56    if (!h[j].n[0]) break;
57    h[++j].n=hbuffer+i+1;
58   }
59   if (hbuffer[i]==':' && !h[j].v){
60     hbuffer[i]=0;
61     h[j].v = hbuffer + i + 1;
62   }
63  }
64
65         printf("%s\n",reqline);
66         if(len == -1) { perror("Read Fallita"); return -1;}
67         method = reqline;
68         for(i=0;reqline[i]!=' ';i++); reqline[i++]=0;
69         url=reqline+i;
70         for(; reqline[i]!=' ';i++); reqline[i++]=0;
71         ver=reqline+i;
72         for(; reqline[i]!=0;i++); reqline[i++]=0;
73         if ( !strcmp(method,"GET")){
74                 filename = url+1;
75                 fin=fopen(filename,"rt");
76                 if (fin == NULL){
77                         sprintf(response,"HTTP/1.1 404 Not Found\r\n\r\n");
78                         write(s2,response,strlen(response));
79                         }
80                 else{
81                         sprintf(response,"HTTP/1.1 200 OK\r\n\r\n");
82                         write(s2,response,strlen(response));
83                         while ( (c = fgetc(fin))!=EOF) write(s2,&c,1);
84                         fclose(fin);
85                         }
86 }
87         else {
88                         sprintf(response,"HTTP/1.1 501 Not Implemented\r\n\r\n");
89         write(s2,response,strlen(response));
90         }
91         close(s2);
92 }
```

```
93 close(s);
```

Molte elementi che costituiscono il web server sono identici a elementi utilizzati nel client web 1.1.
La prima differenza che notiamo è una secconda struct sockaddr_in che verrà utilizzata per inizzializzare l'indirizzo locale oltre che all'indirizzo remoto.
Possiamo notare anche altre variabili come una stringa (char *) che compone la requestline, ed altre tre stringhe (sempre char*) che conterranno il metodo, l'URL e la versione HTTP della richiesta fatta dall'utente che vuole accedere al nostro server.
Il primo socket che inizzializziamo risulta essere il socket che verrà reso passivo. L'inizializzazione del socket è del tutto analoga a quella nelle versioni del client web.
Ora bisogna passare ad inizzializzare l'indirizzo locale che vogliamo che faccia parte della famiglia AF_INET e che la porta che utilizziamo sia quella che abbiamo scelto noi(oviamente in big endian). L'indirizzo che usiamo per l'indirizzo locale è lo 0.
Non ci resta ora che impostare il sockopt e memorizzarne il valore all'interno di una variabile. Possiamo ora passare alla funzione di bind e di listen in modo che il nostro primo socket sia effettivamente diventato passivo.
Si può, quindi, inizzializzare l'indirizzo remoto, che farà parte della famiglia AF_INET, con port 0 e indirizzo 0. La variabile len farà riferimento alla dimensione di una struct sockaddr_in generica.
Dopo aver effettuato tutto questo processo di inizializzazione possiamo passare al ciclo di richieste che dovranno essere gestite dal nostro server web.
Innanzitutto, dobbiamo fare in modo che creiamo un socket che possa comunicare con il client che ha appena effettuato la richiesta. Possiamo fare in modo che questo socket venga creato attraverso l'uso della funzione accept, che prende come parametri il socket passivo, l'indirizzo remoto e la grandezza della struttura indirizzo utilizzatao. Effettuato ciò abbiamo tutti gli strumenti necessari per gestire le richieste del client e per rispondere in maniera adeguata.
Dato che ad ogni richiesta del client riceviamo nuovi header, la scelta migliore è quella di pulire la struttura h e l'array hbuffer, in modo che non ci siano errori in richieste successive.
Come sappiamo, la requestline è sempre la prima parte degli header, per cui possiamo farla puntare al nome del primo elemento degli header, che a sua volta punta al primo elemento del buffer degli header. Un volta eseguito questo setup, possiamo passare al parseing degli header, come già visto all'interno del client web 1.1.
Come dettaglio minore, si può voler stampare la requestline che si è ricevuta, diciamo più come aiuto a noi programmatori che all'utente finale.
Successivamente dobbiamo effettuare la divisione della requestline in modo da aver separati nelle proprie variabili il metodo, l'URL e la versione http utilizzate nella richiesta del client. Per fare ciò, controlliamo semplicemente gli spazi presenti all'interno della request line. Ogni volta che troviamo uno spazio significa che abbiamo trovato uno dei 3 elementi che compongono la requestline, per cui possiamo mettere uno 0 al posto dello spazio e far puntare la nostra variabile al pezzo di stringa che compone la request stessa. Ripetiamo questo passaggio fintantoché non possediamo tutti e 3 gli elementi che ci interessano.
Dato che la nostra request da parte del client sarà una GET /filename, sappiamo che il filename sarà l'URI senza /, per cui possiamo creare una stringa filename che punta a un carattere dopo l'URL.
Cerchiamo ora di aprire il contenuto del file che il client a richiest. Se il file ritorna valore nullo, mandiamo indietro un errore 404 file not found, altrimenti mandiamo indietro al client un 200 ok, leggiamo tutto il contenuto del file e lo spediamo al nostro client.
Data la natura del nostro server, per ora se una qualsiasi request non è una semplice GET verrà restituito all'utente l'errore 501 Not Implemented.
Una volta finita l'interpretazione del method chiudiamo il socket che ha effettuato l'accept, se il ciclo conclude verrà chiuso anche il socket passivo e verrà terminato il programma.

2.2 Server web con Authentication

2.2.1 Authentication

Quando il client richiede una risorsa il server controlla che quest'ultimo possa accedere ad una risorsa. Se l'utente trova una risorsa con il lucchetto il server risponde con un messaggio http/1.1 unauthorized ed un messaggio www.authenticate:basic (Vedi figura 2.7). Dato che il server può avere un insieme di utenti e password diverse associati ad una stessa risorsa, tipo servizi diversi o similari, viene indicato anche l'insieme di utenti alla quale quella risorsa fà riferimento. Si usa allora anche il realm = name. Si aprirà una finestra nell'user agent con due campo: utente e password con un tasto ok (Dialog del browser).
ç'utenete e password devono essere codificati in base64. Dato che l'header authorization viene mandato sempre nasce il primo concetto di cookie (Vedi figura 2.8). La basic authentication è una semplice login con user e password che permette o meno di far accede un utente ad un contenuto. Dopo che avviene l'autenticazione, il client continua, a ogni richiesta

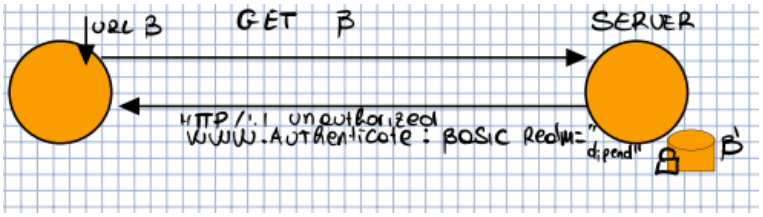


Figure 2.7: Richiesta di un file con authentication

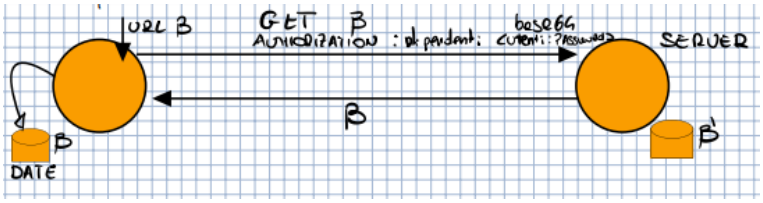


Figure 2.8: Richiesta di un file con authentication più in dettaglio

successiva, ad aggiungere la authentication basic, che è un header.
Una sessione è un via vai di richiesta-risposta memoryless. Ogni richiesta è a sè stante e ogni risposta è legata ad una specifica richiesta.
Bisogna fare attenzione a non confondere l'identità con la sessione, in quanto il numero della sessione cambia di sessione in sessione e quini un numero associato a un utente in una sessione può essere riassegnato a un altro utente in una nuova sessione.

Cookie

Il cookie è un elemento di stato, di mantenimento di memoria, che sopravvive tra un elemento di risposta e un altro. Il cookie è fornito dal server e il client lo utilizza ad ogni richiesta. Se è gia presente all'interno della richiesta del client, il server non lo fornisce.
Viene generato casualmente dal serve e deve essere diverso da quello di tutti gli altri.

2.2.2 Base64

Il base64 è un concetto base dello scambio di dati. Nasce dall'esigenza delle attachment nelle e-mail. L'intuizione che si possano mappare 3 byte in 4 celle da 6 bit. Ci possono essere casi in cui ho solo 2 blocchi da 8 byte, per cui otterrò 2 coppie da 6 bit più una quarta che conterrà 4bit con l'aggiunta di 2 bit a 0 per finire il riempimento.
Nel caso in cui viene mandato un solo byte, si avrà un blocco completo da 6 bit e un secondo blocco da 6con 2 bit e gli ultimi quattro riempiti da 0.
I blocchi di 6 bit che sono stati tagliati vengono rimpiazzati da un =, quindi nel primo caso si avranno 3 cifre seguite da un =, mentre nel secondo caso 2 cifre seguite da due =.

2.2.3 Parte pratica

```c
#include <stdio.h>
#include <string.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <errno.h>
#include <arpa/inet.h>
#include <stdint.h>
#include <unistd.h>

struct sockaddr_in local, remote;
char request[1000001];
char response[1000];

struct header {
        char * n;
        char * v;
```

```
} h[100];

char transform(char con){
        if(con<=25) return (65 + con);
        else if(con>25 && con<=51) return (97 + (con - 26));
        else if(con>51 && con<=61) return (48 + (con - 52));
        else if(con==62) return '+';
        else if(con==63) return '/';
}
char* encode(char* s){
        int ind=0;
        int res=1;
        int flag=0;
        char str[1000];
        char* ans=(char*)malloc(100);
        int l=0;
        int len=0;
        int pos=0;
        while(*(s+l)!=0){
                len++;
                l++;
        }
        int newLen=0;
        while(newLen<len){
                str[newLen]=*(s+newLen);
                newLen++;
        }
 int delta=0;
        while(newLen%3!=0){
                str[newLen]=0;
                delta++;
                newLen++;
        }
        while(ind<newLen){
                if(ind%3==0){
                        res=str[ind] >> 2;
                        ind++;
                }
                else if(ind%3==1){
                        res=((str[ind-1] & 0x03) << 4) + (str[ind] >> 4);        //0000 0011
                        ind++;
                }
                else if(ind%3==2 && !flag){
                        res=((str[ind-1] & 0x0F) << 2) + (str[ind] >> 6);        //0000 1111
                        flag=1;
                }
                else if(flag){
                        res=str[ind] & 0x3F;                                     //0011 1111
                        flag=0;
                        ind++;
                }
                ans[pos]=transform((char)res);
                if(res==0)
                        ans[pos]='=';
                pos++;
        }
        return ans;
}

int main() {
char hbuffer[10000];
char *reqline, *method, *url, *ver, *filename;
FILE *fin;
int c, n, i, j, t, s, s2, len;
int found = 0;
int prompt_flag = 0;
int yes = 1;
if (( s = socket(AF_INET, SOCK_STREAM, 0 )) == -1){
        printf("errno = %d\n",errno);
        perror("Socket Fallita");
 return -1;
}
local.sin_family = AF_INET;
local.sin_port = htons(17997);
```

