

Università degli studi di Padova

Corso di ingegneria dell'informazione

Laboratorio di ingegneria informatica

# Link application

Anno accademico 2015-2016

Davide Talon 1075692

Copyright © 2016 Davide Talon

LABORATORIO DI INGEGNERIA INFORMATICA, UNIVERSITÀ DEGLI STUDI DI PADOVA, ITALIA.

[HTTPS://BITBUCKET.ORG/DAVIDETALON/LINK](https://bitbucket.org/davidetalon/link)

[HTTPS://GITHUB.COM/DAVIDETALON/LINK](https://github.com/davidetalon/link)

Progetto per il laboratorio di ingegneria informatica, corso tenuto dal professore Nicola Ferro. Svolto nel secondo semestre dell'anno accademico 2015 - 2016.



## Contents

|          |                                   |           |
|----------|-----------------------------------|-----------|
| <b>1</b> | <b>Introduzione</b>               | <b>5</b>  |
| 1.1      | Contesto                          | 5         |
| 1.2      | Motivazioni                       | 5         |
| 1.3      | Obiettivi                         | 6         |
| 1.4      | Link utili e installazione        | 6         |
| 1.4.1    | Licenza                           | 7         |
| <b>2</b> | <b>Progettazione</b>              | <b>9</b>  |
| 2.1      | Idea iniziale                     | 9         |
| 2.2      | Architettura                      | 9         |
| 2.2.1    | Socket                            | 10        |
| 2.2.2    | Sleave                            | 10        |
| 2.2.3    | Master                            | 11        |
| 2.2.4    | Compressione dati                 | 12        |
| 2.2.5    | Interfaccia utente                | 12        |
| <b>3</b> | <b>Sviluppo e implementazione</b> | <b>15</b> |
| 3.1      | Inizio dello sviluppo             | 15        |
| 3.2      | Implementazione del codice        | 15        |
| 3.2.1    | Scoprire i master nella rete      | 15        |
| 3.2.2    | Connessione TCP                   | 17        |
| 3.2.3    | Ricezione del file                | 18        |
| 3.2.4    | Argp e il parsing dei comandi     | 19        |

|            |  |           |
|------------|--|-----------|
| 3.2.5      | Chiamate di sistema . . . . .            | 21        |
| <b>4</b>   | <b>Valutazione e collaudo .....</b>      | <b>23</b> |
| <b>4.1</b> | <b>Testing sul campo</b>                 | <b>23</b> |
| <b>4.2</b> | <b>Valutazione e bug riscontrati</b>     | <b>25</b> |
| <b>5</b>   | <b>Conclusioni e lavoro futuro .....</b> | <b>27</b> |
| <b>5.1</b> | <b>Lavoro futuro</b>                     | <b>27</b> |
| <b>5.2</b> | <b>Conclusioni</b>                       | <b>27</b> |



# 1. Introduzione

## 1.1 Contesto

Seguendo il corso Laboratorio di ingegneria informatica il cui obiettivo è l'acquisizione di competenze di programmazione nel linguaggio C per applicarle a casi di interesse industriale ci è stato assegnato il seguente compito:

Realizzazione di un progetto individuale su un argomento di proprio interesse.

Pertanto, pensando ai possibili sviluppi, mi sono imbattuto in un problema pratico. Quante volte ci è capitato, in casa o in ufficio, di dover scambiare dei file tra computer vicini e di utilizzare le classiche introvabili chiavette o l'email il cui upload è lentissimo? Nonostante l'avanzare della tecnologia, infatti, non disponiamo ancora di un metodo semplice e veloce per lo scambio di file tra computer vicini e, in particolare, per computer connessi alla stessa rete. Gli utenti Apple già conoscono l'utilità di Airdrop che consente di inviare documenti, immagini e video ai computer vicini creando una rete ad hoc, ma perchè non estendere questa funzionalità a Linux, comprendendo dunque tutti i sistemi operativi Unix-like? Nasce, dunque, la necessità di avere una applicazione multi-piattaforma che permette, in modo facile e veloce, di inviare file da un computer a un altro.

