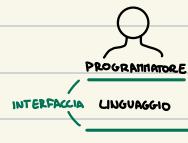


Reti di Calcolatori

1. Algebra dei Puntatori

Nel corso utilizzeremo il linguaggio C. Questo perché è uno dei pochi linguaggi che espongono alla macchina.



C ci dà accesso ad istruzioni che sono quasi 1:1 con l'assembly, ma allo stesso tempo è molto più comodo nella scrittura.

1.1 Puntatori

1.1.1 Puntatori a Byte

```
#include <stdio.h>
```

```
char *p;  
char a, b;
```

```
int main() {  
    a = 'A';  
    b = 'B';
```

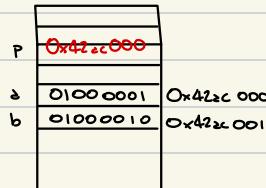
repräsent come char
printf("%c e %c\n", a, b); // A e B
printf("%d e %d\n", a, b); // 65 e 66
printf("%x e %x\n", &a, &b); // 42ac000 e 42ac000
 come hex

```
p = &a;  
*p = 10;
```

```
printf("%d\n", a); // 10 ] ho modificato a senza nominarlo
```

```
*p+1 = 12;
```

printf("%d\n", b); // 12] ho modificato b senza avere un puntatore esplicito



1.1.2 Puntatori ad Array

```
#include <stdio.h>
```

```
char *p;  
char a[5];
```

```
int main() {  
    a[0] = 'c';  
    a[1] = 'i';  
    a[2] = 'a';  
    a[3] = 'o';
```

```
p = &a[0];
```

```
printf("%c\n", *(p+2)); // a
```

printf("%s\n", a); // ciao
 rappresenta come stringa

```
return 0;
```

OSS $a \equiv \&a[0]$ sono modi diversi per esprimere le stesse cose.

$*(p+x) \equiv p[x]$ l'unica differenza è che a non può essere modificata.

OSS In nessuno dei due casi c'è una verifica per l'overflow del vettore,

la verifica può essere fatta dal programmatore.

In questo modo se le prestazioni contano si può ottimizzare.

OSS Una stringa è un array di char terminato da uno 0

OSS O perché esiste il bit di stato che verifica la condizione.

c'è uno in memoria → il bit di stato è zero? → si → fine

no → stampo e continuo

1.1.3 Tipi Multi-Byte

```
#include <stdio.h>

u_int8_t *p;
u_int16_t a;

int main() {
    a = 0xF032;
    p = &a;
    printf("%x\n", *p); // 0x32
}
}
```

OSS Qui il programma può ritornare 0xF0 oppure 0x32, dipende se l'architettura utilizza il big endian o il little endian come standard.

BIG ENDIAN

00	0x4002
F1	0x4003

inizie dai bit più significativi

LITTLE ENDIAN

F1	0x4002
00	0x4003

inizie dai bit meno significativi

OSS little endian è usato dalle principali architetture (x86, arm64,...)
big endian è il network standard, usato dalle reti di calcolatori.

1.1.4 Puntatori Multi-Byte

```
#include <stdio.h>

short int *p;
short int a, b;

int main() {
    a = 0x00F1;
    b = 0xF032;
    p = &a;
    printf("%x\n", *p); // 0x00F1
    printf("%x\n", *(p+1)); // 0xF032
}

return 0;
}
```

oss la somma tra un puntatore e un intero N non è la semplice somma tra l'indirizzo puntato ed N. La formula usata è:

$$\text{indirizzo puntato} + Nx(\text{dimensione del tipo del puntatore})$$

oss queste convenzione rende coerenti puntatori ed array anche con tipi multi-byte.

1.2 Struct

Sono un modo per raggruppare un insieme di variabili.

```
#include <stdio.h>
```

```
struct esempio {  
    char c;  
    char d;  
    short int x;  
    short int y;  
};
```

```
struct esempio v;  
  
int main()  
{  
    v.c = 'A';  
    v.d = 10;  
    v.x = 1024;  
    v.y = 1;  
  
    return 0;  
}
```

non alloca memoria, ho definito un nuovo tipo chiamato "struct esempio"

0100 0001	0x4000 ≡ &v ≡ &v.c
0000 1010	0x4001 ≡ &v.d
1024	0x4002 ≡ &v.x
	0x4003
1	0x4004 ≡ &v.y
	0x4005

oss vedremo che ci sono casi in cui l'allocazione delle variabili non avviene in ordine.

se le variabili sono in una struct allora devono seguire l'ordine.

La struct è un "picchietto" di dati che può essere trasferito in una rete di calcolatori.

1.2.1 Lettura non Allineata

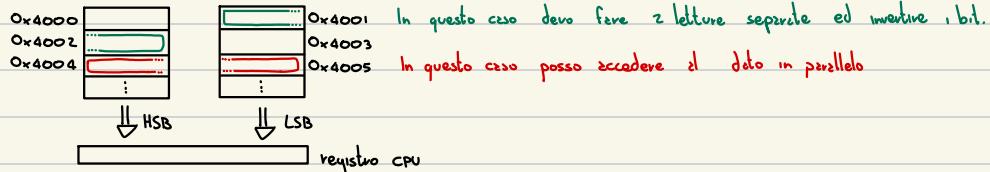
```
#include <stdio.h>
```

```
struct problema {  
    short int x;  
    char c;  
    short int y;  
    char d;  
};  
  
struct problema v;  
  
int main()  
{  
    v.c = 'A';  
    v.d = 10;  
    v.x = 1024;  
    v.y = 1;  
  
    return 0;  
}
```

riempimento o → padding

1024	0x4000 ≡ &v.x ≡ &v.c
0100 0001	0x4001
0000 1010	0x4002 ≡ &v.y
	0x4003
1	0x4004 ≡ &v.d
	0x4005
	0x4006 ≡ &v.x

La CPU legge dalla memoria in parallelo i byte che formano una parola. Ad esempio in un'architettura a 16 bit



L'operazione in verde non è gestita dalle CPU più vecchie, solo quelle più recenti lo gestiscono (anche se più costose)

1.3 Operatori

	bitwise operator	boolean operator
AND	&	&&
OR		
XOR	^	X

```
#include <stdio.h>
```

```
char a, b, c, d, e, f;
```

```
int main() {
    a = 0xE6;
    b = 0x83;
    c = a & b;
    printf("%x\n", c); // 0x82
    d = b | c;
    printf("%x\n", d); // 0x83
    printf("%x\n", a&b); // 1 (true)
```

a	1110 0110
b	1000 0011
c	1000 0010
d	1000 0011

Oss le operazioni bitwise sono effettivamente parallelizzabili.

L'operazione `&&` ritorna false \Leftrightarrow tutti i bit di `a` o `b` sono 0

L'operazione `||` ritorna false \Leftrightarrow tutti i bit di `a` e `b` sono 0

```
e = d << 3;
printf("%x\n", e); // 0x18
f = d >> 3;
printf("%x\n", f); // 0xF0
}
```

d	1000 0011
e	0001 1000
f	1111 0000

L'operatore `<<` aggiunge zeri a destra
L'operatore `>>` aggiunge 1 a sinistra se il lipo
NON è unsigned E il $b_7 = 1$

L'operatore `<< n` (left shift) è equivalente ad una moltiplicazione per 2^n

L'operatore `>> n` (right shift) è equivalente ad una divisione per 2^n

0	1	3
0	0	2
0	0	1
0	0	0
1	1	1
1	1	2
1	0	3
1	0	4
0	0	4
0	0	3
0	0	2
0	0	1
0	0	0

$\frac{-4}{2^2} = -1 \checkmark$

1.3.1 BitMask

```

int bit_select(unsigned int sourceBits, unsigned char pos) {
    if (sourceBits & (1 << pos)) {
        return 1;
    }
    return 0;
}

void set_bit(unsigned int *sourceBits, unsigned char pos) {
    *sourceBits = *sourceBits | (1 << pos);
}

void toggle_bit(unsigned int *sourceBits, unsigned char pos) {
    *sourceBits = *sourceBits ^ (1 << pos);
}

```

1.4 Aree di Memoria

```

#include <stdio.h>
#include <stdlib.h>

int x;

void stack() {
    int y;
    printf("y: %x\n", &y);
}

stack();

void heap() {
    int* z = (int*)malloc(1 * sizeof(int));
    printf("z: %x\n", z);
}

heap();
free(z);
}

```

The diagram illustrates the memory layout. On the left, code snippets show variable declarations and allocations. To the right, memory is divided into three main sections:

- GLOBAL SCOPE:** At the top, four memory locations are shown with addresses: 0x2a000, 0x53904060, 0x53904060, and 0x53904060.
- LOCAL SCOPE:** Below the global scope, a vertical bar labeled "HEAD" indicates the current stack pointer. The stack grows downwards, with addresses 0x619e321c, 0x619e323c, and 0x619e325c listed from top to bottom.
- MEMORIA DINAMICA:** At the bottom, a vertical bar labeled "STACK" shows dynamically allocated memory starting at address 0x619e321c, with addresses 0x619e323c and 0x619e325c listed below it.

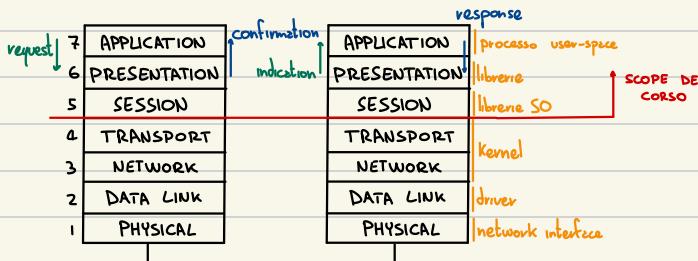
2. Modello OSI

Il modello OSI (Open Systems Interconnection) è uno standard per l'architettura di una rete di calcolatori.

Il layering permette di estrarre la complessità delle comunicazioni tra due sistemi diversi:

- permette l'estrazione, la modularità e la riutilizzabilità del software.

- permette l'interoperabilità tra diversi dispositivi di produzione diversi



request => indication

response => confirmation

se scende sono chiamate → funzione
se sale sono → callback (signal/interrupt)
chiamate bloccanti (multithreading)

ES - HTTP

request: HTTP request es GET <https://www.google.com>

indication: il server di google riceve la richiesta HTTP

response: il server invia la risposta es 200 OK

confirmation: il client riceve la risposta

Ogni livello è un'estensione del precedente:

7. Fornisce i servizi alle applicazioni (HTTP, FTP)

6. Trasforma i dati in un formato standard compresso (JPEG, ASCII)

5. Apre una connessione tra due punti della rete seguendo un protocollo. (RPC - remote procedure call)

4. Trasferimento affidabile dei dati (TCP/UDP)

3. Forma i pacchetti e li indirizza (IP)

2. Verifica che non ci siano errori nella comunicazione (ACK $\xrightarrow{\text{bit}} \xleftarrow{\text{bit}}$)

1. Converte i bit in segnali da trasmettere nel mezzo (modulazione/codice)

2.1 Protocol Data Unit

Ogni livello del modello OSI aggiunge un header e passa i dati al livello inferiore.

PDU₇ ← [H₇ | SDU₇]

PDU₆ ← [H₆ | SDU₆]

⋮

In ricevimento ogni livello toglie un header e passa i dati al livello superiore.

3. Modello Client-Server



È il modello più comune, è però soggetto a diversi vincoli.

- Il client deve sempre prendere l'iniziativa

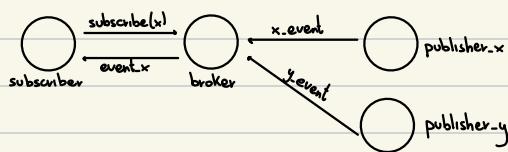
- server è in perpetua attesa del client

- la risposta deve arrivare in tempo brev (timeout)

- tanti client possono fare richieste allo stesso tempo sovraccaricando il server

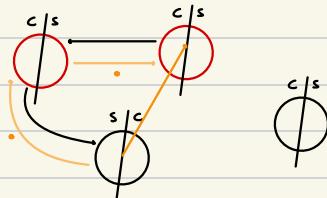
3.1 Altri modelli

3.1.1 Publish-Subscribe-Notify



È il modello duale del client-server

3.1.2 Peer-2-Peer



4. Web Client HTTP

L'internet nasce nel 1976 ed è una connessione tra calcolatori. Inizialmente utilizzato da accademici.

Successivamente, nel 1991, Berners Lee utilizza l'internet per trasmettere file di ipertestivo contenenti link ad altri file.

Nasce così il World Wide Web.

Lee descrive un modello client-server ed il protocollo HTTP, un protocollo a livello applicativo.

4.1 HTTP 0.9



Lo standard è ancora implementato dai siti moderni, questo perché HTTP 1.0 e 1.1 sono dei superset di HTTP 0.9.

4.1.1 webclient

```
int socket(int domain, int type, int protocol)
```

Apre un endpoint da cui posso inviare dati ad un server remoto.

Ritorna un indice ad un file descriptor (un oggetto su cui si possono fare operazioni input/output)

- domain:

seleziona una famiglia di protocolli che saranno utilizzati per la comunicazione.

ES AF_INET: ipv4 AF_INET6: ipv6 AF_X25: radio

- type:

selezione il tipo di socket

ES SOCK_STREAM: stream, va aperto e chiuso, trasmissione affidabile con ritrasmissione dei pacchetti

TCP

SOCK_DGRAM: messaggio finito che si aspetta una risposta in breve tempo.

UDP

non implementa ritrasmissione, devo gestire io una sequenza richieste.

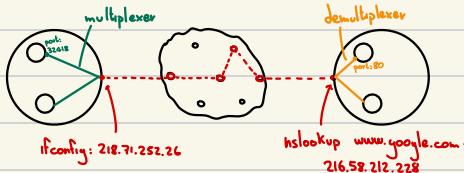
- protocol:

selezione il protocollo nel dominio, se ce n'è solo uno basta usare "0"

oss Può essere utile avere più socket tra un client ed un server.