```
local.sin_addr.s_addr = 0;

t= setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
if (t==-1){
        perror("setsockopt fallita");
        return 1;
}

if ( -1 == bind(s, (struct sockaddr *)&local, sizeof(struct sockaddr_in))){
        perror("Bind Fallita");
        return -1;
}

if ( -1 == listen(s,10)){
        perror("Listen Fallita");
        return -1;
}

remote.sin_family = AF_INET;
remote.sin_port = htons(0);
remote.sin_addr.s_addr = 0;

len = sizeof(struct sockaddr_in);

while (1){
        s2 = accept(s,(struct sockaddr *)&remote,&len);
        bzero(hbuffer,10000);
        reqline = h[0].n = hbuffer;
        for (i=0,j=0; len = read(s2, hbuffer+i, 1); i++){
                if(hbuffer[i]=='\n' && hbuffer[i-1]=='\r'){
                        hbuffer[i-1]=0; // Termino il token attuale
                        if (!h[j].n[0])
                                break;
                        h[++j].n=hbuffer + i + 1;
                }
                if (hbuffer[i]==':' && !h[j].v){
                        hbuffer[i]=0;
                        h[j].v = hbuffer + i + 1;
                }
 }
        printf("%s\n",reqline);

        if(len == -1){
                perror("Read Fallita");
                return -1;
        }

        len = strlen(reqline);
        method = reqline;

        for(i=0; i<len && reqline[i]!=' '; i++); reqline[i++]=0;
        url = reqline+i;

        for(; i<len && reqline[i]!=' '; i++); reqline[i++]=0;
        ver = reqline+i;

        for(;i<len && reqline[i]!='\r';i++); reqline[i++]=0;

        if(!strcmp(method, "GET")){
                filename = url+1;
                fin = fopen(filename, "rt");
                if(fin == NULL){
                        sprintf(response,"HTTP/1.1 404 Not Found\r\n\r\n");
                        write(s2,response,strlen(response));
                }
                if(!strncmp(filename,"secure/",7)){
                        char* temp_auth;
                        for (int i = 0; i < j; i++){
                                if(!strncmp(h[i].n,"Authorization",13)){
                                        found = 1;
                                        temp_auth = h[i].n + 21;//13 di authentication 8 di :,spazio,
    Basic e spazio
                                        printf("%s\n",temp_auth);
                                }
```

```
                        }
                        if (found == 0){
                                sprintf(response,"HTTP/1.1 401 Authorization Required\r\nWWW-
    Authenticate: Basic realm=\"protected\"\r\n\r\n");
                                write(s2,response,strlen(response));
                        }
                        else{
                                char login[100];
                                int temp_counter = 0;
                                FILE* log_file = fopen("login.txt","rt");
while((c = fgetc(log_file))!=EOF){
                                        login[temp_counter] = c;
                                        if(c == '\n'){
                                                login[temp_counter] = 0;
                                                temp_counter = 0;
                                                for(i=0; i<100 && login[i]!= ' '; i++); login[i]=':';
                                                        printf("%s\n",login);
                                                        if(!strcmp(encode(login),temp_auth)){
                                                        printf("trovato laser\n");
                                                        sprintf(response, "HTTP/1.1 200 OK\r\n\r\n");
                                                        write(s2, response, strlen(response));
                                                        while ((c = fgetc(fin))!=EOF) write(s2,&c,1);
                                                        fclose(fin);
                                                        prompt_flag = 1;
                                                }
                                        }
                                        else{
                                                temp_counter++;
                                        }
                                }
                                fclose(log_file);
                                if (prompt_flag == 0){
                                        sprintf(response,"HTTP/1.1 401 Authorization Required\r\nWWW-
    Authenticate: Basic realm=\"protected\"\r\n\r\n");
                                        write(s2,response,strlen(response));
                                }
                                prompt_flag = 0;
                        }
                        found = 0;
                }
                else{
                        sprintf(response, "HTTP/1.1 200 OK\r\n\r\n");
                        write(s2, response, strlen(response));
                        while ((c = fgetc(fin))!=EOF) write(s2,&c,1);
                        fclose(fin);
                }
        }
        else{
                sprintf(response,"HTTP/1.1 501 Not Implemented\r\n\r\n");
                write(s2,response,strlen(response));
        }
        close(s2);
}
close(s);
}
```

2.3 URI e variabili di ENV

2.3.1 Parte teorica

Come tutte le cose, anche l'URI ha una grammatica propria. Nasce dalla necessità di unire URL (e.g. www.xyz.com) con il path (/path).
L'absolute URI è una struttura generica, composta da un scheme : (uchar — reserved).
Il relative uri è composto da: (netpath — absolutepath — relativepath).
L'url è un caso particolare di uri, dove http è lo scheme, viene seguito dai ":" e successivamente il netpath risulata essere www.zyx.com/path.
www.xyz è il net locator, mentre /path è l'absolute path (che non è altro che un / seguito da un relative path (path/filename)).
L'uri in realtà può avere qualunque forma, senza vincoli. Se lo specializzo mi trovo in una caso specifico. L'absolute uri è

il più generico, il relative uri è un qualcosa on sheme e path, mentre l'url è ciò che contiene anche http.

2.3.2 Parte pratica

Vediamo ora un esempio di codice che ci permette di accedere alle variabili di ambiente.

```c
#include<stdio.h>
int main(int argc, char *argv[], char *env[])
{
int i;
for(i=0;i<argc;i++)
                printf("argv[%d]: %s\n",i,argv[i]);
for(i=0;env[i];i++)
                printf("env[%d]: %s\n",i,env[i]);
return 0;
}
```

Il programma deve sapere l'ambiente in cui sta girando, difatto quando stampiamo il vettore env abbiamo anche la pwd dove stiamo lavorando e tutte le altre informazioni che ci possono risultare utili.

2.4 CGI: Common Gateway Interface

2.4.1 Parte teorica

Query

Una query è tutto ciò che nella ricerca su browser, per esempio, segue il ?. Il browser deve interpretare tutto ciò che arriva dopo il GET come programma e non come filename.
Nasce così la CGI: la common gateway interface. Noi useremo la CGIBin, ovvero la CGI connessa ad un eseguibile, ovvero un file binario. Dal punto di vista di efficienza è il migliore, ma dato che bisogna dare accesso ad un programma binario a tutto, rende molto facile la possibilità di attacchi hacker.
Dato che deve essere eseguito un file, bisogna passare dei parametri al CGI.
Il web server prenderà i valori di queste variabili in parte dagli header http, in parte dall'url e in parte dalle varaibili interne. I parametri sono passati come enviroment al binario che viene chiamato dal web server (Vedi figura 2.9). Dopo

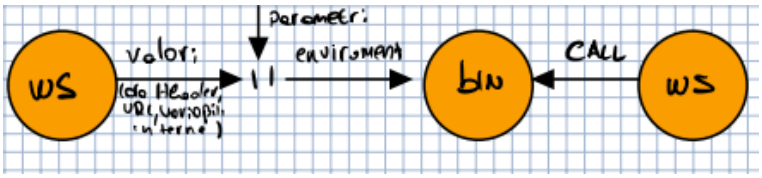


Figure 2.9: Modello concettuale del CGI

aver chiamato il programma, lo standard input viene collgato al request message body che viene passato in standard input al programma. Ma come fa il programma a sapere che è uno standard input? Ci sarà un parametro con il content length. L'entity body può arrivare tutto assieme o chunked. Nel secondo caso, il server deve aspettare tutto il corpo del

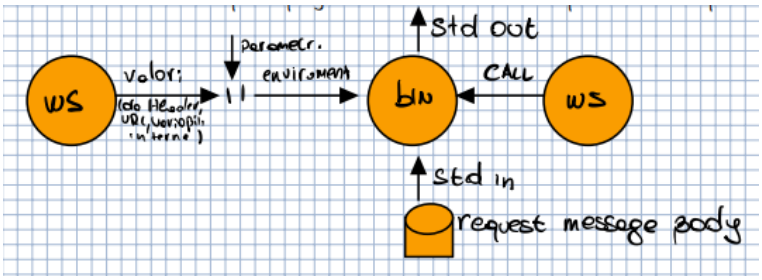


Figure 2.10: Modello concettuale del CGI con request body

programma per poi mandarlo. Si passa poi a mandare il tutto come standard output.
Grazie a questo modello, il web server diventa un gateway applicativo. Risponde alla richiesta http e ha anche un ambiente di esecuzione applicativo. Gli elementi del cgi accetta parametri che il web server conosce, tra i più importanti la query

string. In realtà i parametri dovrebbero essere passati tramite una variabile di enviroment, e questo parametro è unico, la query string. Si passano altri parametri, ma abbiamo visto che se la richiesta fatta dal client contiene anche un entity body nella richiesta, questo viene passato in standard input al programma.

2.4.2 Parte pratica

2.4.3 Server web CGI