## 1.2 Motivazioni

Con l'occasione del Laboratorio di ingegneria informatica e la possibilità di sviluppare un progetto inherente ai propri interessi nasce l'applicazione Link che permette, a due computer connessi alla stessa rete, di scambiarsi dei file. Questo progetto, infatti, parte principalmente dalla propria passione per internet e le telecomunicazioni, un mondo che mi appassiona e entusiasma facendomi capire la potenza delle nuove tecnologie. Tante volte mi è capitato di dover inviare dei file da un computer ad un altro e mi sono sempre chiesto quale fosse un metodo pratico e veloce per farlo, quindi ciò che ha spinto allo sviluppo di questo progetto è la ricerca di una soluzione personale per risolvere un problema che si incontra spesso nella vita quotidiana. Da qui l'idea di utilizzare la connessione locale per lo scambio di file.

### 1.3 Obiettivi

L'obiettivo primario del progetto è quello di sviluppare un'applicazione per inviare qualsiasi tipo di file tra computer connessi alla stessa rete e far in modo che tale applicazione sia disponibile per tutti i sistemi operativi Unix-like. Le caratteristiche principali che il prodotto finale dovrà avere saranno intuitività nell'utilizzo, velocità di trasmissione e leggerezza:

**Intuitività** : l'applicazione sviluppata dovrà essere estremamente intuitiva e di facile comprensione, pertanto ci limitiamo a pochi comandi essenziali autoesplicativi.

**Velocità** : l'invio dei file deve avvenire in pochi semplici passaggi e potrà sfruttare la potenza della connessione locale.

**Leggerezza** : il programma sarà estremamente leggero e permetterà di inviare file di dimensioni arbitrariamente grandi senza doverli completamente caricare nella RAM, non influenzando le prestazioni generali del sistema durante l'esecuzione.

Inoltre, seguendo la propria passione per le telecomunicazioni, tra gli obiettivi di questo progetto vi è quello di crescere nella conoscenza delle reti e dei socket oltre che nell'intero mondo Unix. Volendo schematizzare gli obiettivi prefissati, si ha:

- Permettere a un utente di inviare files da un computer a un altro.
- Creare una applicazione multi-piattaforma disponibile per i SO Unix-like.
- Semplicità e intuitività nell'utilizzo dell'applicazione.
- Acquisire conoscenze di programmazione in C
- Crescita personale nella conoscenza delle reti e del loro funzionamento.

### 1.4 Link utili e installazione

L'intero progetto, comprensivo di report finale e slides di presentazione è disponibile sia su Bitbucket <https://bitbucket.org/davidetalon/link> sia su GitHub alla pagina <https://github.com/davidetalon/link> da cui si possono scaricare i file sorgenti e quanto necessario per l'intallazione dell'applicazione.

In particolare per l'intallazione dell'applicazione:

1. Crea la cartella dove installare link e spostati al suo interno

```
$ mkdir link
$ cd link
```

2. Scarica la repository da GitHub

```
$ git clone https://github.com/davidetalon/link.git
```

3. Compila tramite Cmake (è richiesta almeno la versione 3.4)

```
$ cmake .
$ make
```

4. Ti suggerisco di aggiungere link al path di sistema

```
$ PATH=<path to link>:$PATH
```

5. Ora puoi conoscere link utilizzando il suo help

```
$ link --help
```

#### 1.4.1 Licenza

Tutto il codice è disponibile sotto la licenza Apache 2.0 "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND pertanto puoi sentirti libero di modificarlo e ridistribuirlo. Puoi ottenere una copia della licenza al link

<http://www.apache.org/licenses/LICENSE-2.0>





## 2.1 Idea iniziale

Deciso di sviluppare un software per lo scambio di file tra computer connessi alla stessa rete, si è passati alla fase di progettazione dell'applicazione Link. L'idea di fondo è quella di creare un unico software che funzioni sia per la ricezione che per l'invio di dati, pertanto il software dovrà essere in grado di agire sia come server, ovvero in attesa di ricevere delle richieste, sia come client che richiede la connessione. L'invio di un file deve avvenire in pochi semplici passaggi; Link permette di cercare tutti i dispositivi che sono in ascolto nella rete locale e lascia all'utente la scelta del destinatario. Dunque, scelto il ricevente, l'applicazione si occuperà, tramite le varie chiamate di sistema, di comprimere i dati e di eseguire tutti i controlli necessari per la trasmissione dei dati.