L'indirizzo ip identifica 1 pc su un network, è un numero unico a 32 bit (es. 218.71.252.26)

Il port identifica il servizio e con il client vuole accedere (es. port 80 per fare una richiesta HTTP, 22 per SSH,...)



```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
```

Connette un socket ad un determinato indirizzo (handshake se necessario)

- sockfd: l'id del file descriptor del socket da connettere

- addr: struct sockaddr è una struttura molto generica

```
struct sockaddr {  
    unsigned short sa_family; // address family, AF_XXX  
    char sa_data[14];  
};
```

Visto che stiamo usando sa_family=AF_INET usiamo la struct sockaddr_in

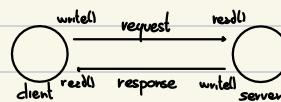
```
struct sockaddr_in {  
    short sin_family; // famiglia di indirizzi  
    unsigned short sin_port; // numero di porta (big-endian)  
    struct in_addr sin_addr; // indirizzo IP  
};  
  
struct in_addr {  
    unsigned int s_addr; // indirizzo IP  
};
```

- addrlen: lunghezza della struct addr

```
int write(int socket, const void* buf, int length)
```

Scrive length bytes dal buffer buf al socket.

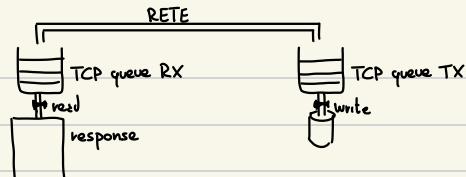
Ritorna il numero di byte trasmessi.



```
int read(int socket, const void* buf, int length)
```

Scrive length bytes dal socket al buffer buf.

Ritorna il numero di byte trasferiti.



OSS

Non abbiamo il controllo sui dati in ingresso, è quindi necessario attendere il loro arrivo.

La `read()` è bloccante, ritorna solo quando riceve un dato.

Se l'informazione è divisa in più pacchetti, devo usare un `while() {read()}` per leggerli tutti.

OSS

C'è inoltre il rischio che la TCP queue RX si riempie, in questo caso c'è un meccanismo che blocca la trasmissione, riempendo la TCP queue TX.

Quando si riempie anche la TCP queue RX il write scrive solo quelli che ci stanno, bisogna gestire gli altri dati.

4.2 HTTP 1.0

Nel 1996 (5 anni dopo HTTP 0.9) viene formalizzato lo standard HTTP tramite l'RFC 1945.

Aggiunge nuovi metodi di richiesta oltre al GET e formalizza il protocollo.

4.2.1 Convenzioni e Grammatica

È un protocollo application level, vuole quindi essere human-readable, per questo si usa "GET" e non "o".

ES-date

È una definizione di definizioni.

```

HTTP-date = rfc1123-date | rfc850-date | asctime-date
rfc1123-date = wkday "," SP date1 SP time SP "GMT"
rfc850-date = weekday "," SP date2 SP time SP "GMT"
asctime-date = wkday SP date3 SP time SP 4DIGIT
date1 = 2DIGIT SP month SP 4DIGIT
                  ; day month year (e.g., 02 Jun 1982)
date2 = 2DIGIT "-" month "-" 2DIGIT
                  ; day-month-year (e.g., 02-Jun-82)
date3 = month SP 2DIGIT | ( SP 1DIGIT )
                  ; month day (e.g., Jun 2)
time = 2DIGIT ":" 2DIGIT ":" 2DIGIT
                  ; 00:00:00 - 23:59:59
wkday = "Mon" | "Tue" | "Wed"
                  | "Thu" | "Fri" | "Sat" | "Sun"
weekday = "Monday" | "Tuesday" | "Wednesday"
                  | "Thursday" | "Friday" | "Saturday" | "Sunday"
month = "Jan" | "Feb" | "Mar" | "Apr"
                  | "May" | "Jun" | "Jul" | "Aug"
                  | "Sep" | "Oct" | "Nov" | "Dec"

```

Sab, 29 Mar 2025 09:30:20 GMT

4.2.2 Request

Molte scelte sono prese per mantenere la retrocompatibilità con HTTP 0.9.

Ecco perché c'è Simple-Request e perché le richieste non iniziano con la versione di HTTP.

Request = Simple-Request **or** Full-Request

Simple-Request = "GET" SP Request-URI CRLF

NOTA L'URI (Uniform Resource Identifier) è un caso più generale dell'URL (Uniform Resource Locator).

NOTA CRLF = Character Return / Line Feed = "\r\n" = new line

Full-Request = Request-Line * (General-Header | Request-Header | Entity-Header) CRLF [Entity-Body]

Request-Line = Method SP Request-URI SP HTTP-Version CRLF

Method = "GET" | "HEAD" | "POST" | extension-method

HTTP-Version = "HTTP" "/" 1*DIGIT "." 1*DIGIT

4.2.3 Response

Response = Simple-Response | Full-Response

Simple-Response = [Entity-Body]

Full-Response = Status-Line * (General-Header | Response-Header | Entity-Header) CRLF [Entity-Body]

OSS La Simple-Response è particolarmente problematica, non ci sono informazioni oltre al file. Ad oggi quasi nessun server la supporta. Une richieste HTTP 0.9 riceverà una risposta in HTTP 1.0.

oss - Status codes

Ixx: Informazioni / Non utilizzato, me riservato per uso futuro.

2xx: OK, Success

3xx: Redirect, il client deve intraprendere delle azioni per ottenere la risposta

4xx: client error

5xx: Server error

4.2.4 Codice

Faremo una richiesta al server di google. Per il parsing delle risposte vogliamo inserire gli header in una struct:

```
struct headers {
    char* name;
    char* value;
};
```

Per farlo non copio i dati, utilizzo il buffer e ci sovriscrivo sopra l'ency di strutt.

```
int main() {
    int sock;
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("Socket fallito");
        return 1;
    }

    struct sockaddr_in server;
    server.sin_family = AF_INET;
    server.sin_port = htons(80);
    unsigned char* p = (unsigned char*)&server.sin_addr.s_addr;

    p[0] = 216;
    p[1] = 58;
    p[2] = 213;
    p[3] = 4;

    if (connect(sock, (struct sockaddr*)&server, sizeof(struct sockaddr_in)) ==
        -1) {
        perror("Connect fallito");
        return 1;
    }

    char* request = "GET / HTTP/1.0\r\n\r\n"; // GET /CRLF
    write(sock, request, strlen(request));

    char raw_headers[100000];
    struct headers h[100];

    int j = 0;
    h[j].name = raw_headers;
    char first_separator = 1;

    for (int i = 0; read(sock, raw_headers + i, 1) > 0; i++) {
        if ((raw_headers[i] == ':') && first_separator) {
            raw_headers[i] = 0;
            h[j].value = raw_headers + i + 1;

            first_separator = 0;
        }
        if ((raw_headers[i - 1] == '\r') && (raw_headers[i] == '\n')) {
            raw_headers[i - 1] = 0;
        }
        h[++j].name = raw_headers + i + 1;

        if (h[j - 1].name[0] == 0)
            break;
    }

    first_separator = 1;
}

for (int i = 0; i < j; i++) {
    printf("None: %s → Valore: %s\n", h[i].name, h[i].value);
}

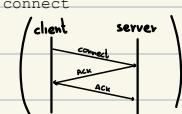
return 0;
}
```

4.3 HTTP 1.1

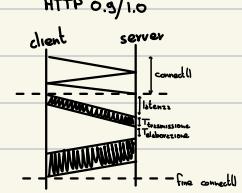
HTTP 1.1 risolve un altro problema di HTTP 0.9 e 1.0 ovvero il fatto che ogni richiesta chiede una nuova connessione.

Primo infatti usavamo char 0 per riconoscere la fine del body, ma questo indica anche la chiusura del connect

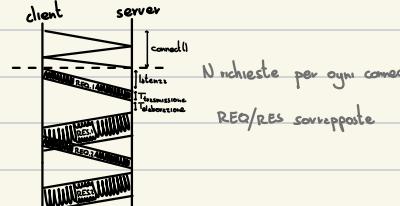
Avere un connect per ogni richiesta è particolarmente inefficiente per via del 3-way handshake



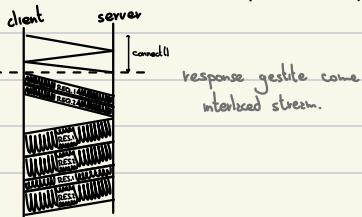
HTTP 0.9/1.0



HTTP 1.1



HTTP 2



Il parsing del body diventa:

```
int content_length = 100;
char chunked = 0;
for (int i = 0; i < j; i++) {
    printf("%s-> Valore: %s\n", h[i].name, h[i].value);
}

if (!strcmp(h[i].name, "Content-Length")) {
    content_length = atoi(h[i].value);
    printf("Found content-length %d\n", content_length);
}

if (!strcmp(h[i].name, "Transfer-Encoding")) {
    if (!strcmp(h[i].value, "chunked")) {
        chunked = 1;
        printf("Body is chunked\n");
    }
}

if (chunked) {
    int chunk_length = 0;
    char crlf[2];

    do {
        char raw_body[100];

        for (int i = 0; read(sock, raw_body + i, 1); i++) {
            if (raw_body[i - 1] == '\r' && raw_body[i] == '\n') {
                raw_body[i - 1] = 0;
                break;
            }
        }

        chunk_length = strtol(raw_body, NULL, 16);

        char response[chunk_length];
        int n;
        for (int x = 0; (n = read(sock, response + x, chunk_length - x)) > 0; x += n)
            ;
        read(sock, crlf, 2);

        response[chunk_length] = 0;
        printf("%s", response);
    } while (chunk_length > 0);

    printf("\n");
} else {
    char body[content_length];

    int n;
    for (int x = 0; (n = read(sock, body + x, content_length - x)) > 0; x += n)
        ;

    body[content_length] = 0;
    printf("%s\n", body);
}
```

5. Web Server HTTP

Per poter ricevere connessioni devo tenere aperte una porta in modo passivo.

Il server deve poter scegliere le porte su cui ricevere le connessioni (port binding) (Es 80 per HTTP)

```
socket(AF_INET, SOCK_STREAM, 0);
```

```
bind(sock, struct sock_addrin, length);
```

nelle connect avviene in automatico

```
listen(sock, pending_connection_queue);
```

quente connessioni

```
accept(sock, struct sock_addrin, &length);
```

restano in attesa

oss le richieste sono un processo elettorio di Poisson.

Le loro distribuzione è quindi "a blocchi":

contiene l'IP e la port
del client

genera un nuovo socket
connesso al primo in coda

è più probabile avere intervalli più piccoli

oss tutti i socket generati dall'accept sono legati allo stesso port.

Il server identifica le connessioni tramite le quadruple (IPremoto : PORTremoto, IPlocale : PORTlocale)

```

int main() {
    int sock;

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket fallito");
        return 1;
    }

    int yes = 1;
    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int))) {
        perror("setsockopt fallito");
        return 1;
    }

    struct sockaddr_in server;
    server.sin_family = AF_INET;
    server.sin_port = htons(8085);

    unsigned char* p = (unsigned char*)&server.sin_addr;
    p[0] = 0; p[1] = 0; p[2] = 0; p[3] = 0;

    // binding the socket
    if (bind(sock, (struct sockaddr*)&server, sizeof(struct sockaddr_in)) == -1) {
        perror("Bind fallito");
        return 1;
    }

    // listening to the socket
    if (listen(sock, 5) == -1) {
        perror("Listen fallito");
        return 1;
    }

    struct sockaddr_in remote;
    socklen_t remote_length = sizeof(struct sockaddr_in);

    while (1) {
        int sock2;

        if ((sock2 = accept(sock, (struct sockaddr*)&remote, &remote_length)) == -1) {
            perror("accept fallita");
            return 1;
        }

        char raw_headers[10000];
        struct Headers h[100];

        int j = 0;
        h[j].name = raw_headers;
        char first_separator = 1;

        for (int i = 0; read(sock2, raw_headers + i, 1) > 0; i++) {
            if ((raw_headers[i] == ':') && first_separator) {
                raw_headers[i] = 0;
                h[j].value = raw_headers + i + 1;

                first_separator = 0;
            }
            if ((raw_headers[i - 1] == '\r') && (raw_headers[i] == '\n')) {
                raw_headers[i - 1] = 0;
            }
        }

        h[++j].name = raw_headers + i - 1;
        if (h[j - 1].name[0] == 0)
            break;

        first_separator = 1;
    }

    for (int i = 0; i < j; i++)
        printf("Nome: %s → Valore: %s\n", h[i].name, h[i].value);

    int i = 0;
    char* request_line = h[0].name;
    char* method = request_line;

    for (; request_line[i] != '\n'; i++);
    request_line[i] = 0;

    char* url = request_line + i + 1;

    for (; request_line[i] != '\n'; i++);
    request_line[i] = 0;

    char* version = request_line + i + 1;

    printf("Method: %s\nURL: %s\nVersion: %s\n", method, url, version);

    char* filename = url + 1;

    if (!strcmp(url, "/getmay/*")) {
        char* eseguibile = url + 9;
        char comando[1000];
        sprintf(comando, "%s > output\n", eseguibile);
    }
}

```

```

if (system(comando) == -1) {
    perror("system fallita");
    return 1;
}

filename = "output";
}

FILE* fin;
if ((fin = fopen(filename, "rt")) == NULL) {
    char* response_404 = "HTTP/1.0 404 Not Found\r\n\r\n";
    write(sock2, response_404, strlen(response_404));
} else {
    char* response_200 = "HTTP/1.0 200 OK\r\n\r\n";
    write(sock2, response_200, strlen(response_200));
}

char c;
while ((c = fgetc(fin)) != EOF)
    write(sock2, &c, 1);

fclose(fin);
}

close(sock2);
}

return 0;
}

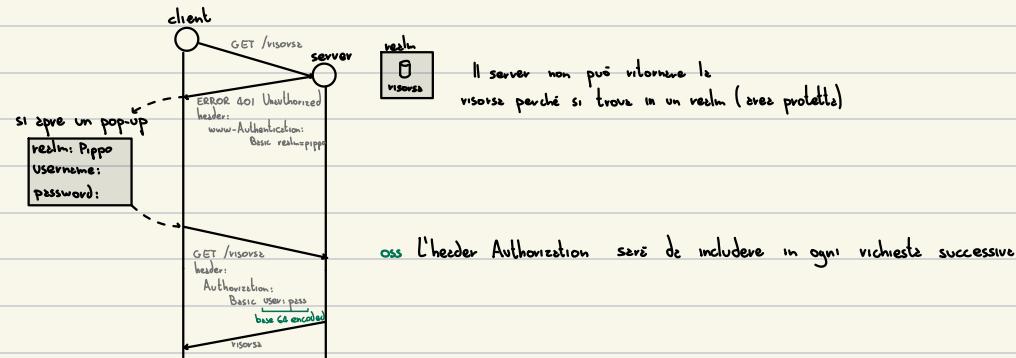
```

5.1 Statefulness

Per definizione della RFC 1345 HTTP è un protocollo stateless. Questo significa che le richieste sono tempo-invarianti.