```
#include<stdio.h>
#include<stdlib.h>
#include <string.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <errno.h>
#include <arpa/inet.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>


struct sockaddr_in local, remote;
char request[100000];
char response[1000];

struct header {
  char * n;
  char * v;
} h[100];

unsigned char  envbuf[1000];
int pid;
int env_i, env_c;
char * env[100];
int new_stdin, new_stdout;
char * myargv[10];

void add_env(char * env_key, char* env_value){
                sprintf(envbuf+env_c,"%s=%s",env_key,env_value);
                env[env_i++]=envbuf+env_c;
                env_c+=(strlen(env_value)+strlen(env_key)+2);
                env[env_i]=NULL;
}


int main()
{
char hbuffer[10000];
char * reqline;
char * method, *url, *ver;
char * filename,*content_type;
char fullname[200];
FILE * fin;
int c;
int n;
int i,j,t, s,s2;
int yes = 1;
int len;
int length;
if (( s = socket(AF_INET, SOCK_STREAM, 0 )) == -1)
        { printf("errno = %d\n",errno); perror("Socket Fallita"); return -1; }
local.sin_family = AF_INET;
local.sin_port = htons(17999);
local.sin_addr.s_addr = 0;

t= setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
if (t==-1){perror("setsockopt fallita"); return 1;}

if ( -1 == bind(s, (struct sockaddr *)&local,sizeof(struct sockaddr_in)))
{ perror("Bind Fallita"); return -1;}

if ( -1 == listen(s,10)) { perror("Listen Fallita"); return -1;}
```

```
remote.sin_family = AF_INET;
remote.sin_port = htons(0);
remote.sin_addr.s_addr = 0;
len = sizeof(struct sockaddr_in);
while ( 1 ){
        s2=accept(s,(struct sockaddr *)&remote,&len);
bzero(hbuffer,10000);
bzero(h,sizeof(struct header)*100);
reqline = h[0].n = hbuffer;
for (i=0,j=0; read(s2,hbuffer+i,1); i++) {
  if(hbuffer[i]=='\n' && hbuffer[i-1]=='\r'){
    hbuffer[i-1]=0; // Termino il token attuale
   if (!h[j].n[0]) break;
   h[++j].n=hbuffer+i+1;
  }
  if (hbuffer[i]==':' && !h[j].v){
    hbuffer[i]=0;
    h[j].v = hbuffer + i + 1;
  }
 }
length=0;
for(i=0;i<j;i++){
 printf("%s ---> %s\n",h[i].n,h[i].v);
        if(!strcmp(h[i].n,"Content-Length")){
                length=atoi(h[i].v);
                }

        if(!strcmp(h[i].n,"Content-Type")){
                add_env("CONTENT_TYPE",h[i].v+1);
        }
        }
        len=1000;
        printf("%s\n",reqline);
        if(len == -1) { perror("Read Fallita"); return -1;}
        method = reqline;
        len=1000;
        for(i=0;i<len && reqline[i]!=' ';i++); reqline[i++]=0;
        url=reqline+i;
        for(;i<len && reqline[i]!=' ';i++); reqline[i++]=0;
        ver=reqline+i;
        for(;i<len && reqline[i]!='\r';i++); reqline[i++]=0;
        add_env("METHOD",method);
        filename = url+1;
        if (!strncmp(url,"/cgi/",5)){ //CGI
                filename=url+5;
                if (!strcmp(method,"GET")){
                for(i=0;filename[i] && (filename[i]!='?');i++);
                        if(filename[i]=='?'){
                                filename[i]=0;
                                add_env("QUERY_STRING",filename+i+1);
                        }
                        add_env("CONTENT_LENGTH","0");
                }
                else if(!strcmp(method,"POST")){
                        char tmp[10];
                        sprintf(tmp,"%d",length);
                        add_env("CONTENT_LENGTH",tmp);
                }
                else {
                        sprintf(response,"HTTP/1.1 501 Not Implemented\r\n\r\n");
        write(s2,response,strlen(response));
                        close(s2);
                        continue;
                }
                fin=fopen(filename,"rt");
if (fin == NULL){
                        sprintf(response,"HTTP/1.1 404 Not Found\r\n\r\n");
                        write(s2,response,strlen(response));
                }
                else{
                        sprintf(response,"HTTP/1.1 200 OK\r\n\r\n");
                        write(s2,response,strlen(response));
                        fclose(fin);
                        for(i=0;env[i];i++)
```

```
                                printf("environment: %s\n",env[i]);
                        sprintf(fullname,"/RDC22/%s",filename);
                        myargv[0]=fullname;
                        myargv[1]=NULL;
                        printf("Executing %s\n",fullname);
                        if(!(pid=fork())){
                                dup2(s2,1);
                                 dup2(s2,0);
                                  if(-1==execve(fullname,myargv,env))
                                                                                { perror("
    execve"); exit(1);}

                                                        }
                        waitpid(pid,NULL,0);
                        printf("Il figlio e' morto...\n");
                        }

        }
        else if ( !strcmp(method,"GET")){ //NOT CGI
                filename = url+1;
                printf("filename: %s\n",filename);
                fin=fopen(filename,"rt");
                if (fin == NULL){
                        sprintf(response,"HTTP/1.1 404 Not Found\r\n\r\n");
                        write(s2,response,strlen(response));
                        }
                else{
                        sprintf(response,"HTTP/1.1 200 OK\r\n\r\n");
                        write(s2,response,strlen(response));
                        while ( (c = fgetc(fin))!=EOF) write(s2,&c,1);
                        fclose(fin);
                        }
        }
else {
                        sprintf(response,"HTTP/1.1 501 Not Implemented\r\n\r\n");
write(s2,response,strlen(response));
        }
        close(s2);
        env_c=env_i=0;
}
close(s);
}
```

2.5 DNS

Il DNS, ovvero domain name system, parte dall'esigenza di risolvere i nomi in indirizzi IP, ad esempio www.dei.unipd.it risolto nell'indirizzo 147.162.2.100.
Non c'è nessuna relazione tra i pezzi del nome e i pezzi dell'indirizzo. Dei non è 163 così come 2 non è unipd. Il nome completo deve essere risolto ad indirizzo e nient'altro. Il computer che stanno nelle diverse reti, devono essere mantenuti

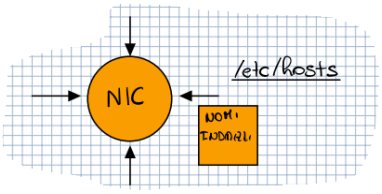


Figure 2.11: Figura

da diverse organizzazioni e le reti devono essere gestite da un'organizzazione. Queste organizzazioni esistono e le vediamo con una figura cerarchica (Figura 2.12).
Come le cartelle che sono una guida che ci fa fare un percorso fino al file, il nome è come se fosse un path, con il percorso fatto cosi che dato un pezzo di nome noi sappiamo a chi chiedere il successivo.
Il DNS è un database di path con tipi, cioè i record hanno dei tipi, ed i tipi fondamentali sono il tipo A, ovvero address, e NS, ovvero name server.
Ogni dominio, ad esempio it, ha un name server che ci può dare infromazioni u se stesso e informazioni sul name server del dominio sottostante. Ogni dominio ha il suo web server ch noi possiamo interrogare che noi possiamo interrogare. Il

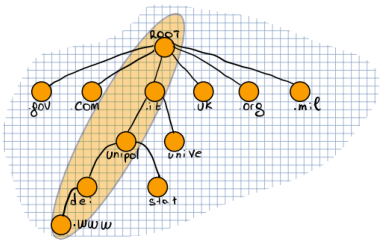


Figure 2.12: Rappresentazione gerarchica dei nomi

name server sarà il nome del name server che può essere risolto.
Il dominio it avrà tantissime informazioni relative ai domini sottostanti, in quanto tutti i domini che niniscono con .it sono molteplici.
Ogni dominio aggiunge una entry per il nuovo computer, sito o qualsivoglia applicativo, al loro database personale.

2.5.1 Nomi

Rappresentano un servizio infrastrutturale della rete internet che è stato progettato bene dall'inizio. La conversioni dei nomi tipo www.qualcosa.it, in numero, ovvero l'indirizzo ip, è un qualcosa che si usa continuamente.Non esiste applicazione che non faccia uso dei nomi.

2.5.2 Risolvere un nome

Normalmente una macchina ha un nameserver configurato.
Proviamo ora a risolvere www.dei.unipd.it usando il comando dig . +norecurse NS. Sto chiedendo al mio name server quali sono i name server di root, informazione ottenibile in quanto c'è un file di sistema del server che ha già questi nomi. Questi nomi sono chiesti dal nostro server e sono 13 perchè non ce ne starebbero di più in una richiesta DNS.
Notiamo un authority:0, ovvero chi mi da questi nomi non sono chi li mantiene, altrimenti chi li modificasse manderebbe a puttane tutto per tutti. C'è ne sono 13 così che se uno non funziona c'è ne un altro e così via. Proviamo ora:
dig c.root-servers.net +norecurse NS
Mi da come risultato niente in quanto non c'è niente al di sopra del root.
Un client può interrogare e fare la prima, seconda richiesta, ottenendo sempre il name server della zona sottostante, sino ad arrivare al name server che gestisce il nome della foglia, che è www.
Un fattore importante è che tutti i server chiedono i nameserver al root.
Le informazioni dei nomi sono sparsi da tanti server, ciasuno chiamato name server e porta le informazioni relative al dominio identificato dal nome che otteniamo percorrendo l'albero dala root fino ad un certo punto.
Il root server non permette di fare richieste ricorsive e i glue record no sono nella zona di memoria che li forniscono.

Glue record

Un glue record è un indirizzo ip che viene incollato ad un dominio o a un sotto dominio e che viene quindi memorizzato nella zeona dell'ente responsabile per la registrazione dei domini.

Robe a caso

La cache ha una scadenza di 12 secondi, CNAME indica il canonical name e MX, invece, indica il mail exchange.

Flags

I flag che abbiamo visto sono:

- qr: query ready.
- ra: recursion allowed.
- ad: authentication date.
- aa: authorization answer.

Chapter 3

Proxy web

3.1 Parte teorica

Il proxy fa da intermediario tra client e server, simulando ancora un risultato come se il client si fosse collegato direttamente al server. Il client viene configurato per usare il proxy, il server riceverà le richieste dal proxy.
Il primo scenario per la quale si usa il proxy è quando si hanno le reti private, perchè non si vuole esporre la propria rete all'esterno.
Il primo esempio di utilizzo del proxy è un modo per consentire a dei client che fanno parte di una rete privata, concetto

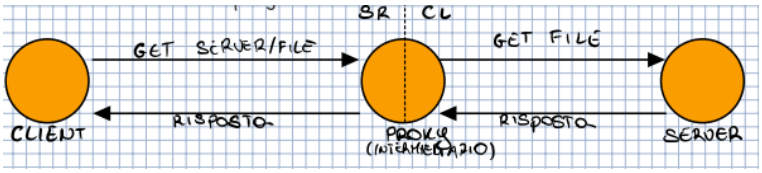


Figure 3.1: Schema modello client server con il proxy

di livello 3, di essere filtrati e non essere esposti al mondo esterno.
Ci sono 2 connessioni http per ogni richiesta, una tra client e proxy, e una tra proxy e server. La prima sarà tra due indirizzi privati, mentre l'altra tra un indirizzo privato e uno pubblico.
Il proxy si usa spesso perchè può vedere le richieste e i contenuti che passano tra il client e server, vede richieste applicative, header, uri e, dato che lavora al livello 7 (applicativo), può capire i contenuti che passano attraverso. Questo è un valore aggiunto nel momento in cui si vuole sorvegliare l'utilizzo di un servizio web.
Il proxy può fare da filtro in modo da fargli scartare degli uri o delle keyword, filtrando così determinate richieste e bloccando eventuali interazioni col server. Posso fare lo stesso analogamente per le risposte che il client riceve. Potrebbe aiutare a prevenire attacchi del tipo query injection, ovvero attacchi che mirano ad inniettare query SQL così da far fare azioni anomale al nostro dispositivo.
L'HTTPS scambia una chiave crittografica prima della prima richiesta, in modo che solo chi scambia i dati può interpretare correttamente i dati.
L'unico modo per evitare che uno possa vedere il traffico che passa, tipo password o codici delle carte di credito, deve fare in modo che chi trasmette e chi riceve abbiano la possibilità di avere un codice condiviso con la quale sia possibile criptare o decriptare i dati.
Sia il server che il client possiedono una propria chiave, dalla quale si possono generare due mezze chiavi a loro volta. Se il client manda una metà della chiave ed il server manda una metà della sua chiave, più precisamente la metà nella stessa posizione della mezza chiave del client (se il client manda la prima metà della propria allora anche il server manda la prima metà), è possibile derivare matematicamente dalle coppie di mezze chiavi diverse una chiave unica, utilizzabile da entrambi sia per leggere che per scrivere i dati. Vengono scambiate le stesse metà in quanto, anche se un utente malevolo le riesca a ricavare, esso non può ricavare nulla da esse.
Ma allora il proxy cosa può fare? Abbiamo due scenari:

- Fidarsi del proxy, creando un canale sicuro tra client e proxy e tra proxy e server. Ovviamente lascia un punto di vulnerabilità all'interno del proxy. L'amministratore del proxy può violare tutto. Con questa soluzione, si riescono a supportare tutte le applicazioni del proxy elencate precedentemente.

- il tunneling: un metodo che prende il nome di connect, che prende un ip e un port come argomento ("ip" : "port"). Questo accade quando il client non si fida del proxy, rendendo così il proxy incapace di analizzare i dati che transitano

tramite esso. Viene usato quando si vuole un canale sicuro end-to-end tra client e server. Comporta due connessioni tcp normali e le chiavi vengono create solo tra client e server, mentre il proxy sarà solo un connettore tra i due punti. Il proxy, dunque, vedrebbe solo transitare byte, senza avere la possibilità di saperli interpretare. Il client effettuerà una connect e il server la stabilira (established).

Vediamo ora un ultimo schema del modello client server con il proxy:

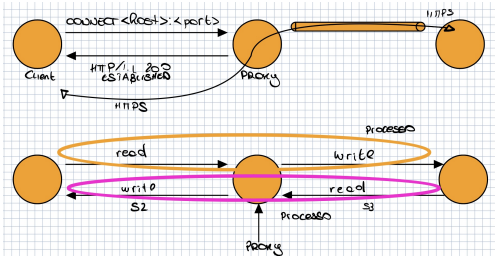


Figure 3.2: Schema modello client server con il proxy 2

3.2 Parte pratica

3.2.1 Pw.c