L'invio di una specifica richiesta comporta sempre le stesse risposte.

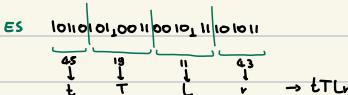
Tuttavia nasce la necessità di mantenere una sessione per poter identificare gli utenti.



5.1.1 Base64

Utile per convertire caratteri speciali in caratteri stampabili. Funziona come l'hex, ma codifica 6 bit per carattere.

Usa i 64 caratteri 0-2, A-Z, 0-9, /, +

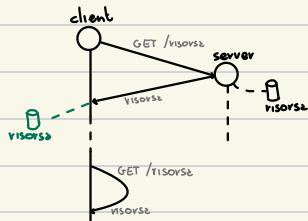


oss Può essere necessario aggiungere 1 o 2 byte per allinearsi ad un multiplo di 6 bit.

In tal caso aggiungo '=' o '==' alla fine per segnalare l'aggiunta.

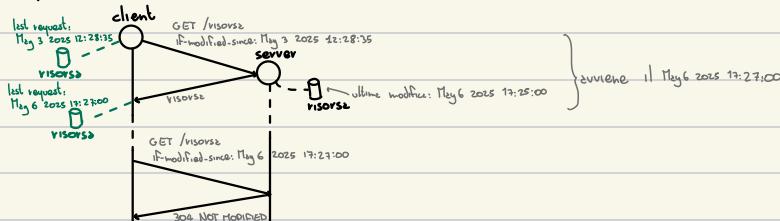
5.2 Caching

Il client salva in cache le risorse inviate dal server.



Un possibile problema si può trovare nel caso in cui le risorse sul server cambino.

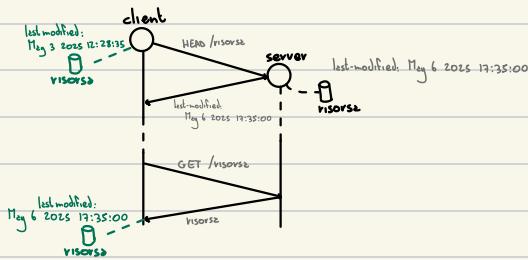
- Si può usare un header (If-modified-since)



- Quando scarico il file salvo l'header last-modified

Per le richieste successive faccio prima una richiesta HEAD /resource

Che mi ritorna solo gli header, tra cui il last-modified. Se è più recente di quello locale lo riscrivo.



oss Dopo un po' il client può provare a stimare ogni quanto tempo il file viene aggiornato.

Per risparmiare richieste HEAD che anche se più leggere sono comunque costose.

- Il server può dare una data di scadenza (expires)

Funziona bene con dati che sono aggiornati regolarmente.

5.3 URL e URI

URI = (absoluteURI | relativeURI)

absoluteURI = scheme ":" * (uchar | reserved)

relativeURI = net_path | abs_path | rel_path

non trattato, ma sono vari relativi tipo:

images/meme.jpg o /home/m/hello.jpg o //www.discord.com/images/meme.jpg
(rel.path) (abs.path) (net-path)

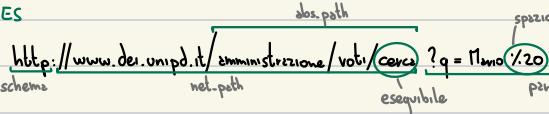
scheme = 1* (ALPHA | DIGIT | "+" | "-" | ".")

reserved = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+"

uchar = unreserved | escape

unreserved = ALPHA | DIGIT | safe | extra | national

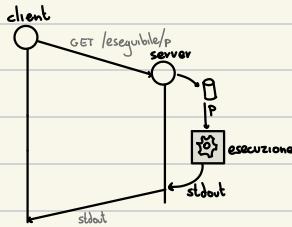
escape = "%" HEX HEX



5.4 Web Dinamico

Abbiamo appena descritto come invocare un eseguibile e come specificarne i parametri tramite URL.

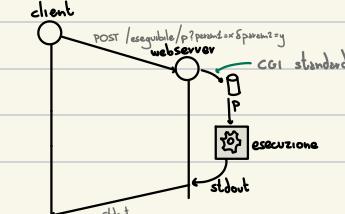
Lato server l'eseguibile viene eseguito e lo stdout inviato al client.



Il codice per questo caso è sotto `if (!strcmp(url, "/gateway/", 9))`

5.4.1 Parametri/CGI

è possibile passare dei parametri agli eseguibili, lo standard che il webserver deve seguire è definito dall'RFC 3875, Common Gateway Interface



Le informazioni delle richieste sono passate come variabili d'environment. Invece il messaggio body è passato tramite stdin.
oss Una delle meta-variabili è CONTENT_LENGTH e indica la lunghezza dell'entity body delle HTTP request che viene passato all'eseguibile p.

es QUERY_STRING contiene questo stringo: "?param1=x¶m2=y"

5.4.2 Esecuzione Parallela

Non si vuole che l'esecuzione di processi interrompe la ricezione di richieste.

Una combinazione di fork e execve ci può aiutare.

- fork crea un nuovo processo e restituisce al genitore il process id del figlio e il figlio restituisce 0
- execve avvia l'esecuzione di un nuovo processo e uccide il chiamante.

Per parallelizzare l'esecuzione si può quindi eseguire:

```
int main(int argc, char* argv[], char** env)
{
    if (fork() == 0) {
        // child
        execve("./test", argv, env);
        printf("here is child\n"); // this will not print
    } else {
        // parent
        printf("here is dad\n");
    }
}
```

5.4.3 Implementazione

```
struct environment {
    char* buf;
    char** env;
    int i;
    int c;
};

void add_env(struct environment* env, char* env_key, char* env_value) {
    sprintf(env->buf + env->c, "%s=%s", env_key, env_value);
    env->env[env->i+1] = env->buf + env->c;
    env->c += (strlen(env_value) + strlen(env_key) + 2);
    env->env[env->i] = NULL;
}

int main() {
    [...]
    while (1) {
        struct environment env;
        char b[1000];
        char* e[100];
        env.buf = b;
        env.env = e;
        env.i = env.c = 0;

        int sock2;
        if ((sock2 = accept(sock, (struct sockaddr*)&remote, &remote_length)) ==
            -1) {
            perror("accept-fallita");
            return 1;
        }

        char raw_headers[100000];
        struct headers h[100];

        int j = 0;
        h[j].name = raw_headers;
        char first_separator = 1;
```

```

for (int i = 0; read(sock2, raw_headers + i, 1) > 0; i++) {
    if ((raw_headers[i] == ':') && first_separator) {
        raw_headers[i] = 0;
        h[j].value = raw_headers + i + 1;

        first_separator = 0;
    }
    if ((raw_headers[i - 1] == '\r') && (raw_headers[i] == '\n')) {
        raw_headers[i - 1] = 0;
    }
    h[++j].name = raw_headers + i + 1;

    if (h[j - 1].name[0] == 0)
        break;

    first_separator = 1;
}
}

int length = 0;
for (int i = 0; i < j; i++) {
    printf("Nome: %s → Valore: %s\n", h[i].name, h[i].value);
    if (!strcmp(h[i].name, "Content-Length")) {
        length = atoi(h[i].value);
    }
}

if (!strcmp(h[i].name, "Content-Type")) {
    add_env(&env, "CONTENT_TYPE", h[i].value + 1);
}
}

int i = 0;
char* request_line = h[0].name;
char* method = request_line;

for (; request_line[i] != '\0'; i++)
;
request_line[i] = 0;

char* url = request_line + i + 1;

for (; request_line[i] != ' '; i++)
;
request_line[i] = 0;

char* version = request_line + i + 1;

printf("Method: %s\nURL: %s\nVersion: %s\n", method, url, version);

add_env(&env, "METHOD", method);

char* filename = url + 1;
FILE* fin;

if (istrncmp(url, "/cgi/", 5)) { // CGI
    if (!strcmp(method, "GET")) {
        int i;
        for (i = 0; filename[i] && (filename[i] != '?'); i++)
        ;
        if (filename[i] == '?') {
            filename[i] = 0;
            add_env(&env, "QUERY_STRING", filename + i + 1);
        }
        add_env(&env, "CONTENT_LENGTH", "0");
    } else if (!strcmp(method, "POST")) {
        char tmp[10];
        sprintf(tmp, "%d", length);
        add_env(&env, "CONTENT_LENGTH", tmp);
    } else {
        char* response_501 = "HTTP/1.0 501 Not Implemented\r\n\r\n";
        write(sock2, response_501, strlen(response_501));
        close(sock2);
        continue;
    }
}

printf("%s\n", filename);

if ((fin = fopen(filename, "rt")) == NULL) {
    char* response_404 = "HTTP/1.0 404 Not Found\r\n\r\n";
    write(sock2, response_404, strlen(response_404));
}

```

```

} else {
    char* response_200 = "HTTP/1.0 200 OK\r\n\r\n";
    write(sock2, response_200, strlen(response_200));
    fclose(fin);

    char fullname[200];

    sprintf(fullname, "/Users/m/Developer/ing-informatica-unipd/Reti di Calcolatori/Code/webServer/%s", filename);

    char* myargv[10];
    myargv[0] = fullname;
    myargv[1] = NULL;

    int pid;
    if (!pid = fork()) {
        dup2(sock2, 1);
        dup2(sock2, 0);
        if (-1 == execve(fullname, myargv, env.env)) {
            perror("execve");
            exit(1);
        }
    }
    waitpid(pid, NULL, 0);
}

else if (!strcmp(method, "GET")) { // NOT CGI
    if (!strncmp(url, "/gateway/", 9)) {
        char* eseguibile = url + 9;
        char comando[1000];
        sprintf(comando, "%s > output\n", eseguibile);

        if (system(comando) == -1) {
            perror("system fallita");
            return 1;
        }
    }

    filename = "output";
}

if ((fin = fopen(filename, "rt")) == NULL) {
    char* response_404 = "HTTP/1.0 404 Not Found\r\n\r\n";
    write(sock2, response_404, strlen(response_404));
} else {
    char* response_200 = "HTTP/1.0 200 OK\r\n\r\n";

    char c;
    while ((c = fgetc(fin)) != EOF)
        write(sock2, &c, 1);

    fclose(fin);
}

else { // NOT IMPLEMENTED
    char* response_501 = "HTTP/1.0 501 Not Implemented\r\n\r\n";
    write(sock2, response_501, strlen(response_501));
}

close(sock2);

close(sock);

return 0;
}

```

un ipotetico file cy1exe.c è:

```
int main(int argc, char* argv[], char* env[]) {
    int length = 0;

    for (int i = 0; env[i]; i++) {
        char* key = env[i];
        int j;
        for (j = 0; env[i][j] != '='; j++)
            ;
        env[i][j] = 0;
        char* value = env[i] + j + 1;

        if (!strcmp(key, "CONTENT_LENGTH"))
            length = atoi(value);
    }

    char* buffer = malloc(length);
    int t;

    printf("body:\n");
    for (int i = 0; i < length && (t = read(0, buffer + i, length - i)); i += t)
        ;

    buffer[length] = 0;
    printf("%s", buffer);

    printf("environment:\n");
    for (int i = 0; env[i]; i++)
        printf("%s--> %s\n", env[i], env[i] + strlen(env[i]) + 1);
}
```

6. Domain Name System

È particolarmente utile associare dei nomi agli indirizzi IP.

L'RFC 1034 descrive il database che serve queste associazioni. È stato scritto nel 1987, per un internet molto meno utilizzato, ma i principi sono tali per cui è riuscito a scalare con l'aumento delle richieste.

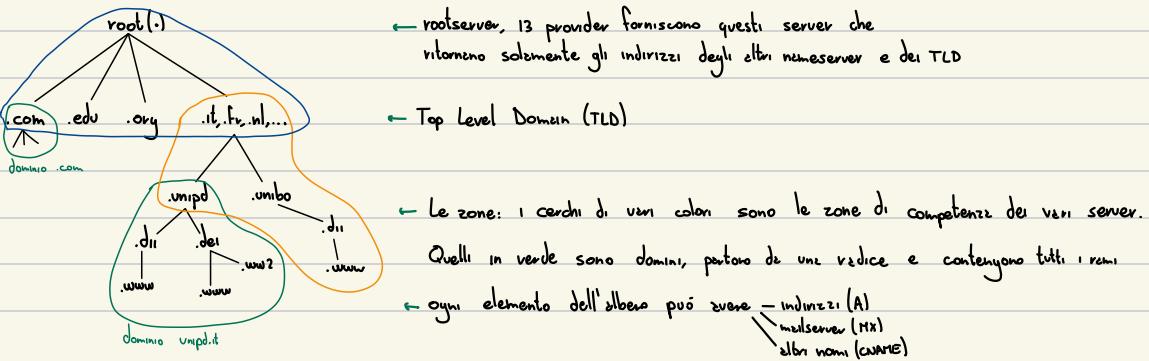
- database distribuito con utilizzo massivo delle cache

- La fonte del dato deve controllare i tradeoff (es la fonte controlla dopo quanto tempo far scadere le cache)

- Più applicazioni ne devono beneficiare, visti i costi fissi. (es A:ipv4, AAAA:ipv6, MX:mail exchange)

6.1 Architettura

Per gestire la grande quantità di dati essi sono distribuiti secondo una struttura ad albero



Per trovare un indirizzo IP del nome si fanno una serie di richieste

ES

www.google.com.

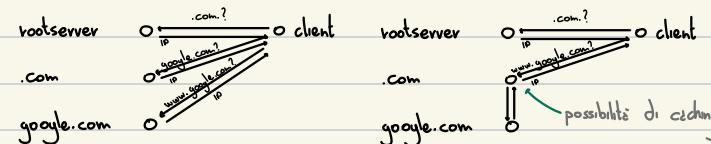
con richieste recursive

nel codice

```
struct hostent* he = gethostbyname("www.google.com");
```

```
printf("IP Address: %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]);
```

```
struct sockaddr_in server;  
server.sin_family = AF_INET;  
server.sin_port = invert_byte_order(80);  
server.sin_addr.s_addr = *(unsigned int*)(he->h_addr);
```



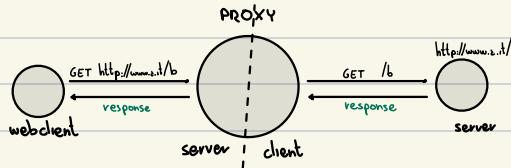
OSS

Ogni resolver ha i 13 rootserver e poi si aggiornano a vicenda

7. Proxy / Tunnel

Il proxy è sia un client che un server. Un client ci si connette e lui fa le richieste al server riportando le risposte al client.

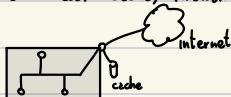
Il client deve sapere l'indirizzo del client e deve inviare sempre URL assoluti al proxy.



ES

Un proxy può essere utilizzato per connettere una rete locale all'internet.

Fornendo così cache, firewall e autenticazione a tutti i client



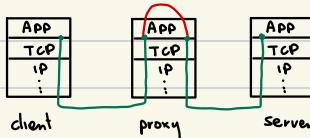
7.1 Tunnel

Il proxy necessita di leggere le richieste, introducendo vulnerabilità.

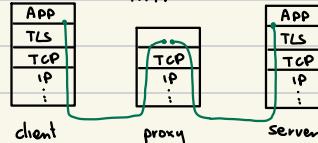
Venne creato un tunnel con una richiesta http (metodo CONNECT) e poi il proxy non guarda più il contenuto.

(non è quindi più possibile fare caching e firewall)

HTTP



HTTP



```

#include <netdb.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

struct headers {
    char* name;
    char* value;
};

int main() {
    int sock;

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket fallito");
        return 0;
    }

    int yes = 1;
    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int))) {
        perror("setsockopt fallito");
        return 1;
    }

    struct sockaddr_in local;
    local.sin_family = AF_INET;
    local.sin_port = htons(8045);
    local.sin_addr.s_addr = 0;

    // binding the socket
    if (bind(sock, (struct sockaddr*)&local, sizeof(struct sockaddr_in)) == -1) {
        perror("Bind fallito");
        return 1;
    }

    // listening to the socket
    if (listen(sock, 5) == -1) {
        perror("Listen fallito");
        return 1;
    }

    struct sockaddr_in remote;
    remote.sin_family = AF_INET;
    socklen_t remote_length = sizeof(struct sockaddr_in);

    while (1) {
        int sock2;

        if ((sock2 = accept(sock, (struct sockaddr*)&remote, &remote_length)) ==
            -1) {
            perror("accept fallita");
            return 1;
        }

        if (fork())
            continue;

        char raw_headers[100000];
        struct headers h[100];

        int j = 0;
        h[j].name = raw_headers;
        for (int i = 0; read(sock2, raw_headers + i, 1) > 0; i++) {
            if ((raw_headers[i] == '\r') && !h[j].value && j > 0) {
                raw_headers[i] = 0;
                h[j].value = raw_headers + i + 1;
            }
            if ((raw_headers[i - 1] == '\r') && (raw_headers[i] == '\n')) {
                raw_headers[i - 1] = 0;
            }
            h[++j].name = raw_headers + i + 1;
        }

        if (h[j - 1].name[0] == 0)
            break;
    }
}

```

```

int i = 0;
char* request_line = h[0].name;
printf("%s\n", request_line);
char* method = request_line;

for (; request_line[i] != ' '; i++)
;
request_line[i] = 0;

char* url = request_line + i + 1;

for (; request_line[i] != ' '; i++)
;
request_line[i] = 0;

char* version = request_line + i + 1;

if (!strcmp(method, "GET")) {
    int i = 0;

    char* schema = url;

    for (; strncmp(url + i,(":/", 3) && url[i]; i++) {
        if (url[i] == 0) {
            printf("Parse error, expected '://'");
            exit(1);
        }
    }

    url[i] = 0;
    i = i + 3;

    char* hostname = url + i;

    for (; url[i] != '/' && url[i]; i++) {
        if (url[i]) {
            url[i+1] = 0;
        } else {
            printf("Parse error, expected '://'");
            exit(1);
        }
    }

    char* filename = url + i;
    printf("schema: %s hostname: %s filename: %s\n", schema, hostname,
        filename);

    struct hostent* he = gethostbyname(hostname);

    int sock3;

    if ((sock3 = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        printf("Socket Fallita");
        exit(1);
    }

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

    if (-1 == connect(sock3, (struct sockaddr*)&server,
        sizeof(struct sockaddr_in))) {
        printf("Connect Fallita");
        exit(1);
    }

    char request[1000];
    sprintf(request,
        "GET /%s HTTP/1.1\r\nHost:%s\r\nConnection:close\r\n\r\n",
        filename, hostname);
    write(sock3, request, strlen(request));

    int t;
    char buffer[2000];
    while (t = read(sock3, buffer, 2000))
        write(sock2, buffer, t);

    close(sock3);
} else if (!strcmp("CONNECT", method)) {
    int i = 0;
    char* hostname = url;

    for (; url[i] != ':' && url[i]; i++)
    ;

```

```

if (url[i] == ':') {
    url[i++ ] = 0;
} else {
    perror("Parse error, expected ':'\n");
    return 1;
}

char* port = url + i;

printf("hostname: %s, port: %s\n", hostname, port);

struct hostent* he = gethostbyname(hostname);

int sock3;

if ((sock3 = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("Creazione socket 3 fallita!\n");
    return 1;
}

struct sockaddr_in server;
server.sin_family = AF_INET;
server.sin_port = htons((unsigned short)atoi(port));
server.sin_addr.s_addr = *(unsigned int*)he->h_addr;

if (connect(sock3, (struct sockaddr*)&server,
            sizeof(struct sockaddr)) == -1) {
    perror("Connect to server fallita");
    return 1;
}

char* response = "HTTP/1.1 200 Established\r\n\r\n";
write(sock2, response, strlen(response));

int pid;
if (! (pid = fork())) {
    // child
    char request[2000];
    int t;

    while (t = read(sock2, request, 2000))
        write(sock3, request, t);
} else {
    // parent

    char request[2000];
    int t;

    while (t = read(sock3, request, 2000))
        write(sock2, request, t);

    kill(pid, SIGTERM);
    close(sock3);
}
} else {
    char* response_501 = "HTTP/1.0 501 Not Implemented\r\n\r\n";
    write(sock2, response_501, strlen(response_501));
}

close(sock2);
exit(1);
}

close(sock);
}

```

Multimedia

0. Introduzione

L'utilizzo di internet è in costante crescita e con esso lo scambio di contenuti multimediali.

Le applicazioni multimedia (videochiamate, streaming,...) sono implementate grazie ad un servizio fornito da una rete di calcolatori: il trasporto di bit.

Cercheremo di passare da descrizioni qualitative (che qualità del video voglio raggiungere) a descrizioni quantitative sulla rete richieste.

0.1 Segnali Multimediali

Un segnale è una funzione del tempo, e sono oggetti che permettono il trasporto di informazione.

oss I segnali possono essere digitali o analogici, nel corso li tratteremo sempre come digitali, il teorema di Shannon ci assicura che possiamo effettuare le conversioni senza perdite di informazioni.

0.1.1 Sorgenti

Le sorgenti di informazioni possono essere classificate in:

- statiche (indipendenti nel tempo) es testo, immagini,...
- dinamiche (producono un segnale dipendente nel tempo) es televisione, sottotitoli, ...

0.3 Servizi Multimediali

Una prima classificazione divide i servizi multimediali in:

- streaming: quando la fruizione dei contenuti inizia prima che l'intero file sia ricevuto
es visione di film, eventi live, DAD, webcast, videochiamate e videoconferenze,...
- bulk transfer: i contenuti possono essere utilizzati solo dopo che l'intero file è stato ricevuto

È possibile classificare ulteriormente in:

- stored es HTML, FTP,...
- live es e-mail, messaggi, ...

1. Conversioni Analogico-Digitale

1.1 Creazione dei Segnali Multimediali (da Analogico a Digitale)

Le sorgenti multimediali sono spesso segnali di natura analogica (ovvero è variabile e valori continui)

$$s: \mathbb{R}^n \rightarrow \mathbb{R}^k$$

Per utilizzarle in sistemi digitali dovranno effettuare la conversione analogico-digitale.

L'operazione avviene in tre operazioni: campionamento, quantizzazione e codifica

È importante che la conversione compatti una perdita di informazione trascurabile, che il segnale sia fedele all'originale.

1.1.1 Campionamento

RICHIAMO SEGNALI

Dato un segnale a tempo continuo $s(t)$, la sua versione campionata, con periodo (o quanto) di campionamento T_c , è il segnale a tempo discreto (ma sempre a valori continui).



TRASFORMATA DI FOURIER - RICHIAMO SEGNALI

Dato un segnale $s(t)$ è possibile calcolare lo spettro $S(f)$, ovvero le ampiezze e frequenze delle sinusoidi che compongono il segnale originale $s(t)$.

Siamo in particolare interessati a segnali a banda limitata ovvero con contributi frequentistici contenuti in un intorno finito



TEOREMA DI NYQUIST-SHANNON

Dato un segnale $s(t)$ a banda limitata, con frequenza massima f_n .

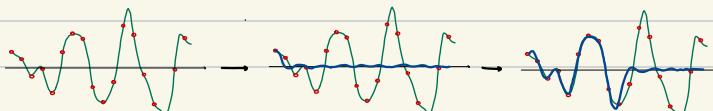
Esso può essere ricostruito con errore nullo a partire dalla successione dei campioni

se e solo se la frequenza di campionamento $F_c = \frac{1}{T_c}$ è almeno il doppio delle frequenze massime f_n ($F_c \geq 2f_n$ è il criterio di Nyquist)

La formula di interpolazione ideale è:

$$s(t) = \sum_{n \in \mathbb{Z}} s(nT_c) \text{sinc}\left(\frac{t-nT_c}{T_c}\right)$$

sinc centrato nell'istante di campionamento



OSS I segnali reali non sono mai a banda limitata, possiamo utilizzare un filtro passa basso per eliminare le altre frequenze, mantenendo una buona fedeltà del segnale.

1.1.2 Quantizzazione

Dopo il campionamento il segnale viene quantizzato, i suoi valori vengono mappati su di un insieme discreto.

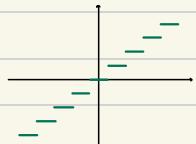
Indichiamo con L il numero di livelli usati. Indichiamo con $R = \log_2 L$ il costo di codifica, ovvero il numero di bit necessari per rappresentare un campione.

QUANTIZZATORE UNIFORME

Il tipo più comune di quantizzazione è quella uniforme, dove ogni intervallo ha ampiezza Δ .

$$Q(x, \Delta) = \Delta \text{round}\left(\frac{x}{\Delta}\right)$$

valore di x quantizzato

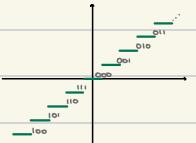


oss! L'operazione di quantizzazione è irreversibile e comporta una perdita di informazione.

Usiamo quindi un numero di livelli sufficiente a rendere impercettibile la differenza.

1.1.3 Codifica

Assegnamo ad ogni livello un valore unico in bit.



oss! In un sistema di trasmissione multimediale sono presenti diversi punti in cui si perdono informazioni.

- nella conversione Analogico → Digitale: qui la perdita di dati deve essere trascurabile
- algoritmo di compressione: lo vedremo, ma può indurre una distorsione anche importante

BITRATE

Un segnale multimediale ha bitrate di:

$$R = F_c \log_2 L \text{ bits/s}$$

1.2 Misure di Distorsione

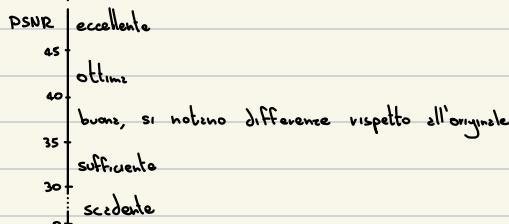
È importante saper quantificare quanto è stato modificato il segnale codificato da quello originale.