```c
#include<stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <errno.h>
#include <arpa/inet.h>
#include <stdint.h>
#include <unistd.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <signal.h>

int pid;
struct sockaddr_in local, remote, server;
char request[10000];
char request2[10000];
char response[1000];
char response2[10000];

struct header {
  char * n;
  char * v;
} h[100];

struct hostent * he;

int main()
{
char hbuffer[10000];
char buffer[2000];
char * reqline;
char * method, *url, *ver, *scheme, *hostname, *port;
char * filename;
FILE * fin;
int c;
int n;
int i,j,t, s,s2,s3;
int yes = 1;
int len;
if (( s = socket(AF_INET, SOCK_STREAM, 0 )) == -1)
        { printf("errno = %d\n",errno); perror("Socket Fallita"); return -1; }
local.sin_family = AF_INET;
local.sin_port = htons(17999);
```

```
local.sin_addr.s_addr = 0;

t= setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
if (t==-1){perror("setsockopt fallita"); return 1;}

if ( -1 == bind(s, (struct sockaddr *)&local,sizeof(struct sockaddr_in)))
{ perror("Bind Fallita"); return -1;}

if ( -1 == listen(s,10)) { perror("Listen Fallita"); return -1;}
remote.sin_family = AF_INET;
remote.sin_port = htons(0);
remote.sin_addr.s_addr = 0;
len = sizeof(struct sockaddr_in);
while ( 1 ){
        s2=accept(s,(struct sockaddr *)&remote,&len);
printf("Remote address: %.8X\n",remote.sin_addr.s_addr);
if (fork()) continue;
if(s2 == -1){perror("Accept fallita"); exit(1);}
bzero(hbuffer,10000);
bzero(h,100*sizeof(struct header));
reqline = h[0].n = hbuffer;
for (i=0,j=0; read(s2,hbuffer+i,1); i++) {
        printf("%c",hbuffer[i]);
  if(hbuffer[i]=='\n' && hbuffer[i-1]=='\r'){
    hbuffer[i-1]=0; // Termino il token attuale
   if (!h[j].n[0]) break;
   h[++j].n=hbuffer+i+1;
  }
  if (hbuffer[i]==':' && !h[j].v && j>0){
    hbuffer[i]=0;
    h[j].v = hbuffer + i + 1;
  }
 }

        printf("Request line: %s\n",reqline);
        method = reqline;
        for(i=0;i<100 && reqline[i]!=' ';i++); reqline[i++]=0;
        url=reqline+i;
        for(;i<100 && reqline[i]!=' ';i++); reqline[i++]=0;
 ver=reqline+i;
        for(;i<100 && reqline[i]!='\r';i++); reqline[i++]=0;
        if ( !strcmp(method,"GET")){
                scheme=url;
                        // GET http://www.aaa.com/file/file
                printf("url=%s\n",url);
                for(i=0;url[i]!=':' && url[i] ;i++);
                if(url[i]==':') url[i++]=0;
                else {printf("Parse error, expected ':'"); exit(1);}
                if(url[i]!='/' || url[i+1] !='/')
                {printf("Parse error, expected '//'"); exit(1);}
                i=i+2; hostname=url+i;
                for(;url[i]!='/'&& url[i];i++);
                if(url[i]=='/') url[i++]=0;
                else {printf("Parse error, expected '/'"); exit(1);}
                filename = url+i;
                printf("Schema: %s, hostname: %s, filename: %s\n",scheme,hostname,filename);

                he = gethostbyname(hostname);
                printf("%d.%d.%d.%d\n",(unsigned char) he->h_addr[0],(unsigned char) he->h_addr[1],(
   unsigned char) he->h_addr[2],(unsigned char) he->h_addr[3]);
                if (( s3 = socket(AF_INET, SOCK_STREAM, 0 )) == -1)
                        { printf("errno = %d\n",errno); perror("Socket Fallita"); exit(-1); }

                server.sin_family = AF_INET;
                server.sin_port =htons(80);
                server.sin_addr.s_addr = *(unsigned int *)(he->h_addr);

                if(-1 == connect(s3,(struct sockaddr *) &server, sizeof(struct sockaddr_in)))
                                {perror("Connect Fallita"); exit(1);}
                sprintf(request,"GET /%s HTTP/1.1\r\nHost:%s\r\nConnection:close\r\n\r\n",filename,
   hostname);
                printf("%s\n",request);
                write(s3,request,strlen(request));
                while ( t=read(s3,buffer,2000))
```

```
                        write(s2,buffer,t);
                close(s3);
                }
        else if(!strcmp("CONNECT",method)) { // it is a connect  host:port
                hostname=url;
                for(i=0;url[i]!=':';i++); url[i]=0;
                port=url+i+1;
                printf("hostname:%s, port:%s\n",hostname,port);
                he = gethostbyname(hostname);
                if (he == NULL) { printf("Gethostbyname Fallita\n"); return 1;}
                printf("Connecting to address = %u.%u.%u.%u\n", (unsigned char ) he->h_addr[0],(unsigned char ) he->h_addr[1],(unsigned char ) he->h_addr[2],(unsigned char ) he->h_addr[3]);
                s3=socket(AF_INET,SOCK_STREAM,0);

                if(s3==-1){perror("Socket to server fallita"); return 1;}
                server.sin_family=AF_INET;
                server.sin_port=htons((unsigned short)atoi(port));
                server.sin_addr.s_addr=*(unsigned int*) he->h_addr;
                t=connect(s3,(struct sockaddr *)&server,sizeof(struct sockaddr_in));
                if(t==-1){perror("Connect to server fallita"); exit(0);}
                sprintf(response,"HTTP/1.1 200 Established\r\n\r\n");
                write(s2,response,strlen(response));
                        // <==============
                if(!(pid=fork())){ //Child
                        while(t=read(s2,request2,2000)){
                                write(s3,request2,t);
                        //printf("CL >>>(%d)%s \n",t,hostname); //SOLO PER CHECK
                                }
                        exit(0);
                        }
                else { //Parent
                        while(t=read(s3,response2,2000)){
                                write(s2,response2,t);
                        //printf("CL <<<(%d)%s \n",t,hostname);
                        }
                        kill(pid,SIGTERM);
                        close(s3);
                        }
                }
        else {
                        sprintf(response,"HTTP/1.1 501 Not Implemented\r\n\r\n");
        write(s2,response,strlen(response));
        }
        close(s2);
        exit(1);
}
close(s);
}
```

3.2.2 Pw1.c