Sia $f(n)$ il segnale originale e $g(n)$ il segnale distorto

Definiamo $e(n) = f(n) - g(n)$ il segnale d'errore

Definiamo il mean square error tra f e g $MSE(f, g) = \frac{1}{N} \sum_{n=0}^{N-1} [e(n)]^2 = \frac{1}{N} \|e\|_2^2 = \frac{1}{N} \|f - g\|^2$ è una misura di distanza euclidea tra f e g .

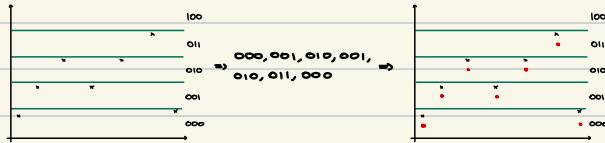
Possiamo portarla in scala logaritmica $PSNR(f, g) = 10 \log_{10} \frac{\max_{n=0}^{N-1} |f(n)|^2}{\text{potenza del disturbo}}$ (b è il numero di bit utilizzato)



1.3 Riproduzione dei Segnali Multimedia (da Digitale ad Analogico)

Partendo dal segnale digitale è semplice svolgere l'operazione inversa della quantizzazione.

Si utilizzza un inverse bit mapper che converte insiem di bit in valori.



Per arrivare ad un segnale continuo nel tempo è necessario interpolare i valori.

1.3.1 Interpolazione

Il teorema di Shannon ci dice che per ricostruire $s(t)$ dai campioni va usata la formula:

$$s(t) = \sum_{n \in \mathbb{Z}} s(nT_c) \operatorname{sinc}\left(\frac{t-nT_c}{T_c}\right)$$

Tuttavia il sinc è una funzione a supporto infinito, non realizzabile fisicamente.

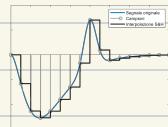
Vogliamo quindi trovare $h(t)$ da sostituire a sinc tale che:

- $h(t)$ ha supporto finito

- $h(n) = \begin{cases} 1 & \text{se } n=0 \\ 0 & \text{se } n \neq 0 \end{cases}$ → In questo modo l'interpolazione conserva i valori negli istanti di campionamento.

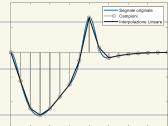
SAMPLE AND HOLD

$$h(t) = \operatorname{rect}\left(t - \frac{1}{2}\right)$$



ORDINE 1

$$h_1(t) = \operatorname{triang}(t)$$



ORDINE 3

Interpolazione $h_{3,n}(t)$ dei punti $(n, s(n)) \in \mathbb{Z}/10\mathbb{Z}$ con un polinomio di grado 3.

$$h(t) = \dots$$

È possibile ottenere approssimazioni migliori all'aumentare di n

SINC TRONCATO

$$h(t) = \text{sinc}(t) \text{rect}\left(\frac{t}{T_2}\right), \quad z \in \mathbb{N}$$

Altra ottima approssimazione, più difficile da realizzare in pratica.

APPLICAZIONI

Per l'audio: - se la frequenza di campionamento >= frequenza di Nyquist allora interpolatori semplici offrono buone prestazioni

- se la frequenza di campionamento ~ frequenza di Nyquist allora è opportuno usare il sinc troncato

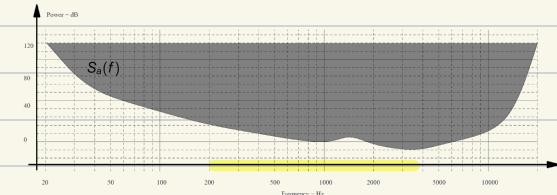
Immagini: reinterpolazione z di ordine 3 (da veloci discreti a veloci discreti, ma con più dettagli 1280x720 \rightarrow 1920x1080)

1.3 Esempio - Conversione del Segnale Audio

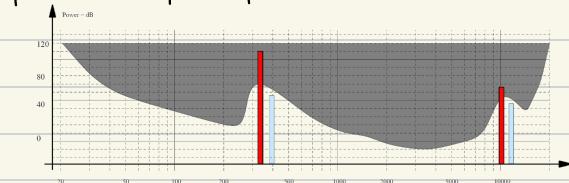
1.3.1 Percezione del Suono

L'orecchio contiene diverse fibre nervose che sono eccitate da frequenze diverse.

In particolare siamo più sensibili alle frequenze della voce.



Questo è vero solo quando in presenza di un'unica tono, in presenza di due toni simili quello di potenza maggiore può mascherare quello di potenza minore.



OSS Sono meschinate più marcatamente le frequenze inferiori.

OSS Questo effetto avviene anche nel tempo, un suono forte ci rende insensibili ad altri suoni per 100/200 ms.

Inoltre non riusciamo a distinguere suoni nei precedenti 2/5 ms

Vogliamo codificare il segnale in modo che il rumore sia meschinato da altre frequenze.

1.3.2 Compressione del Segnale Vocale

COMPRESSEIONE A FORMA D'ONDA

La voce si concentra nella banda [200, 3000] Hz.

Venne usato un passo basso con $f_{out} = 4\text{ kHz}$ e $f_c \geq 8\text{ kHz}$ e un quantizzatore a $7 + 8$ bit.

$$\text{Quindi, } R = 8k \cdot (7 + 8) = 56 \cdot 64 \text{ kbps}$$

CODIFICATORI VOCALI - LCP10

I dati in ingresso sono campionati a 8kHz e sono suddivisi in finestre di 20ms (160 campioni/finestra)

Dato che la voce è stazionaria nei 20ms vogliamo trasmettere solo 10 campioni su 160, il resto lo ottieniamo

per predizione lineare: $x_p(n) = \sum_{k=1}^{10} a_k x(n-k)$

dove i coefficienti a_k si ottengono minimizzando l'errore $e(n) = x(n) - x_p(n)$

Se la predizione è accurata $e(n)$ sarà rumore bianco (non vocale) o rumore con una frequenza dominante, detta pitch (vocale)

quindi $R_{vocale} = \frac{36 + 16}{20\text{ ms}} \text{ vocale/non-vocale} = 2.5 \text{ Kbps}$

$$R_{non-vocale} = \frac{36 + 16}{20\text{ ms}} \text{ vocale/non-vocale} = 2.15 \text{ Kbps}$$

$$R_{media} = 2.4 \text{ kbps}$$

1.3.3 Compressione del Segnale Musicale

Si concentra nella banda [15, 20000] Hz, la banda udibile dell'uomo

Quindi $f_c \geq 40\text{ kHz}$, in particolare $f_c = 44.1\text{ kHz}$ e un quantizzatore a 16 bit con 2 canali.

$$\text{Quindi, } R = 44.1\text{ K} \cdot 16 \cdot 2 = 1.411 \text{ Mbps}$$

1.4 Esempio - Conversione del Segnale Video

Il colore di un pixel è determinato da un vettore di 3 numeri: $\begin{pmatrix} r \\ g \\ b \end{pmatrix}$

Gli elementi di questi vettori sono spesso a 8 bit, ma esistono formati a 10 o 12 bit (High Dynamic Range)

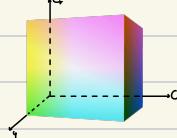
1.4.1 Percezione del Colore

Vogliamo dividere l'informazione portata dai singoli pixel:

- Informazione più rilevante all'occhio umano (luminosità)
- Informazione meno rilevante all'occhio umano (tinta e saturazione)

Lo spazio YC_bC_r fa esattamente questo:

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = T \begin{bmatrix} r \\ g \\ b \end{bmatrix}$$



oss Questo è un esempio di sversificazione del segnale. Ovvero creare un segnale sparso da uno non sparso.

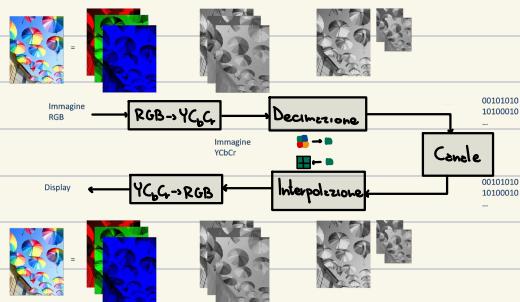
Un segnale non sparso è un segnale dove ogni campione ha pari importanza. Un segnale sparso è l'opposto.

L'occhio umano è molto meno sensibile alla crominanza che alla luminosità.

Possiamo scartare i $\frac{3}{4}$ dei campioni di crominanza senza ottenere degradazioni visibili.

In totale scarto $\frac{1}{2}$ dell'informazione. ($3 = 1 + 1 + 1 \rightarrow 2 + \frac{1}{4} + \frac{1}{4} = \frac{3}{2}$)
R G B Y C_b C_r

Pipeline



2. Codifica

2.1 Codifica Lossless

RICHIAMO TELECOMUNICAZIONI

Def Sia X una sorgente che emette simboli appartenenti ad un certo alfabeto $X = \{x_1, \dots, x_n\}$

Ogni simbolo è caratterizzato da una probabilità $P(X=x_i) = p_i$.

Def L'informazione associata all'evento $X=x_i$ è $i(x_i) = -\log_2 \frac{1}{p_i} = -\log_2 p_i$.

Def Un codice C è una funzione iniettiva $C: x_i \rightarrow c_i$:
$$x \rightarrow \begin{bmatrix} c_{1,i} \\ \vdots \\ c_{N,i} \end{bmatrix}, c_{i,n} \in \{0,1\} \quad \forall n=1\dots N$$

Le stringhe c_i sono caratterizzate da una lunghezza l_i .

Def La lunghezza media del codice è $L = \sum_{i=1}^n p_i l_i$.

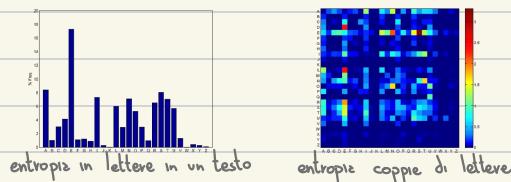
Def L'informazione media è detta entropia $H(X) = E[i(x)] = \sum_{i=1}^n p_i \log_2 \frac{1}{p_i}$.

Cerchiamo codici e lunghezze medi più bassi e univocamente decodificabili ovvero codici a prefisso.

Def In un codice di sorgente è prefisso nessuna parola del dizionario D_X è la prima parte (prefisso) di un'altra parola di codice.

L'entropia è il grado di incertezza sulla realizzazione di una variabile aleatoria:

- Se i simboli sono equiprobabili l'entropia è massima.
- Se un simbolo appare ogni volta l'entropia è 0.



TEO di Shannon

La lunghezza media del codice ottimo per una sorgente X è vincolata da:

$$H(X) \leq L \leq H(X) + 1$$

2.1.1 Codifica di Huffman

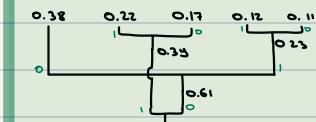
La codifica di Huffman permette di costruire un codice a prefisso ottimo.

Dato la distribuzione di ingresso $p_x(z_i), z_i \in D_X$

1. Ordina le probabilità in ordine decrescente
2. Raccolgo le due probabilità più basse in un'unica parola di codice con probabilità la somma delle due probabilità.
3. Ripeto dal punto 1.

ES

$$p_x(1) = 0.12 \quad p_x(2) = 0.22 \quad p_x(3) = 0.17 \quad p_x(4) = 0.11 \quad p_x(5) = 0.38$$



$$\begin{array}{ll} 1 \rightarrow 011 & 4 \rightarrow 010 \\ 2 \rightarrow 11 & 5 \rightarrow 00 \\ 3 \rightarrow 10 & \end{array}$$

La codifica di Huffman codifica singolarmente ogni simbolo, non sfruttando possibili dipendenze tra simboli.

Raggruppando i simboli in blocchi di dimensione K , indicati con X^K :

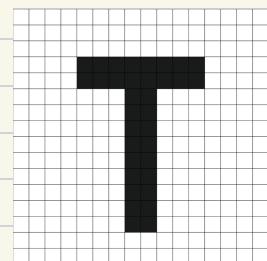
- L'entropia per simbolo decresce (rimane costante se i simboli sono indipendenti) $\frac{H(X^K)}{K} \leq H(X)$
- Rimane valido il teorema di Shannon $H(X^K) \leq L \leq H(X^K) + 1$.

È interessante il fatto che il L tende a zero all'aumentare di K

$$H(X^k) \leq L \leq H(X^K) + 1 \Rightarrow \frac{H(X^k)}{K} \leq \frac{L}{K} \leq \frac{H(X^K)}{K} + \frac{1}{K} \Rightarrow \frac{L}{K} = \lim_{K \rightarrow \infty} \frac{H(X^k)}{K} \Rightarrow L^* = H(X) \leq H(X)$$

Quindi le prestazioni migliorano all'aumentare di K, infatti l'entropia per simbolo diminuisce e si rimuove l'overhead del bit.

ES



$$\text{Se } K=1 \quad P(\square) = 0.867 \quad H(X) = 0.586 \text{ b}$$

$$P(\blacksquare) = 0.133$$

codice ottimo: $\square \rightarrow 1$ $\blacksquare \rightarrow 0$

$$\text{Se } K=2 \quad P(\square\square) = 0.8 \quad H(X_1, X_2) = 1.022 \text{ b} \quad H(X) = 0.511 \text{ b}$$

$$P(\square\blacksquare) = 0.05$$

$P(\blacksquare\square) = 0.05$ codice ottimo: $\square\square \rightarrow 1$ 0.65 bpp
 $\square\blacksquare \rightarrow 010$
 $\blacksquare\square \rightarrow 011$
 $\blacksquare\blacksquare \rightarrow \infty$

$$\text{Se } K=4: \quad H(X_1, X_2, X_3, X_4) = 1.533$$

$$H(X) = 0.383$$

$$0.433 \text{ bpp}$$

2.1.2 Codifica Aritmetica

La codifica di Huffman è esponenziale con K, la codifica aritmetica è subottima, ma è lineare con K.

$$L_A < H(X) + 1$$

Tuttavia il +1, come l'1+, non influenza sulle prestazioni asintotiche, per $K \rightarrow \infty$.

Siano $z_1, \dots, z_n \in D_X$ le parole in ingresso ordinate.



dove m_1, \dots, m_n sono i punti medi

Rappresento m_i in forma binaria: $m_i = 0.010011$

Le parole associate è la parte dopo la virgola (010011) con $b_i = \lceil \log_2 p(z_i) \rceil + 1$

Permette di aggiornare dinamicamente codificatore e decodificatore.

Inizialmente assumo parole equiprobabili al codificatore e decodificatore, poi ad ogni invio (per il trasmettitore) e ricevimento (per il ricevitore) viene aumentata la probabilità di quel simbolo

È anche possibile utilizzare probabilità condizionate invece di $P(x_i)$.

In pratica si sceglie il codificatore (e decodificatore) in base ai bit trasmessi (o ricevuti) in precedenza.

ES

Se trasmetto "ciò" so quasi di per certo che la prossima lettera sarà "o".

Queste codifiche con le probabilità condizionate è lo stato dell'arte. Vediamo ora altre codifiche più semplici e prestazioni inferiori.

2.1.3 Codifica Exp-Golomb

Si usa quando l'alfabeto di ingresso è costituito da $\mathbb{N} \cup \mathbb{Z}$, e quando non si conosce la distribuzione di probabilità.

La codifica assume che la probabilità di un numero decresca con il modulo.

ALGORITMO

Sia $n \in \mathbb{N}$ il numero da codificare, $b = \lceil \log_2(n+1) \rceil + 1$

$$c_b(n) = \underbrace{0 \dots 0}_{b-1 \text{ il numero } n+1 \text{ rappresentato su } b \text{ bit}}$$

n	$n+1$ in b bits	Leading zeros	$c_b(n)$
0	1	-	1
1	10	0	010
2	11	0	011
3	100	00	00100
4	101	00	00101
5	110	00	00110
6	111	00	00111
7	1000	000	0001000
...

$$\begin{array}{l} 0 \rightarrow 0 \\ 1 \rightarrow 1 \\ -1 \rightarrow 2 \\ 2 \rightarrow 3 \\ -2 \rightarrow 4 \\ \vdots \end{array}$$

Per codificare numeri negativi li devo codificare in naturali

2.1.4 Codifica Categorica e Ampiezza

Simile a Exp-Golomb, riduce le dimensioni dei numeri grandi a spese di quelli piccoli.

ALGORITMO

Si definiscono 11 categorie codificate con un codice a prefisso

Dato il numero n la sua categoria è $K = \lceil \log_2(\ln(n)) \rceil$

$$c_K(n) = \underbrace{\dots}_{\text{categoria codifica di } n \text{ su } K \text{ bit}} \quad (\text{se no si fa il complemento a 1})$$

Category	Amplitude	Code
0	0	-
1	+1	1
	-1	0
2	+2	$\xrightarrow{-10} \dots 10$
3	+3	$\xrightarrow{-11} \dots 11$
4	+2	$\xrightarrow{-10} 0$
5	+3	$\xrightarrow{-11} 00$
6	+10	$\xrightarrow{-10} 00$
7	+11	$\xrightarrow{-11} 00$
8	+111110	$\xrightarrow{-111110} 000$
9	+1111110	$\xrightarrow{-1111110} 000$
10	+11111110	$\xrightarrow{-11111110} 000$
11	+1024	$\xrightarrow{-1024} 0000000000$
		\dots

2.1.5 Codifica con Dizionario

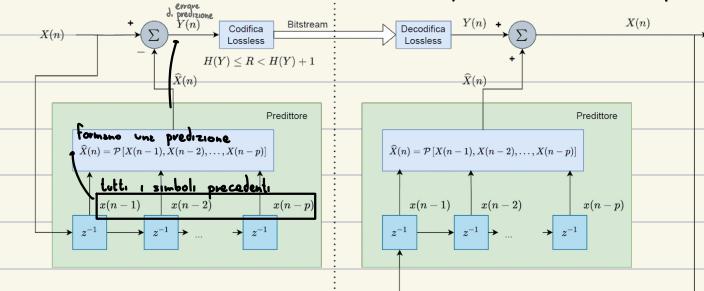
Si crea un dizionario con i simboli più comuni. Il decodificatore deve poter ottenere le labelle.

Efficace per il testo, meno per i multimedia. Utilizzato nei software zip.

2.1.6 Codifica Lossless Predittiva

Se i dati presentano una dipendenza statistica allora posso codificare l'errore di predizione invece del dato

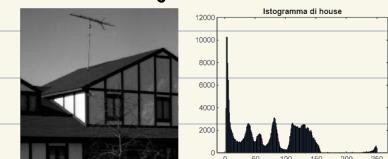
Questo errore avrà una piccola entropia, si può quindi usare un qualsiasi codificatore entropico.



Questo funziona se $H(Y) < H(X)$

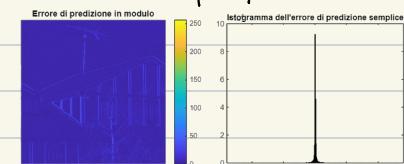
ES

Codificare l'immagine



$$H(X) = 7.056 \text{ bpp}$$

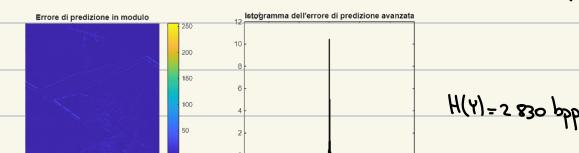
Posso usare un semplice preduttore orizzontale $\hat{X}(n) = X(n-1)$



$$H(Y) = 3.312 \text{ bpp}$$

Un preduttore più avanzato può essere:

$$\hat{X}(n) = \begin{cases} A & \text{se } |C-B| < |C-A| \\ B & \text{altrimenti} \end{cases}$$



$$H(Y) = 2.830 \text{ bpp}$$

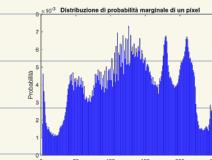
Immagine	Entropia	Exp-Golomb	Huffman	ZIP
Oriģinale	7.056	11.320	7.081	4.003
Preditore semplice	3.312	3.428	3.383	3.026
Preditore avanzato	2.830	2.941	2.893	2.936

2.2 Compressione Immagini

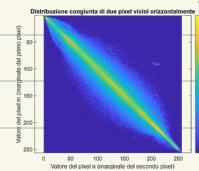
Le codifiche lossless non ha le prestazioni richieste per la trasmissione di segnali video.

Anche la semplice riquantizzazione dell'immagine non ha prestazioni sufficienti (la qualità si degrada velocemente)

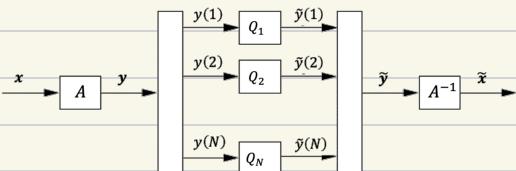
Questo perché i dati dei singoli pixel non sono sparsi:



Lo sono però se raggruppiamo due bit. In particolare è raro che due bit adiacenti abbiano valori molto diversi:

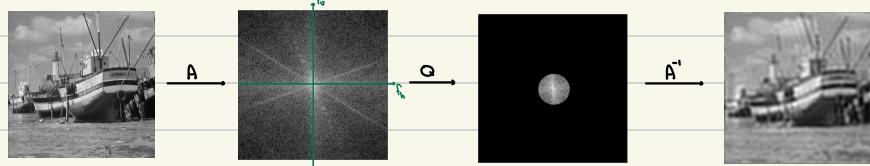


Cerchiamo altre rappresentazioni sparse, in particolare cerchiamo una trasformata lineare A .

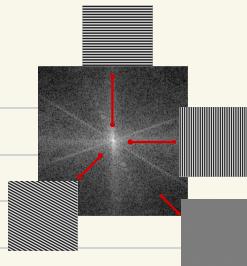


A deve essere ortogonale così che $A^T = A^{-1}$ e deve essere sparsificante, molti valori di y devono essere trascurabili.

ES - Fourier 2D



INTUIZIONE ogni punto (f_x', f_y') su questo grafico rappresenta l'intensità delle sinusoidi di frequenze f_x' e f_y'



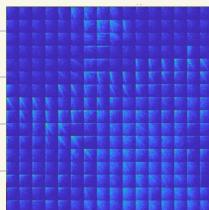
Al centro si trovano frequenze basse, ai bordi frequenze alte.

Visualizzazione

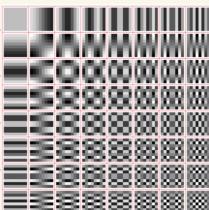
ES - discrete cosine transform

Totalmente simile alla trasformata di Fourier, ma produce solo valori positivi.

Inoltre è possibile dividere l'immagine in blocchi per aumentare la sparsificazione dei dati dove non ci sono bordi.



Spesso si divide l'immagine in blocchi di 8x8 pixel, che quando gli viene applicata la trasformata diventano:



(Ad ogni blocco si associa un valore di intensità, per calcolarla si fa la media del prodotto scalare tra un blocco 8x8 e gli elementi del grafico a sinistra).

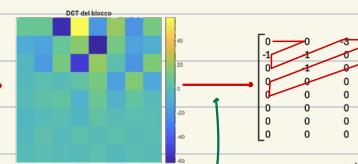
2.2.1 JPEG

Lo standard JPEG nasce nel 1991 ed è il più comune formato per le immagini digitali.

Descriviamo la procedura per un blocco 8x8 in grayscale (le immagini più grandi e a colori sono composizioni di questi blocchi, 3 blocchi rappresentano i colori in YCbCr e possono essere effettuati per creare immagini più grandi)



Blocco selezionato
Y del blocco 8x8



i pixel arrivano da una tabella customizzabile
che viene salvata nell'header del file

0	0	-3	2	0	0	0	0
1	0	0	0	-1	0	0	0
0	4	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

dopo EOF
ci sono solo 0

Il primo coefficiente è detto DC (perché è l'ampiezza quantizzata della sinusode costante (e^0)) viene codificato in modo lossless in 2 passaggi: predizione e codifica categoria/valore.

I coefficienti AC sono codificati in modo lossless in 2 passaggi:

0 -1 0 -1 -3 2 0 -1 0 0 0 0 1 -1 0 -1 -1

→ nella coppia il primo numero è il numero di
zen che precedono il secondo numero

$$(1, -1) \quad (1, -1) \quad (0, -3) \quad (0, 2) \quad (1, -1) \quad (4, 1) \quad (0, -1) \quad (1, -1) \quad (0, -1) \quad (0, 0)$$

dette le coppie (R, C) rappresentano C in categorie (k) e valore

1100|0 1100|0 01|00 01|10 1100|0 11|01||1 00|0 1100|0 00|0 10|0

le coppie (R_i, R_j) sono codificate da una tabella

I valori sono codificati in complemento ad:

SINTASSI

Imagine / Frame

header (dimensione, spazio di colore e codice) es YCbCr es 4:2:0

- Scan (es. R/G/B)

- header (identifica le componenti e le tabella di quantizzazione)

- Segment (divisione arbitraria per permettere la parallelizzazione)

- header (tabelle d. Huffman)

↳ Block

RISULTATI



original



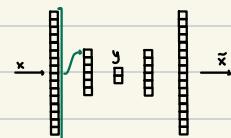
1.02 bpp 33.9 dB



0.21 bpp 29.5 dB

2.2.2 Reti Neurali

Recentemente si è iniziato ad applicare le reti neurali per le codificazioni immagine in quanto risultano essere più efficienti.



L'obiettivo è trovare la funzione non lineare in verde, ogni elemento dipende da tutti i bit dei livelli precedenti.

La funzione trovata dovrebbe minimizzare il valore $\|x - \hat{x}\|^2 + \lambda H(y)$
piccolo errore alta sparsificazione

Il contro sono il costo computazionale di training e la difficoltà di implementazione hardware di encoder/decoder

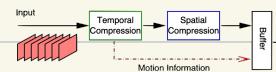
2.3 Compressione Video

Le compressioni video utilizzano molte delle strategie viste per le immagini, ma ci aggiunge la riduzione delle ridondanze temporali.

Sfrutta il fatto che frame successivi sono molto simili tra loro, si fa quindi una predizione

più accurata possibile del prossimo frame, prima di una successiva compressione frame-by-frame.

In particolare si può provare a prevedere il movimento degli oggetti e delle camere, l'errore di predizione sarà spesso e quindi facilmente codificabile.



$$\hat{f}_n(p) = f_{n+N}(p+u(p))$$

frame di riferimento

predizione del pixel p vettore di movimento
del frame n

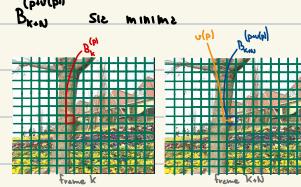
Il vettore di movimento è calcolato mediante block-matching:

dato 2 immagini divise in blocchi $B_n^{(p)}$ (blocco centrale nel frame), si cerca v t.c. la distanza del blocco $B_n^{(p)}$ e del blocco $B_{n+N}^{(p+v)}$ sia minima

In formula:

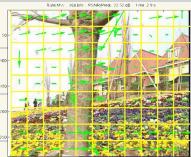
$$\text{Sia } d(v) = d(B_n^{(p)}, B_{n+N}^{(p+v)}) \text{ la distanza di due blocchi nel frame K e K+N al variare di } v$$

$$\text{Allora cerchiamo } v^* \text{ t.c. } v^* = \underset{v \in W}{\operatorname{arg\,min}} d(v)$$

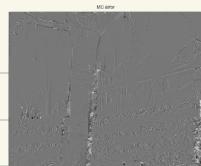


La codifica ha un costo spaziale (per memorizzare i vettori di movimento) e temporale (per cercare i vettori v^*)
questi costi si riducono con blocchi più grandi, oppure riducendo le dimensioni W (il costo di una minor fedeltà)
è inoltre possibile usare una strategia di ricerca mediamente più efficiente.