```
#include<stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <errno.h>
#include <arpa/inet.h>
#include <stdint.h>
#include <unistd.h>
#include <netdb.h>
#include <netinet/in.h>



struct sockaddr_in local, remote, server;
char request[10000];
char response[1000];

struct header {
  char * n;
  char * v;
} h[100];
```

```
23
24 struct hostent * he;
25
26 int main()
27 {
28 char hbuffer[10000];
29 char buffer[2000];
30 char * reqline;
31 char * method, *url, *ver, *scheme, *hostname;
32 char * filename;
33 FILE * fin;
34 int c;
35 int n;
36 int i,j,t, s,s2,s3;
37 int yes = 1;
38 int len;
39 if (( s = socket(AF_INET, SOCK_STREAM, 0 )) == -1)
40         { printf("errno = %d\n",errno); perror("Socket Fallita"); return -1; }
41 local.sin_family = AF_INET;
42 local.sin_port = htons(17999);
43 local.sin_addr.s_addr = 0;
44 t= setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
45 if (t==-1){perror("setsockopt fallita"); return 1;}
46
47 if ( -1 == bind(s, (struct sockaddr *)&local,sizeof(struct sockaddr_in)))
48 { perror("Bind Fallita"); return -1;}
49
50 if ( -1 == listen(s,10)) { perror("Listen Fallita"); return -1;}
51 remote.sin_family = AF_INET;
52 remote.sin_port = htons(0);
53 remote.sin_addr.s_addr = 0;
54 len = sizeof(struct sockaddr_in);
55 while ( 1 ){
56         s2=accept(s,(struct sockaddr *)&remote,&len);
57 printf("Remote address: %.8X\n",remote.sin_addr.s_addr);
58 if (fork()) continue;
59 if(s2 == -1){perror("Accept fallita"); exit(1);}
60 bzero(hbuffer,10000);
61 bzero(h,100*sizeof(struct header));
62 reqline = h[0].n = hbuffer;
63 for (i=0,j=0; read(s2,hbuffer+i,1); i++) {
64         printf("%c",hbuffer[i]);
65   if(hbuffer[i]=='\n' && hbuffer[i-1]=='\r'){
66     hbuffer[i-1]=0; // Termino il token attuale
67    if (!h[j].n[0]) break;
68    h[++j].n=hbuffer+i+1;
69   }
70   if (hbuffer[i]==':' && !h[j].v && j>0){
71     hbuffer[i]=0;
72     h[j].v = hbuffer + i + 1;
73   }
74  }
75
76         printf("Request line: %s\n",reqline);
77         method = reqline;
78         for(i=0;i<100 && reqline[i]!=' ';i++); reqline[i++]=0;
79         url=reqline+i;
80         for(;i<100 && reqline[i]!=' ';i++); reqline[i++]=0;
81         ver=reqline+i;
82         for(;i<100 && reqline[i]!='\r';i++); reqline[i++]=0;
83         if ( !strcmp(method,"GET")){
84                 scheme=url;
85                         // GET http://www.aaa.com/file/file
86                 printf("url=%s\n",url);
87                 for(i=0;url[i]!=':' && url[i] ;i++);
88                 if(url[i]==':') url[i++]=0;
89                 else {printf("Parse error, expected ':'"); exit(1);}
90                 if(url[i]!='/' || url[i+1] !='/')
91                 {printf("Parse error, expected '//'"); exit(1);}
92                 i=i+2; hostname=url+i;
93                 for(;url[i]!='/'&& url[i];i++);
94                 if(url[i]=='/') url[i++]=0;
95                 else {printf("Parse error, expected '/'"); exit(1);}
96                 filename = url+i;
```

```
                printf("Schema: %s, hostname: %s, filename: %s\n",scheme,hostname,filename);

                he = gethostbyname(hostname);
                printf("%d.%d.%d.%d\n",(unsigned char) he->h_addr[0],(unsigned char) he->h_addr[1],(unsigned char) he->h_addr[2],(unsigned char) he->h_addr[3]);
                if (( s3 = socket(AF_INET, SOCK_STREAM, 0 )) == -1)
                        { printf("errno = %d\n",errno); perror("Socket Fallita"); exit(-1); }

                server.sin_family = AF_INET;
                server.sin_port =htons(80);
                server.sin_addr.s_addr = *(unsigned int *)(he->h_addr);

                if(-1 == connect(s3,(struct sockaddr *) &server, sizeof(struct sockaddr_in)))
                                {perror("Connect Fallita"); exit(1);}
                sprintf(request,"GET /%s HTTP/1.1\r\nHost:%s\r\nConnection:close\r\n\r\n",filename,hostname);
                printf("%s\n",request);
                write(s3,request,strlen(request));
                while ( t=read(s3,buffer,2000))
                        write(s2,buffer,t);
                close(s3);
                }
        else {
                        sprintf(response,"HTTP/1.1 501 Not Implemented\r\n\r\n");
        write(s2,response,strlen(response));
        }
        close(s2);
        exit(1);
}
close(s);
}
```

3.2.3 Proxy commentato

```
#include<stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <errno.h>
#include <arpa/inet.h>
#include <stdint.h>
#include <unistd.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <signal.h>

int pid;
struct sockaddr_in local, remote, server;
char request[10000];
char request2[10000];
char response[1000];
char response2[10000];

struct header {
char * n;
char * v;
} h[100];

struct hostent * he;

int main()
{
char hbuffer[10000];
char buffer[2000];
char * reqline;
char * method, *url, *ver, *scheme, *hostname, *port;
char * filename;
FILE * fin;
int c;
int n;
int i,j,t, s,s2,s3;
```

```
int yes = 1;
int len;
if (( s = socket(AF_INET, SOCK_STREAM, 0 )) == -1)
{ printf("errno = %d\n",errno); perror("Socket Fallita"); return -1; }
local.sin_family = AF_INET;
local.sin_port = htons(17999);
local.sin_addr.s_addr = 0;
t= setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
if (t==-1){perror("setsockopt fallita"); return 1;}
if ( -1 == bind(s, (struct sockaddr *)&local,sizeof(struct sockaddr_in)))
{ perror("Bind Fallita"); return -1;}
if ( -1 == listen(s,10)) { perror("Listen Fallita"); return -1;}
remote.sin_family = AF_INET;
remote.sin_port = htons(0);
remote.sin_addr.s_addr = 0;
len = sizeof(struct sockaddr_in);
while ( 1 ){ //classico ciclo
s2=accept(s,(struct sockaddr *)&remote,&len);//socket che si occupa delle richieste
printf("Remote address: %.8X\n",remote.sin_addr.s_addr);//stamp l'indirizzo in exa formattando tutto in
    maiuscolo
if (fork()) continue;//se e un genitore, ha figli, se fork() ritorna zero e un figlio senza figli.
    Verifico che sia un genitore.
if(s2 == -1){perror("Accept fallita"); exit(1);}//se fallisco la connessione con il socket, errore e
    termino la connessione.
bzero(hbuffer,10000);//classica iniziallizzazione.
bzero(h,100*sizeof(struct header));//same as 66
reqline = h[0].n = hbuffer;//parser e robe
for (i=0,j=0; read(s2,hbuffer+i,1); i++) {
printf("%c",hbuffer[i]);
if(hbuffer[i]=='\n' && hbuffer[i-1]=='\r'){
hbuffer[i-1]=0; // Termino il token attuale
if (!h[j].n[0]) break;
h[++j].n=hbuffer+i+1;
}
if (hbuffer[i]==':' && !h[j].v && j>0){
hbuffer[i]=0;
h[j].v = hbuffer + i + 1;
}
}
printf("Request line: %s\n",reqline);//classica gestione della request line.
method = reqline;
for(i=0;i<100 && reqline[i]!=' ';i++); reqline[i++]=0;
url=reqline+i;
for(;i<100 && reqline[i]!=' ';i++); reqline[i++]=0;
ver=reqline+i;
for(;i<100 && reqline[i]!='\r';i++); reqline[i++]=0;
if ( !strcmp(method,"GET")){//siamo al get method.
scheme=url;
// GET http://www.aaa.com/directory/file
printf("url=%s\n",url);//questo e il parseing dell'url http perche' e' un proxy brutto e puzzolente
    come giulio
for(i=0;url[i]!=':' && url[i] ;i++);
if(url[i]==':') url[i++]=0;
else {printf("Parse error, expected ':'"); exit(1);}
if(url[i]!='/' || url[i+1] !='/')
{printf("Parse error, expected '//'"); exit(1);}
i=i+2; hostname=url+i;
for(;url[i]!='/'&& url[i];i++);
if(url[i]=='/') url[i++]=0;
else {printf("Parse error, expected '/'"); exit(1);}
filename = url+i;
printf("Schema: %s, hostname: %s, filename: %s\n",scheme,hostname,filename);//Ho finito il parseing del
    link example di sopra.
he = gethostbyname(hostname);//converte il nome www.aaa.com in un indirizzo ip.
printf("%d.%d.%d.%d\n",(unsigned char) he->h_addr[0],(unsigned char) he->h_addr[1],(unsigned char) he->
    h_addr[2],(unsigned char) he->h_addr[3]);//qui sto stampando l'ip appena convertito. he e una
    struct prefatta e contiene(suppongo) un intero che essendo puntato da un unsigned char, mi permette
    di accedere ai singoli byte che lo compongono.
if (( s3 = socket(AF_INET, SOCK_STREAM, 0 )) == -1)//socket del proxy che comunichera con il server.
{ printf("errno = %d\n",errno); perror("Socket Fallita"); exit(-1); }//Errore se il proxy non puo
    comunicare con un cazzo di niente.
//Inizializzo l'indirizzo per connettersi al server effettivo.
server.sin_family = AF_INET;
server.sin_port =htons(80);
```