ES



vettori di movimento



errore di predizione

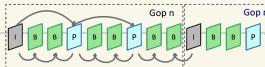
2.3.1 Group of Pictures

Ogni frame viene etichettato con una di 3 etichette.

- Intra-coded (I): la codifica avviene unicamente con le informazioni contenute nel frame.
- Predictive (P): codifica con predizione del movimento del precedente frame I o P. qualità elevate e piccole dimensioni, ma hanno elevata complessità.
- Bi-prediction (B): codifica a partire da 2 frame (il precedente ed il successivo) con predizione. prestazioni ancora migliori, ma complessità più elevate.

Il GOP è l'organizzazione dei frame in un gruppo

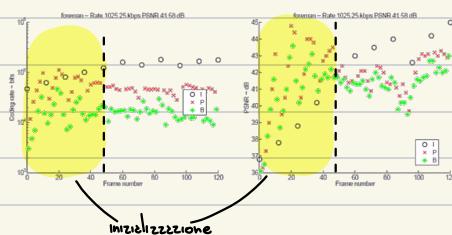
es.



oss I frame intra sono fondamentali per garantire:

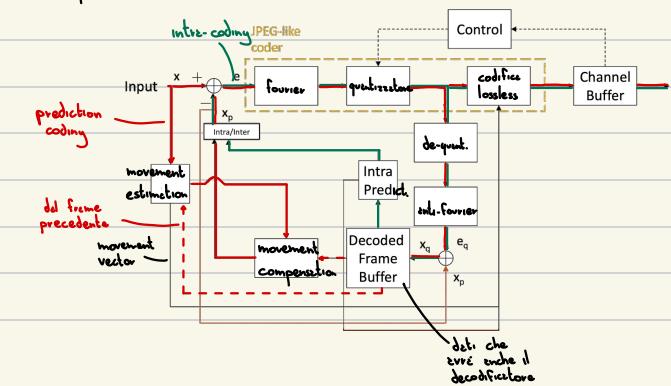
- accesso casuale (senza dover decodificare il video dall'inizio)
- correzione degli errori (senza un singolo bit errato può propagare l'errore all'intero video)

I frame intra richiedono più bit di codifica, quindi vengono utilizzati al minimo.



A regime i frame I sono 3-5 volte più grandi dei P e 10-20 volte i B. Questo per prestazioni (PSNR) simili.

ES - semplificazione in Intra/Inter



Io scelto di quanti frame usare di inter e di inter si fa per bilanciare bitrate e qualità.

2.3.2 Implementazioni

MPEG-2

È un codice video ibrido con frame I, P e B.

Supporta fino a $1920 \times 1080 @ 60\text{fps}$ e bitrate fino a 80 Mbps

Supporta YCbCr e ha 8 bit per campione per canale.

H.264

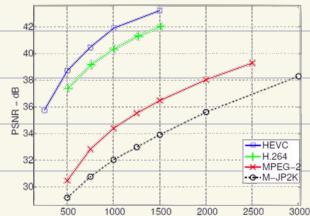
Castruisce su MPEG-2 con dei miglioramenti:

- Predizione spaziale nelle codifiche Intre (si prevede un blocco 8×8 da quelli adiacenti)
- trasformata DCT a coefficienti interi
- codifica lossless ottimizzata
- dimensione dei blocchi variabile per le stime del movimento

H.265 / H.266

Condividono la struttura, con miglioramenti incrementali

H.265 sta prendendo piede, ma ci sono ancora molti dispositivi che utilizzano H.264 e il supporto deve essere mantenuto per la retrocompatibilità



2.3.3 Codifica Video Scalabile

Vogliamo avere una codifica con un livello base che ha un bitrate piccolo, e poi le possibilità di aggiungere uno o più livelli enhancement che però richiedono un bitrate aggiuntivo.

Posso, ad esempio, cercare e codificare l'errore del video codificato a livello base.

In questo modo potrai trasmettere i tutti gli stessi pacchetti base, e solo chi ne ha bisogno, i pacchetti enhancement.

3. Architettura e Metriche di Rete

3.1 Architettura di Rete

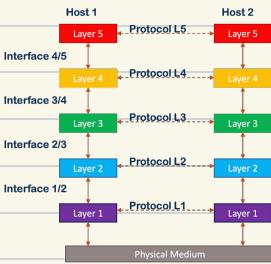
Come per ogni sistema complesso è utile organizzare le reti come pile di livelli.

Ogni livello utilizza i servizi forniti dal livello precedente e ne fornisce al successivo.

La comunicazione tra livelli in due macchine diverse avviene attraverso un protocollo.

ES

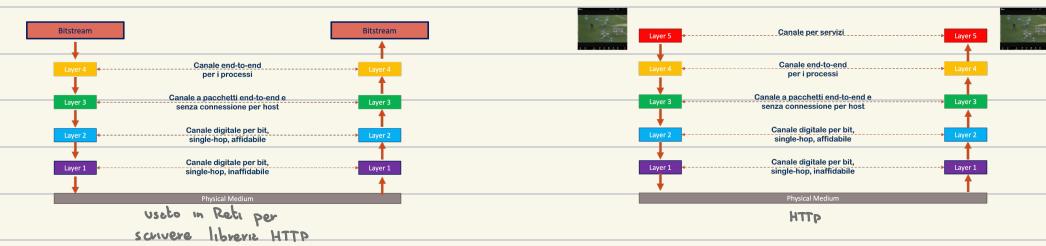
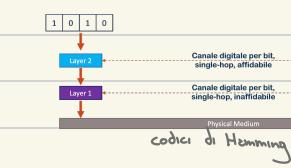
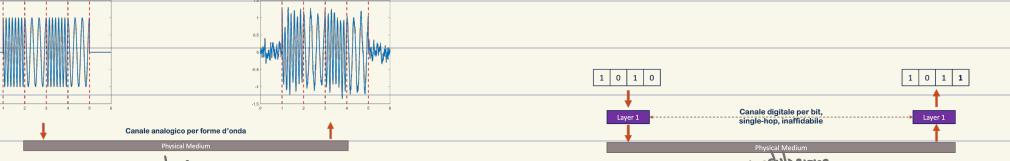
Il livello 1 si occupa della modulazione (e demodulazione) e fornisce il servizio di canale digitale.



I servizi sono offerti attraverso un'interfaccia.

L'insieme di regole che definiscono formato e semantica dei pacchetti scambiati dai livelli N di due macchine si dicono protocollo.

ES

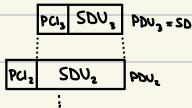


OSS

A livello teorico, i livelli superiori non dovrebbero conoscere o poter influenzare l'implementazione di un livello più basso. Questo non è però sempre vero, ad esempio è possibile pacchettizzare i video frame per frame.

Ogni livello passa al sottostante i suoi dati come protocol data unit PDU, la quale deve arrivare all'entità dello

Ogni livello aggiunge un header e passa i dati al livello inferiore.



In ricevimento ogni livello toglie un header e passa i dati al livello superiore.

I questo modo ogni livello riceve gli stessi dati, PDU o protocol data unit

SDU = service data unit

PCI = protocol control information, header

3.2 Indici di Prestazione di Rete

Sono metriche quantitative utilizzate per valutare le prestazioni di una rete di comunicazione.

3.2.1 BitRate

Def Il bitrate (o tasso, lunghezze di banda, capacità, velocità) è il numero di bit al secondo che si possono trasmettere

$$[\text{bit/s}] = [\text{bps}] = [\text{b/s}] \quad \text{oppure} \quad [\text{byte/s}] = [\text{B/s}]$$

$$1 \text{ B/s} = 8 \text{ b/s}$$

Def solitamente si indica con R_o il numero di bit al secondo che passano al livello fisico

Def solitamente si indica con throughput S_n il numero di bit al secondo che passano al livello n-2 ($S_n \leq R_o$)

$$S_{n+1} = S_n \cdot \frac{|\text{SDU}_n|}{|\text{PDU}_n|} = S_n \cdot \frac{|\text{SDU}_n|}{|\text{PCI}_n + |\text{SDU}_n||}$$

$$\text{quindi l'efficienza del livello è } \eta = \frac{S_{n+1}}{S_n} = \frac{|\text{SDU}_n|}{|\text{PDU}_n|} = \frac{|\text{SDU}_n|}{|\text{PCI}_n + |\text{SDU}_n||} = \frac{|\text{SDU}_n|}{|\text{PDU}_n|}$$

Def solitamente si indica con goodput il valore medio di S_{app} (throughput a livello applicativo)

ES

$$S_n = 100 \text{ Mbps}$$

$$|\text{PDU}_{n+1}| = 1200 \text{ byte}$$

$$|\text{PCI}_n| = 60 \text{ byte}$$

$$S_{n+1} = S_n \cdot \frac{|\text{PDU}_{n+1}|}{|\text{PDU}_{n+1}| + |\text{PCI}_n|} = 100 \text{ Mbps} \cdot \frac{1200}{1260} = 95.24 \text{ Mbps}$$

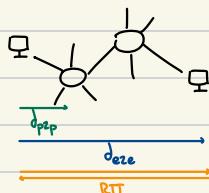
3.2.2 Ritardo

Def Il ritardo medio è il tempo medio che pesa dell'inizio di trasmissione di un pacchetto al suo ricevimento completo.

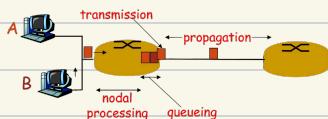
Si divide in d_{p2p} : ritardo tra due nodi

d_{zero} : ritardo end-to-end (somma dei ritardi p2p)

RTT: tempo di andata e ritorno (round trip time) ($\approx 2d_{\text{zero}}$)



$$\text{In dettaglio: } d_{\text{p2p}} = d_{\text{processing}} + d_{\text{queueing}} + d_{\text{transmission}} + d_{\text{propagation}}$$



- * ogni nodo fa 2 operazioni, tempo tipicamente trascurabile:
 - verifica gli errori (è conveniente controllare ad ogni nodo ulteriori per evitare)
 - determina il link di uscita
- * determinato dal bilancio delle connessioni, per trasmettere 1Mb in una rete a 1Mbps impiega 1s.
- * determinato dal tempo di propagazione fisica del segnale elettromagnetico, spesso trascurabile.

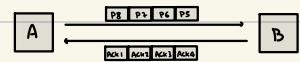
Def Il jitter è la varianza del ritardo medio, i problemi causati dal jitter si possono mitigare con un buffer.

Def Il prodotto banda ritardo (BDP) descrive la dimensione del buffer di trasmissione.

Questa dimensione è determinata dal tempo RTT. Infatti il trasmettitore deve salvare i pacchetti finché non riceve un acknowledgement dal ricevente.

Quindi $(\text{BDP})_{\text{bd}} = S \cdot \text{RTT}$ (si utilizza il throughput stazionario, S)

OSS In caso di connessione multi-hop va ovviamente considerato minS



3.2.2 Affidabilità

Def Sono indici di prestazione anche le probabilità di errore sul bit e le probabilità di perdere un pacchetto.

Un pacchetto può essere perso per le presenze di troppi errori o per troppo ritardo.

RIPASSO TELECOMUNICAZIONI

Questi errori avvengono a causa del canale, modellabile come un canale binario senza memoria.

Possano essere mitigati grazie alle codifiche di canale (ad es. i codici di Hamming)

prob. tutti i bit corretti

Probabilità che almeno un bit su un pacchetto di L bit sia errato: $1 - (1-\epsilon)^L$

Probabilità di avere esattamente k bit errati: $\binom{L}{k} \epsilon^k (1-\epsilon)^{L-k}$

CANALE CON ERRORI IN BURST

OSS Nei casi reali, a causa di fenomeni fisici, gli errori spesso sono temporalmente concentrati.

Si introduce il canale con errori in burst per modellare questo (es. ogni 70 bit 10 bit consecutivi errati)

Questo comportamento non è ben gestito dai codici di Hamming, è necessario aggiungere l'interleaving.

Dati:

1	2	3	4	8	9	10	11	15	16	17	18
---	---	---	---	---	---	----	----	----	----	----	----

Hamming (4,7)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----

Interleaving

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----

Trasmissione

CANALE CON ERRORI IN BURST

De-Interleaving

1	8	15	2	9	16	3	10	17	4	11	18	5	12	19	6	13	20	7	14	21
---	---	----	---	---	----	---	----	----	---	----	----	---	----	----	---	----	----	---	----	----

Hamming

1	8	15	2	9	16	3	10	17	4	11	18	5	12	19	6	13	20	7	14	21
---	---	----	---	---	----	---	----	----	---	----	----	---	----	----	---	----	----	---	----	----

SVANTAGGI

Diminuisce l'overhead, si introduce complessità in trasmissione e ricevimento e si introduce ritardo (in particolare con l'interleaving)

ES

Inviio di 1000 pacchetti di 500 bit su un canale binario simmetrico senza memoria con $\epsilon = 2 \cdot 10^{-6}$ e $R = 10 \text{ Mbps}$

Q1: tempo di trasmissione / Q2: tasso errore pacchetti / Q3: probabilità di ricevimento del messaggio

$$Q1. \frac{1000 \cdot 500}{10 \text{ Mbps}} = \frac{5 \cdot 10^5 \text{ b}}{10 \cdot 10^6 \text{ bps}} = \frac{1}{20} \text{ s} = 0.05 \text{ s} = 50 \text{ ms}$$

$$Q2. \text{PER} = 1 - (1 - \epsilon)^{500} = 9.905 \cdot 1^{-4}$$

$$Q3. (1 - \text{PER})^{1000} = 0.3679 = 36.79\%$$

Introduco un codice di canale in grado di correggere 1 errore che aggiunge 11 bit a pacchetto

$$Q1. \frac{1000 \cdot 511}{10 \text{ Mbps}} = \frac{1 \cdot 10^3 \cdot 511 \text{ b}}{10 \cdot 10^6 \text{ bps}} = 0.0511 \text{ s} = 51.1 \text{ ms}$$

$$Q2. \text{PER} = P(0) + P(1) = (1 - \epsilon)^{511} + \binom{511}{1} \epsilon (1 - \epsilon)^{500} = 5.209 \cdot 10^{-7}$$

$$Q3. 99.9479\%$$

È molto conveniente inserire un codice di canale.