```
105 server.sin_addr.s_addr = *(unsigned int *)(he->h_addr);
106 //Conetto il socket al server effettivo, almeno ci proviamo.
107 if(-1 == connect(s3,(struct sockaddr *) &server, sizeof(struct sockaddr_in)))
108 {perror("Connect Fallita"); exit(1);}
109 sprintf(request,"GET /%s HTTP/1.1\r\nHost:%s\r\nConnection:close\r\n\r\n",filename,hostname);//Traduci
    la richiesta che era del client, scritta in maniera GET http://www.aaa.com/directory/file in una
    richiesta classica del tipo GET / HTTP/1.1/r/nHost:name/r/n/r/n.
110 printf("%s\n",request);//La stampo perche si, mi piace quando mi stampi roba.                write(
    s3,request,strlen(request));//Scrivo nel socket per il server.
111  while ( t=read(s3,buffer,2000))//Scrivo cio che mi manda il server e li scrivo indietro al proxy(
    client).
112 write(s2,buffer,t);//Scriviamo in s2 perche servira al proxy(server) per trasmetterlo indietro al vero
    client.
113 close(s3);//Posso chiudere tranquillamente s3 in quanto ho terminato il lavoro del proxy come un client
    .
114 }
115 else if(!strcmp("CONNECT",method)) { // it is a connect  host:port Ma che cazzo fa?Non ho fatto una GET
    , ma essendo un proxy ho il caso che il clien    t possa aver effettuata una connect.
116 hostname=url;//A questo punto, il proxy sta solo facendo da tunnel, sta stabilendo una connessione "
    diretta" tra il client e il server.
117 for(i=0;url[i]!=':';i++); url[i]=0;//Sto cavando via solo http: dalla request tipo che abbiamo visto
    sopra.
118 port=url+i+1;//Ho preso il port.
119 printf("hostname:%s, port:%s\n",hostname,port);
120 he = gethostbyname(hostname);//COnverto l'hostname in ip come nella GET case.
121 if (he == NULL) { printf("Gethostbyname Fallita\n"); return 1;}
122 printf("Connecting to address = %u.%u.%u.%u\n", (unsigned char ) he->h_addr[0],(unsigned char ) he->
    h_addr[1],(unsigned char ) he->h_addr[2]     ,(unsigned char ) he->h_addr[3]);
123 s3=socket(AF_INET,SOCK_STREAM,0);//Uso un nuovo socket come detto prima.
124 if(s3==-1){perror("Socket to server fallita"); return 1;}
125 server.sin_family=AF_INET;
126 server.sin_port=htons((unsigned short)atoi(port));
127 server.sin_addr.s_addr=*(unsigned int*) he->h_addr;
128 t=connect(s3,(struct sockaddr *)&server,sizeof(struct sockaddr_in));
129 if(t==-1){perror("Connect to server fallita"); exit(0);}
130 sprintf(response,"HTTP/1.1 200 Established\r\n\r\n");
131 write(s2,response,strlen(response));//Mando la risposta del server al client..
132 // <===============
133 if(!(pid=fork())){ //Se ci troviamo all'interno di un figlio, ovvero dato che la variabile PID si trova
     nell'area globale e zero, quindi se     fork ha ritornato 0.
134 while(t=read(s2,request2,2000)){//Leggo la richiesta del client e la mando al nostro server
    cazzutissimo.
135 write(s3,request2,t);
136 //printf("CL >>>(%d)%s \n",t,hostname); //SOLO PER CHECK
137 }
138 exit(0);//Questa e una exit success, almeno cosi dice il sommo Giovanni.
139 }
140 else { //Se invece ci troviamo all'interno di un nodo genitore
141 while(t=read(s3,response2,2000)){//Leggo la risposta del server e la mando indietro al client merdoso.
142 write(s2,response2,t);
143 //printf("CL <<<(%d)%s \n",t,hostname);
144 }
145 kill(pid,SIGTERM);//Eseguo una termination request sul pid designato, ovvero il figlio che stiamo
    considerando.
146 close(s3);//Chiudo il socket che comunica con il server effettivo.
147 }
148 }
149 else {//Classico not implemented
150 sprintf(response,"HTTP/1.1 501 Not Implemented\r\n\r\n");
151 write(s2,response,strlen(response));
152 }
153 close(s2);//Chiudo il socket attivo del proxy.
154 exit(1);//Exit failure termino la richiesta totale che un client ha fatto al socket.
155 }
156 close(s);//Chiudo anche il socket passivo, quindi tutto e finito.
157 }
```

Le cose più importanti che notiamo sono:

- Il proxy presenta 3 socket: i primi 2 sono dovuti al fatto che nei confronti del client, esso si stà comportando come un server, quindi 1 come socket passivo ed uno come socket di comunicazione. Il terzo è dovuto dal fatto che si stà comportando come un client nei confronti del server, è che quindi ha bisogno di un socket per effettuare richieste ad esso.

- Il proxy presenta un albero di bifocazioni, ognina dei quali attribuisce un id, il PID, ad ogni nodo dell'albero. Se il nodo è un nodo interno, il PID risulta essere ¿0. Se il nodo è una folgia, il PID risulta essere = 0. L'ultimo caso è se ci sono problemi a creare una biforcazione, Se ciò accade, il fork ritorna come valore -1.
- Il proxy può effettuare richieste multiple al server, per questo motivo ci sono i fork. Nel caso di una GET, il proxy deve rimodellar la natura della request line, cavando via l'http:// e convertendo l'hostname in un ip. Se il client effettua una CONNECT al posto di una GET, il proxy caverà via solo l'http: in quanto la formattazione sarà semplicemente www.aaa.com/directory/file e convertià l'host name in un indirizzo ip.

Chapter 4

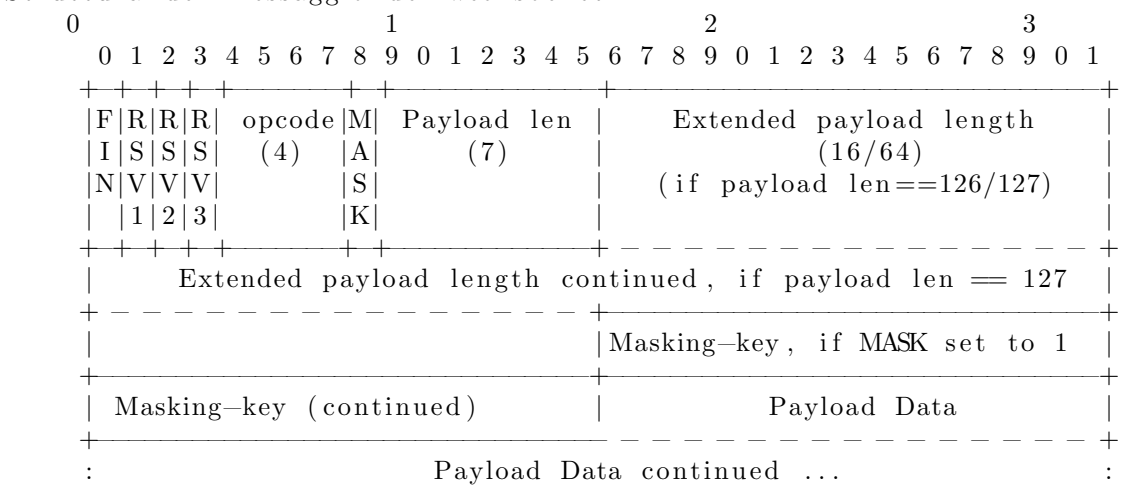
Web socket

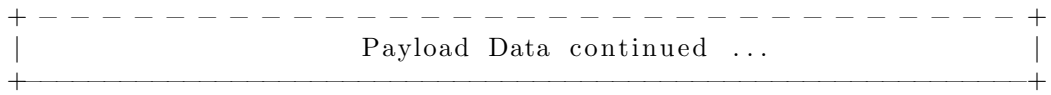
4.1 Parte teorica

Si ha quando vi è un canale bidirezzionale, dove sia client che server fanno robe. Il web socket è un canale sicuro, e si ha quando il client facendo una get al server per creare un canale bidirezionale dove la risposta del server sarà l'upgrade del protocollo. Da quelo momento in poi, tra il client e server c'è un canale bidirezzionale, cioè il server potrà inviare asincronamente dei dati, così come il client, allo script che sta girando sul client. A questo punto, client e server, pur utilizzando la piattaforma web, hanno tutti i gradi di liberta di due programmi qualunque che girano su due sistemi operativi che girano su due macchine remote.
L'effetto risulta nell'avere ua pagina che non dobbiamo ricaricare, ma che si aggiorna automaticamente quando una porzione dei dati viene aggiornata dal server. Tutto questo meccanismo prende il nome di websocket, che simula un socket, un canale di trasporto, tra client e server.
Nell'RFC 6455 si può vedere meglio il web socket. Viene mostrato che il client effettua una GET nella quale ci sono degli elementi nella quale chiede l'upgrade al web socket e manda successivamente una chiave. Richiede l'upgrade della connessione http ad un tipo di protocollo diverso, il web socket. Parte in http e diventa un'altra cosa. Questo meccanismo viene usato anche nell'http2.0. Così esco dai vincoli dell'http.
Ma cosa deve fare il client? Il client deve rispondere con HTTP/1.1 101 Switching Protocol, una risposta informativa. Gli si dice ok per l'upgrade, ok per il web socket e si deve dare una stringa, la Sec-WebSocket-ACcept: QualcosaInBase64. Si prende la stringa di secWebSocketKey, la si concatena con una stringa determinata(Vedi nel programma) e prese queste due stringhe concatenate, bisogna fare un hash usando l'algoritmo SHA-1. Il risultato dell'hash viene codificato in base64 e poi inserito all'interno di Sec-WebSocket-Accept: QuelloCheMiSonoCalcolato.
A questo punto, dopo aver fatto questo, è possibile fare delle read e delle write, dove sul socket che abbiamo utilizzato viene usato un qualcosa che non è http ma web-socket. Il web socket è un protocollo a messaggi, vengono costruiti dei frame binari che vengono inviati. I frame binari sono descritti in una delle lavagne della lezione.

4.1.1 Frame binari

Il messaggio non è più testuale, anche se noi fino a poco fà eravamo abituati ad usare stringhe.





Il messaggio è composto da:

- FIN: che indica se il messaggio è a se o è parte di un altro messaggio.Noi metteremo sempre 1 al campo fin, che indica che è un messaggio unico, fatto e finito.
- 3 bit riservati: RSV1, RSV2,RSV3 che mettiamo a 0.
- 4 bit per l'opcode, che indicano cosa contiene il frame, se è la continuazione di un frame precedente contiene 0 (in esadecimale), 1 se contiene testo, 2 se è binario ecc..
- 1 bit, MASK, che indica se i dati sono mascherati o meno.
- la lunghezza del payload, ovvero quanti caratteri voglio inserire. Posso mettere il valore massimo consentito da 7 bit, 127. da 0 a 125 è la lunghezza permessa, se usp 126 uso la extended length, quindi viene usato lo spazio definito dal campo Extended Payload length. Se non mi bastasse la lunghezza a 16 bit posso usare un campo di lunghezza maggiore. Per fare ciò inserisco 127 all'interno del payload len e posso utilizzare frame lunghi fino a 2 alla 64.
- Masking-key, che risulta essere la chiave usata per il mascheramento.
- i dati.

Per la compilazione bisogna usare il seguente comando, altrimenti non funziona niente: gcc wsock.c -o wsock -lcrypto
Se non si fà così, non si riesce ad usare la funzione SHA1 che ci permette di costruire l'hash di cui abbiamo bisogno.
Necessitiamo anche della pagina websock.html che il professore ha fatto.

4.2 Parte pratica

Per la compilazione bisogna usare il seguente comando, altrimenti non funziona niente: gcc wsock.c -o wsock -lcrypto
Se non si fà così, non si riesce ad usare la funzione SHA1 che ci permette di costruire l'hash di cui abbiamo bisogno.