PACKET DROPPING

Quando il buffer è pieno il nodo deve scegliere di scartare alcuni pacchetti:

- Drop tail policy: se il buffer è pieno i pacchetti sono ignorati.
 - Early dropping: se il buffer è pieno per $\frac{2}{3}$ danni i pacchetti sono casualmente ignorati
- oss I pacchetti possono essere ritrasmessi da un nodo precedente

Def Packet Loss Rate PLOSS e Packet Delivery Ratio PDR = 1 - PLOSS rappresentano la percentuale di pacchetti persi o consegnati.

3.3 Protocolli

Viste l'eterogeneità delle applicazioni multimediali è necessario scegliere correttamente i protocolli dei vari livelli.

3.3.1 TCP vs UDP

I due protocolli di trasporto forniscono un canale per flussi di bit end-to-end (il percorso da seguire è nascosto) e per processi (un processo sorgente trasmette ad un processo destinatario).

- Il protocollo TCP è orientato alle connessioni, ovvero fornisce un servizio reliable con connection handshake, ordinamento e ritrasmmissione dei pacchetti e controllo degli errori.
- Il protocollo UDP è invece connectionless, best-effort, che fornisce pochi servizi fornendo però un canale e bassa latenza e flessibile (perché la trasmissione è gestibile dall'applicazione)

3.3.1 RTP

Al di sopra dei protocolli di trasporto si creano dei protocolli specifici per i servizi multimediali.

- RTP (Real-time Transfer Protocol):

- Si usa spesso sopre ad UDP, mette in sequenza i pacchetti e ne monitora la latenza.
- Supporta il multicast (invia lo stesso stream a più destinazioni)
- Il Secure RTP (SRTP) supporta anche una crittografia real-time

- RTCP (RTP Control Protocol): solo monitoraggio delle qualità del servizio

oss webRTC è un framework javascript che permette di creare trasmissioni sicure RTC che con altri protocolli.

oss Il SIP è un protocollo che permette la comunicazione in tempo reale tra più host (es. Voip o videochiamate)

4. Quality of Experience

Fino adesso abbiamo visto una serie di metriche oggettive sulle trasmissioni multimediali, queste sono dette Quality of Service.

La Quality of Experience si basa invece su metriche soggettive degli utenti.

L'obiettivo finale sarà trovare una correlazione tra una metrica oggettiva e l'esperienza soggettiva così da poter applicare l'ottimizzazione matematica.

4.1 Esperimenti

Per ottenere dati soggettivi è necessario fare un esperimento affidabile e riproducibile:

- grande numero di utenti
- Laboratorio standard (schermo, luminosità, distanze di osservazione)
- Dati set abbazienza grande e vario (ovvero con immagini/video con diverse quantità di informazione spaziale e temporale)

4.1.1 Informazione Spaziale

Misura le quantità di dettagli in un'immagine, più alto per scene più complesse.

Si calcola un'approssimazione del gradiente noto come filtro di Sobel:

Il gradiente di un pixel G è $G = \sqrt{G_x^2 + G_y^2}$.

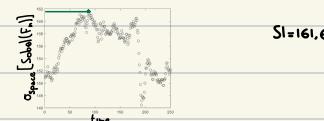
G_x è il gradiente orizzontale  dato il pixel x è $\frac{1(l_{1,1}) + 2(l_{0,1}) + 1(l_{-1,1}) - (l_{1,-1}) - 2(l_{0,-1}) - 1(l_{-1,-1})}{4}$

G_y è il gradiente verticale 

$$SI = \max_{time} \left\{ \sigma_{space} [Sobel(F_t)] \right\}$$

massimo su tutti i frame

es



4.1.2 Informazione Temporale

Misura le quantità di cambiamenti nel tempo di un video, più alto per sequenze con elevato movimento.

$$M_n(i,j) = F_n(i,j) - F_{n-1}(i,j) \quad (\text{sottrazione pixel a pixel tra due frame successivi})$$

$$TI = \max_{time} \left\{ \sigma_{space}[M_n(i,j)] \right\}$$

4.1.3 Procedura dell'Esperimento

1. screening iniziale: per eliminare soggetti non adatti (es persone affette da daltomismo)

2. sessione di training: sessione separata da quelle di test

3. sessione di test: si ignorano i primi risultati, per stabilizzare le opinioni

→ Singolo stimolo

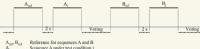


veloce

poco sensibile alle distorsioni

sensibile al contenuto

→ Absolute Category Rating



lento

sensibile alle distorsioni

meno sensibile al contenuto

→ Pair Comparison



lento

sensibile alle distorsioni

meno sensibile al contenuto

oss Vai considerato l'ambiente del test, se è controllato è più accurato, ma è più difficile trovare partecipanti.

In ambiente non controllato è più semplice trovare partecipanti, ma è meno preciso.

Infine si assume che la distribuzione dei dati sia gaussiana e si possono trovare medie e accuratezze di essa.

4.2 Metriche

Def Sia $I(n,m,c,t)$ l'intensità del pixel (nm) al tempo t nella componente c (es pixel (100,200) a t=0 nella componente 'red')

Def L'errore quadratico medio delle componenti di luminanza Y_{Cbc} tra I e \hat{I}

$$\text{MSE}_Y(t) = \frac{1}{NM} \sum_{n,m} (I(n,m,2,t) - \hat{I}(n,m,2,t))^2$$

quindi

$$\text{PSNR}_Y(t) = 10 \log_{10} \frac{255^2}{\text{MSE}_Y(t)} \approx 48 \text{dB} - 10 \log_{10} \text{MSE}_Y(t)$$

Sì definiscono similmente

$$\text{MSE}_{Cb}(t) / \text{MSE}_{Cr}(t) = \frac{1}{\frac{N}{2} \frac{M}{2}} \sum_{n,m} (I(n,m,2/3,t) - \hat{I}(n,m,2/3,t))^2$$

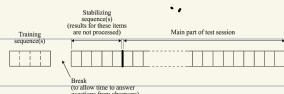
$\text{PSNR}_{Cb}(t)$, $\text{PSNR}_{Cr}(t)$

Quindi:

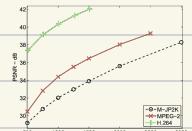
$$\text{PSNR}_{YCbCr}(t) = \frac{3}{4} \text{PSNR}_Y(t) + \frac{1}{8} \text{PSNR}_{Cb}(t) + \frac{1}{8} \text{PSNR}_{Cr}(t)$$

Infine

$$\text{PSNR}_{YCbCr} = \frac{1}{T} \sum_{t=1}^T \text{PSNR}_{YCbCr}(t)$$



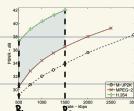
Possiamo ora plottare il PSNR al variare del bitrate per varie codifiche:



Def Si definisce Bjontegard Delta PSNR la differenza media di PSNR tra due codifiche

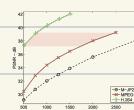
$$\text{BDPSNR} = \frac{1}{R_2 - R_1} \int_{R_1}^{R_2} [f_A(R) - f_B(R)] dR, \quad [R_1, R_2] \text{ intervallo in cui ci sono valori}$$

f_A e f_B ottenute per interpolazione



Def Si definisce Bjontegard Delta Rate la differenza media di tasso tra due codifiche

Si calcola con un integrale, simile a quanto visto sopra



oss L'obiettivo di questi valori è quello di codificare la QoE in un numero.

Il modello PSNR ha tante limitazioni. Non tiene conto di molti effetti visivi percepiti dall'uomo.

SSIM e VMAF sono più accurate, non li svilupperemo matematicamente.

5. Streaming Adattivo

Seppiamo che la QoE migliora all'aumentare del bitrate R_c del video

Vogliamo quindi trovare un bitrate R_c del video più grande possibile, rimanendo però al di sotto di S .

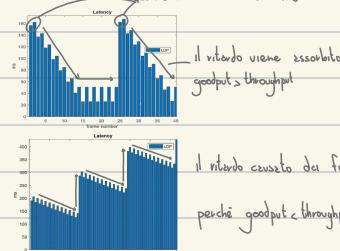
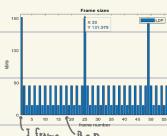
$$R_c \leq S$$

oss R_c è molto variabile nel tempo secondo la codifica GOP, i frame intra richiedono molti più bit dei B o P.

oss R_c non può semplicemente essere impostato, si utilizza un algoritmo evристico che opera sui parametri del codificatore finché trova il valore più vicino ad R_c

oss Anche S è variabile nel tempo, soprattutto nel caso di connessioni mobili.

ES-GOP



Il ritardo causato dal frame intra non viene assorbito
perché goodput > throughput

5.1 Interventi di Riduzione del Throughput

5.1.1 R_c minimo

Codifica il video mantenendo R_c costante e t.c. $R_c \leq S_{min}$. (funziona solo se S_{min} esiste e $S_{min} \approx S$)

soluzione semplice

qualità buone, soprattutto per gli utenti con $S > S_{min}$

5.1.2 Router

Se in un collegamento ScRc il router abbandona alcuni pacchetti casualmente.

soluzione semplice

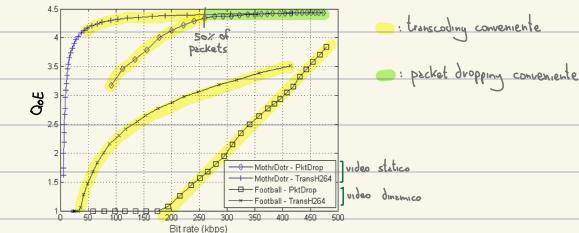
risultato imprevedibile sulla QoE (ci sono pacchetti meno importanti che andrebbero scartati prima)

5.1.3 Transcoding

Se in un collegamento ScRc il router decodifica il video e lo ricodifica a qualità inferiore.

soluzione efficace per l'utente

soluzione costosa (router specifici e complessi)



5.1.4 Scalable Video Coding

Il video viene codificato dal server ad un certo livello base con dei pacchetti enhancement.

I router scartano prima i livelli enhancement

degrado della QoE sotto controllo

I router fanno un'operazione semplice

e per bloccare le qualità risulta più bassa per inefficienza delle codifiche e più livelli,

richiede comunque l'aggiornamento dei router

5.1.5 Adaptive Bitrate

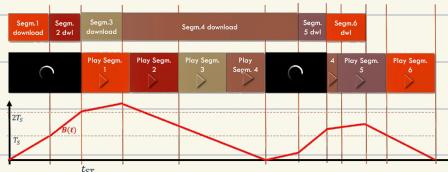
L' soluzione più utilizzata, client e server si accordano in modo dinamico sul bitrate del video.

Il server mette a disposizione molte codifiche possibili. Il client può scegliere dinamicamente la versione.

Le scelte avvengono indipendentemente per ogni segmento di 1-10s del video e avvengono tramite richieste HTTP

Il client costruisce e mantiene un playout buffer per ridurre i freeze.

$B(t)$ è la dimensione (in secondi) di tale buffer. Quando il buffer si svuota avviene il rebuffering, la riproduzione si ferma finché il buffer non è abbastanza pieno.



5.2 QoE

La QoE nel servizio di streaming è definita da:

1. qualità video

nel segmento n : $Q(n) = f(R_c(k_n)) = R_c(k_n) = \lambda_n k_n$

2. numero e durata degli eventi di rebuffer.

Indichiamo con Δ_n la durata del rebuffer dopo il segmento n ($\Delta_n=0$ se non c'è rebuffer)

$$\phi(\Delta_n) = \begin{cases} 0 & \text{se } \Delta_n = 0 \\ \Delta_n b & \text{se } \Delta_n > 0 \end{cases}$$

← eventi anche brevi infissoano molto sulle prestazioni

3. numero e ampiezza delle variazioni di qualità.

$$\lambda_2 |k_n - k_{n-1}|$$

4. tempo del buffer iniziale (t_{ST})

$$J(n) = \lambda_n k_n - \lambda_2 |k_n - k_{n-1}| - \phi(\Delta_n)$$

↓ diminuisce col cambio di qualità
↓ aumenta all'aumentare della qualità
↓ diminuisce con i rebuffer

$$\text{Quindi su un video: } J = \sum_{n=1}^N J(n) - \lambda_3 t_{ST}$$

5.3 Algoritmo Adaptive Bitrate

Il client deve scegliere la qualità del prossimo segmento, k_n , con lo scopo di massimizzare J .

Conosciamo k_{n-1} , B_0 (dimensione del buffer), una stima di throughput \hat{s} e la durata del segmento T_s .

```

J_min = inf

for each l of range(1..K)
    R = R(1)
    β = S/R - 1

    if β >= 0
        Δ = 0
    else
        t_o =  $\frac{B_0}{1-\frac{S}{R}} = \frac{RB_0}{R-S}$  // (B_0 + β t_o = 0)

    if t_o < T_s
        Δ = 0
    else
        Δ = (T_s + M T_s)R/S

    J_k = λ_l l - λ_e |l - K_m| - φ(Δ)

    if J < J_k
        K_m = 1
        J = J_k

```

K_m è la qualità scelta dell'algoritmo

Un algoritmo come questo assieme allo standard DASH permettono lo streaming adattivo.
DASH stabilisce come il server deve comunicare i codec disponibili ai client. Inoltre stabilisce lo standard delle seguenti comunicazioni client-server.