```
#include<openssl/sha.h>
#include<stdlib.h>
#include <unistd.h>
#include<errno.h>
#include<stdio.h>
#include<string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h> /* superset of previous */
#include <arpa/inet.h>

unsigned char line[501],wsk[50],key[50],token[50];
char encode( unsigned char c)
{
if ( c < 26 ) return c + 'A';
if ( c < 52 ) return c - 26 + 'a';
if ( c < 62 ) return c - 52 + '0';
if ( c == 62) return '+';
if ( c == 63) return '/';
else return '?';
}

int base64 (unsigned char * in, char * out, int lungh) {
long int * a;
long int par = 0;
char reb[4]="";
int i=0, cont=0;

reb[3]=in[0];
reb[2]=in[1];
reb[1]=in[2];
```

```
32
33 a = (long int *) reb;
34 par = *a;
35
36         out[3]=encode((char)(par>>8)&63);
37         out[2]=encode((char)(par>>14)&63);
38         out[1]=encode((char)(par>>20)&63);
39         out[0]=encode((char)(par>>26)&63);
40 return 4;
41 }
42
43 void mask(unsigned char * buf, int buflen, unsigned char * msk, int msklen)
44 {
45 while(buflen--) buf[buflen]=buf[buflen]^msk[buflen%msklen];
46 }
47
48
49
50 int b64(unsigned char * inbuf,char *output)
51 {
52 int in_len=0,out_len=0;
53 int i,x;
54 int quanti_per_riga=0;
55 char c;
56 unsigned char input[3];
57 for (; *inbuf ; inbuf++){ //wait for terminator
58         input[in_len]=*inbuf;
59         if(in_len==2){
60                 x = base64(input,output, 3);
61                 in_len = 0;
62                 output+=x;
63                 }
64         else
65                 in_len = in_len + 1;
66 } //wait for terminator
67
68 if(in_len == 2){
69         input[2]=0;
70         x = base64(input,output,2);
71   output+=3;
72         *(output++)='=';
73         *(output++)= 0;
74         }
75 if(in_len == 1){
76          input[1]=input[2]=0;
77          x= base64(input,output,2);
78   output+=2;
79         *(output++)='=';
80         *(output++)='=';
81         *(output++)= 0;
82         }
83 }
84
85
86
87 char * p;
88 int lunghezza;
89 int yes=1;
90 struct header {
91 char * n;
92 char * v;
93 };
94 struct header h[100];
95 struct sockaddr_in indirizzo;
96 struct sockaddr_in indirizzo_remoto;
97 int primiduepunti;
98 char request[10000];
99 char response[10000];
100 char* request_line;
101 char * method, *uri, *http_ver;
102 int c;
103 FILE *fin;
104 int idazien;
105 int main()
```

```
{
int s,s2,s3,t,j,i;
char command[1000];
s = socket(AF_INET,SOCK_STREAM,0);
if (s == -1){
  perror("Socket Fallita");
  return 1;
}
if ( setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1 ) {
    perror("setsockopt");
        return 1;
}
indirizzo.sin_family=AF_INET;
indirizzo.sin_port=htons(8181);
indirizzo.sin_addr.s_addr=0;

t=bind(s,(struct sockaddr *) &indirizzo, sizeof(struct sockaddr_in));
if (t==-1){
  perror("Bind fallita");
  return 1;
  }
t=listen(s,10);
if(t==-1){
  perror("Listen Fallita");
  return 1;
  }
while( 1 ){
lunghezza = sizeof(struct sockaddr_in);
s2=accept(s,(struct sockaddr *)&indirizzo_remoto, &lunghezza);
if (s2 == -1){
  perror("Accept Fallita");
  return 1;
  }
if(fork()) continue; //if parent go to next accept....

// if son  manage the connection

h[0].n=request;
request_line=h[0].n;
h[0].v=h[0].n;
for(i=0,j=0; (t=read(s2,request+i,1))>0;i++){
  printf("%c",request[i]);
  if (( i>1) && (request[i]=='\n') && (request[i-1]=='\r')){
    primiduepunti=1;
    request[i-1]=0;
    if(h[j].n[0]==0) break;
    h[++j].n=request+i+1;
    }
  if (primiduepunti && (request[i]==':')){
    h[j].v = request+i+1;
    request[i]=0;
    primiduepunti=0;
  }
}
if (t == -1 ) {
  perror("Read Fallita");
  return 1;
}
wsk[0]=0;
for(i=1;i<j;i++){
  if(!strcmp("Sec-WebSocket-Key",h[i].n)){
    printf("*");
    strcpy(wsk,h[i].v+1);
    }
  printf("%s ==> %s\n",h[i].n,h[i].v);
}
if(wsk[0]){ // E' un web socket!
  strcat(wsk,"258EAFA5-E914-47DA-95CA-C5AB0DC85B11");
  SHA1(wsk,strlen(wsk),key);
  b64(key,token);
  sprintf(response,"HTTP/1.1 101 Switching Protocols\r\nUpgrade: WebSocket\r\nConnection: Upgrade\r\
    nSec-WebSocket-Accept:%s\r\n\r\n",token);
  printf("%s",response);
  t=write(s2,response,strlen(response));
```

```
  t=read(s2,response,strlen(response));
  printf("Response:");
  for(int u=0;u<t;u++) printf("%.2X ",(unsigned char) response[u]);
  mask(response+6,response[1]&0x7F,response+2,4);
  printf(" (");
  for(int u=6;u<t;u++) printf("%c",(unsigned char) response[u]);
  printf (")\n");
  struct sockaddr_in wsa;
  for(int ch=1; ch<12;ch++){
      printf("\nChat>");
      fgets(request+2,500,stdin);
      t=strlen(request+2);
      request[0]=0x81;request[1]=(t&0x7F);
      if( write(s2,request,t+2)==-1){
        perror("write a web fallita");
        close(s3);close(s2);
        exit(1);
        }
      printf(">>%s\n",request+2);
      for(int u=0;u<t+2;u++) printf("%.2X ",(unsigned char) request[u]);
      printf("\n");
      }
  close(s2);
  exit(1);
  }
method = request_line;
for(i=0;request_line[i]!=' '&& request_line[i];i++){};
if (request_line[i]!=0) { request_line[i]=0; i++;}
uri = request_line + i;
for(;request_line[i]!=' '&& request_line[i];i++){};
if (request_line[i]!=0) { request_line[i]=0; i++;}
http_ver = request_line +i;

printf("Method = %s, URI = %s, Http-Version = %s\n", method, uri, http_ver);

if(!strncmp(uri,"/exec/",6)){
  sprintf(command,"%s > filetemp\n",uri+6);
  printf("Eseguo il comando %s",command);
  if((t=system(command))==0){
    uri = "/filetemp";
  }
  else printf("system ha restituito %d\n",t);
}


if ((fin = fopen(uri+1,"rt"))==NULL){
  printf("File %s non aperto\n",uri+1);
  sprintf(response,"HTTP/1.1 404 File not found\r\n\r\n<html>File non trovato</html>");
t=write(s2,response,strlen(response));
if (t==-1){
  perror("write fallita");
  return -1;
  }
} else {
  sprintf(response,"HTTP/1.1 200 OK\r\n\r\n");
  t=write(s2,response,strlen(response));
  while((c = fgetc(fin))!=EOF){
    if (write(s2,(unsigned char *)&c,1)!=1){
      perror("Write fallita");
    }
  }
  fclose(fin);
  }
  close(s2);
  exit(1);
}
}
```

Per la compilazione bisogna usare il seguente comando, altrimenti non funziona niente: gcc wsock.c -o wsock -lcrypto
Se non si fà così, non si riesce ad usare la funzione SHA1 che ci permette di costruire l'hash di cui abbiamo bisogno.

Chapter 5

Esami passati

5.1 TRACE

```c
#include<stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>
#include <arpa/inet.h>
#include <stdint.h>
#include <unistd.h>
#include <stdlib.h>

struct sockaddr_in remote;
char response[1000001];
struct header {
        char * n;
        char * v;
} h[100];

int main(){
        size_t len = 0;
        int i,j,k,n,offset,chunklen,bodylen=0;
        char * request = "TRACE / HTTP/1.1\r\nHost: www.radioamatori.it\r\n\r\n";
        char * statusline;
        char hbuffer[10000], ckbuffer[100];
        unsigned char ipserver[4] = {46,37,17,205};
        int s;
        if (( s = socket(AF_INET, SOCK_STREAM, 0 )) == -1)
        { printf("errno = %d\n",errno); perror("Socket Fallita"); return -1; }
        remote.sin_family = AF_INET;
        remote.sin_port = htons(80);
        remote.sin_addr.s_addr = *((uint32_t *) ipserver);
        if ( -1 == connect(s, (struct sockaddr *)&remote,sizeof(struct sockaddr_in)))
        {perror("Connect Fallita"); return -1;}

        for(k=0; k < 1; k++){
                write(s,request,strlen(request));
                bzero(hbuffer,10000);
                statusline = h[0].n = hbuffer;
                for (i=0,j=0; read(s,hbuffer+i,1); i++) {
                        if(hbuffer[i]=='\n' && hbuffer[i-1]=='\r'){
                                hbuffer[i-1]=0; // Termino il token attuale
                                if (!h[j].n[0]) break;
                                h[++j].n=hbuffer+i+1;
                        }
                        if (hbuffer[i]==':' && !h[j].v){
                                hbuffer[i]=0;
                                h[j].v = hbuffer + i + 1;
                        }
                }
                bodylen=0;
                for(i=1;i<j;i++){
                        if(!strcmp("Content-Length",h[i].n))
                                bodylen=atoi(h[i].v);
```

```
53                          if(!strcmp("Transfer-Encoding",h[i].n))
54                                  if (!strcmp(" chunked",h[i].v)) bodylen = -1;
55                  }
56                  if(bodylen == -1){
57                          offset = 0;
58                          chunklen = 1;
59                          while(chunklen){
60                                  chunklen = 0;
61                                  for(j=0; (n = read(s, ckbuffer + j, 1)) > 0 && ckbuffer[j] != '\n' &&
    ckbuffer[j-1] != '\r'; j++){
62                                          if(ckbuffer[j] >= 'A' && ckbuffer[j] <= 'F')
63                                                  chunklen = chunklen*16 + (ckbuffer[j]-'A'+10);
64                                          if(ckbuffer[j] >= 'a' && ckbuffer[j] <= 'f')
65                                                  chunklen = chunklen*16 + (ckbuffer[j]-'a'+10);
66                                          if(ckbuffer[j] >= '0' && ckbuffer[j] <= '9')
67                                                  chunklen = chunklen*16 + ckbuffer[j]-'0';
68                                  }
69                                  if (n==-1) { perror("Read fallita"); return -1;}
70                                  ckbuffer[j-1] = 0;
71                                  for(len=0; len<chunklen && (n = read(s,response + offset, chunklen -
    len))>0; len+=n, offset+=n);
72                                  if (n==-1) { perror("Read fallita"); return -1;}
73                                  read(s, ckbuffer, 1);
74                                  read(s, ckbuffer +1,1);
75                                  if(ckbuffer[0]!='\r' || ckbuffer[1] != '\n'){ printf("Errore nel chunk\
    n"); return -1;}
76                          }
77                          response[offset] = 0;
78                          if(!strncmp(h[0].n,"HTTP/1.1 200",strlen("HTTP/1.1 200"))){
79                                  if(!strcmp(request,response)) printf("\n-->Proxy trasparente<--\n\n");
80                          }
81                          else{
82                                  printf("\n-->Il proxy e' un furbacchione<--\n%s\n", response);
83                          }
84                  }
85          }
86  }
```

Non tutti i siti accettano il metodo TRACE. Lo scopo della traccia era individuare se il proxy modifica la richiesta del client e quindi manda indietro un qualcosa di diverso rispetto alla request principale o se la request che viene tornata è quella che effettivamente si vuole. Se la requestline non viene modificata si notifica che il proxy è trasparente, altrimenti si notifica che non lo è.

5.2 Server web chunked

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <sys/types.h>          /* See NOTES */
5  #include <sys/socket.h>
6  #include <errno.h>
7  #include <arpa/inet.h>
8  #include <stdint.h>
9  #include <unistd.h>
10
11 struct sockaddr_in local, remote;
12 char request[1000001];
13 char response[1000];
14
15 struct header {
16         char * n;
17         char * v;
18 } h[100];
19
20
21 int main()
22 {
23         char hbuffer[10000], ckbuffer[1000];
24         char * reqline;
25         char * method, *url, *ver;
26         char * filename;
27         FILE * fin;
```

```
28          char c;
29          int n;
30          int maxck=1024;
31          char rsp[maxck+1];
32          int i,j,t,s,s2;
33          int yes = 1;
34          int len, cklen;
35          if (( s = socket(AF_INET, SOCK_STREAM, 0 )) == -1)
36          { printf("errno = %d\n",errno); perror("Socket Fallita"); return -1; }
37          local.sin_family = AF_INET;
38          local.sin_port = htons(26024);
39          local.sin_addr.s_addr = 0;
40
41          t= setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
42          if (t==-1){perror("setsockopt fallita"); return 1;}
43
44          if ( -1 == bind(s, (struct sockaddr *)&local,sizeof(struct sockaddr_in)))
45          { perror("Bind Fallita"); return -1;}
46
47          if ( -1 == listen(s,10)) { perror("Listen Fallita"); return -1;}
48          remote.sin_family = AF_INET;
49          remote.sin_port = htons(0);
50          remote.sin_addr.s_addr = 0;
51          len = sizeof(struct sockaddr_in);
52          while ( 1 ){
53                  s2=accept(s,(struct sockaddr *)&remote,&len);
54                  bzero(hbuffer,10000);
55                  bzero(h,sizeof(struct header)*100);
56                  reqline = h[0].n = hbuffer;
57                  for (i=0,j=0; read(s2,hbuffer+i,1); i++) {
58                          if(hbuffer[i]=='\n' && hbuffer[i-1]=='\r'){
59                                  hbuffer[i-1]=0;
60                                  if (!h[j].n[0]) break;
61                                  h[++j].n=hbuffer+i+1;
62                          }
63                          if (hbuffer[i]==':' && !h[j].v){
64                                  hbuffer[i]=0;
65                                  h[j].v = hbuffer + i + 1;
66                          }
67                  }
68
69                  printf("%s\n",reqline);
70                  if(len == -1) { perror("Read Fallita"); return -1;}
71                  method = reqline;
72                  for(i=0;reqline[i]!=' ';i++); reqline[i++]=0;
73                  url=reqline+i;
74                  for(; reqline[i]!=' ';i++); reqline[i++]=0;
75                  ver=reqline+i;
76                  for(; reqline[i]!=0;i++); reqline[i++]=0;
77                  if (!strcmp(method,"GET")){
78                          filename = url+1;
79                          fin=fopen(filename,"rt");
80                          if (fin == NULL){
81                                  sprintf(response,"HTTP/1.1 404 Not Found\r\n\r\n");
82                                  write(s2,response,strlen(response));
83                          }
84                          else{
85                                  sprintf(response,"HTTP/1.1 200 OK\r\n");
86                                  write(s2,response,strlen(response));
87                                  sprintf(response,"Transfer-Encoding: chunked\r\n\r\n");
88                                  write(s2,response,strlen(response));
89                                  while(1){
90                                          bzero(rsp,maxck+1);
91                                          for(i=0; i<maxck && (c=fgetc(fin))!=EOF; i++)
92                                                  rsp[i]=c;
93                                          printf("CHUNK SIZE = 0x%x\n",i);
94                                          if(i==0) break;
95                                          sprintf(response,"%x\r\n",i);
96                                          write(s2,response,strlen(response));
97                                          write(s2,rsp,strlen(rsp));
98                                          sprintf(response,"\r\n");
99                                          write(s2,response,2);
100                                         if(c==EOF) break;
101                                 }
```

```
                                fclose(fin);
                                sprintf(response,"0\r\n");
                                write(s2,response,strlen(response));
                                sprintf(response,"\r\n\r\n");
                                write(s2,response,strlen(response));
                        }
                }
                else {
                        sprintf(response,"HTTP/1.1 501 Not Implemented\r\n\r\n");
                        write(s2,response,strlen(response));
                }
                close(s2);
        }
        close(s);
}
```

Questo programma trasmette un file attraverso la codifica chunked. Quando viene effettuata una richiesta viene mandato come header al client un header, il transfer-encoding: chunked. Successivamente si punta a mandare i dati codificati in chunked. Per mandare un corpo in chunked si usa la seguente sintassi: chunklen(in esadecimale)-CRLF-chunkbody-CRLF. Si ripete questa linea fintantoché ci sono caratteri da mandare. Quando il corpo del file è terminato si manda un ultimo chunk di lunghezza nulla, con la struttura: 0-CRLF-trailer(CRLF nel nostro caso)-CRLF. Così facendo, la trasmissione tramite chunk è terminata.

5.3 Linked pages web server

```
#include<stdio.h>
#include<arpa/inet.h>
#include<string.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<errno.h>
#include<stdint.h>
#include<unistd.h>
struct sockaddr_in local,remote;                    //Indirizzo remoto e locale
struct header{                          //Struttura per gli header
  char* n;
  char* v;
}h[100];
char response[1000];                        //Dove memorizzera' la risposta
char request[1000001];
int main(){
  char hbuffer[10000];                        //Buffer degli header
  char * requestline;
  char *method, *url, *ver;
  int s,t,len,s2,i,j;
  int yes = 1;
  //Inizzializziamo il primo socket
  if((s = socket(AF_INET,SOCK_STREAM,0))==-1){printf("errno = %d",errno); perror("Socket fallita");}
  //Inizzializziamo l'indirizzo locale
  local.sin_family = AF_INET;
  local.sin_port = htons(26024);                    //Numero scelto da noi
  local.sin_addr.s_addr = 0;
  //Impostiamo il sockopt
  t = setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
  if(t == -1){printf("errno = %d",errno);perror("setosockpoint fallita");}
  //Effettuiamo la bind, rendiamo il socket passivo?
  if(bind(s, (struct sockaddr*)&local,sizeof(local))==-1){printf("errno = %d",errno);}
  //Controllo che la listen funzioni
  if (listen(s,10) == -1){printf("errno = %d",errno); perror("listen fallita");}
  //Inizializziamo l'indirizzo remoto
  remote.sin_family = AF_INET;
  remote.sin_port = htons(0);
  remote.sin_addr.s_addr = 0;
  //Inizializziamo la len
  len = sizeof(struct sockaddr_in);

      //Effetuiamo ora il ciclo per le richieste
  while(1){
    //La accept restituisc eun nuovo socket, il socket s2.
    s2 = accept(s,(struct sockaddr*)&remote,&len);
    //Ora tocca al parseing con tutto cio' che e' annesso
```

```
47      bzero(hbuffer,10000);
48      bzero(h, sizeof(struct header)*100);
49      requestline = h[0].n = hbuffer;
50      for(i = 0,j = 0;read(s2,hbuffer+i,1);i++){
51        if (hbuffer[i] == '\n' && hbuffer[i-1] == '\r'){
52          hbuffer[i-1] = 0;
53          if(!h[j].n[0]) break;
54          h[++j].n = hbuffer + i + 1;
55        }
56        if(hbuffer[i] == ':' && !h[j].v){
57          hbuffer[i] = 0;
58          h[j].v = hbuffer + i + 1;
59        }
60      }
61      printf("%s \n", requestline);
62      if(len == -1){perror("Read Fallita");}
63      method = requestline;
64      for(i = 0; requestline[i] != ' ';i++);
65      requestline[i++] = 0;
66      url = requestline + i;
67      for(; requestline[i] != ' ';i++);
68      requestline[i++] = 0;
69      ver = requestline + i;
70      for(; requestline[i]!=0; i++);
71      requestline[i++] = 0;
72      if(!strcmp(method,"GET")){
73        char* filename = url+1;
74        FILE* fin = fopen(filename, "rt");
75        if(fin == NULL){
76          sprintf(response, "HTTP/1.1 404 Not Found\r\n\r\n");
77          write(s2,response,strlen(response));
78        }
79        else{
80          sprintf(response, "HTTP/1.1 200 OK\r\n");
81          write(s2,response,strlen(response));
82          int c;
83          int length = 0;
84          char body[1000001];
85          bzero(body,1000001);
86          while((c = fgetc(fin))!=EOF){
87            body[length] = c;
88            length++;
89          }
90          sprintf(response,"Content-Length: %d\r\n\r\n",length);
91          write(s2,response,strlen(response));
92          //sprintf(response,body,strlen(body));
93          //sprintf(response,"%s",body);
94          write(s2,body,strlen(body));
95          fclose(fin);
96        }
97      }
98      else{
99        sprintf(response,"HTTP/1.1 501 Not Implemented\r\n\r\n");
100       write(s2,response,strlen(response));
101     }
102     //close(s2);
103   }
104   close(s2);
105   close(s);
106 }
```

Il problema principale riscontrato in questo esercizio sono state gli apici che il prof aveva dato. Infatti davano problemi per la codifica in utf-8. Per far eseguire una navigazione web, abbiamo spostato la chiusura del socket attivo fuori dal ciclo di richieste. Abbiamo aggiunto l'header Content-Length in modo che possa essere inviato al client, seguito da due write, una che manderà la content-length effettiva, ed un'altra che invierà il corpo del messaggio.