## 2.2 Architettura

Come già sottolineato la piattaforma deve essere in grado sia di interpretare il ruolo di server per poter ricevere i dati in ingresso, sia il ruolo di client per poter inviare i dati. Pertanto, per comprendere meglio il funzionamento dell'applicazione sono state definite due modalità operative:

**Master mode** : è la condizione di ricezione e attesa, infatti, la piattaforma si mette in ascolto su una specifica porta UDP e aspetta di ricevere una richiesta da parte di qualche mittente, ricevuta la richiesta, apre un socket TCP e legge i dati in ingresso.

**Slave mode** : è la modalità con cui è possibile inviare un file, infatti, entrati in modalità slave si cercano tutti i master in ascolto all'interno della rete e si trasmettono i dati.

Volendo rendere l'applicazione disponibile per i sistemi operativi Unix-like la gestione delle connessioni è ricaduta sui socket di C offerti dalle API di Unix i quali hanno permesso di gestirle in modo piuttosto efficiente e di avere abbastanza libertà di decisione.

Il problema principale nella progettazione dell'architettura software è stata la non conoscenza della rete, infatti, generalmente, un server possiede un Ip statico, ma cosa succede con l'utilizzo del moderno DHCP in cui viene assegnato, all'interno della rete, un Ip casuale? E se la rete cambia?

Pertanto si è pensato a un modo per interrogare tutti i master disponibili sulla rete inviando un

messaggio broadcast.

Per quanto riguarda la compressione, invece, si è scelto, per inter-compatibilità dei vari sistemi operativi di utilizzare tar disponibile, in varie versioni, su tutto il mondo Unix.

Analizziamo, nel dettaglio il funzionamento della piattaforma nelle prossime sezioni.

### 2.2.1 Socket

Per comprendere a fondo come è stata progettata l'applicazione è necessario, innanzitutto, comprendere cosa sono i Socket e il loro funzionamento. I socket fanno parte della IPC (interprocess communication facilities) di Unix e permettono lo scambio di dati, oltre che sulla stessa macchina, anche tra più host connessi alla rete. Supponiamo di trovarci davanti alla classica architettura Client/Server:

- Ogni applicazione, sia lato server, che lato client, apre un socket ovvero una interfaccia che permette alle due applicazioni di comunicare l'una con l'altra
- Il server binda il socket su indirizzo conosciuto in modo che il client possa connettersi a lui

Pertanto una trasmissione dati, a livello software, è caratterizzata da una coppia di socket che identificano rispettivamente il client e server. Nella piattaforma Link saranno necessari principalmente due tipo di Socket:

- SOCK\_STREAM i socket di tipo stream ci permettono un canale di comunicazione affidabile, orientato alla connessione e byte-stream, ovvero:  
**Affidabile** viene garantito che i dati arrivino intatti al destinatario, ovvero così come sono stati trasmessi  
**Orientato alla connessione** viene garantito che i vari messaggi arrivo al destinatario nello stesso ordine in cui sono stati inviati  
**Byte-stream** non vi sono limiti di dimensioni alla quantità di dati che possono essere inviati
- SOCK\_DGRAM i socket di tipo datagram ci permettono di inviare dati in modo più veloce, senza però assicurarci che la trasmissione dei dati sia corretta, essi infatti possono arrivare sbagliati al destinatario, non arrivare o arrivare in ordine diverso da quello di partenza

In particolare verranno utilizzati i socket di tipo AF\_INET che permette lo scambio di dati tra host connessi in rete tramite il protocollo *Ipv4*.

Per bindare il socket del server, ovvero assegnargli un indirizzo noto dovremmo assegnargli una precisa porta in cui ascoltare, mentre per quanto riguarda l'indirizzo Ip, possiamo assegnargli tranquillamente INADDR\_ANY ovvero l'interfaccia di loopback che permette di indirizzare il socket su tutte le interfacce di rete disponibili.



Uteriori informazioni sui socket e il loro funzionamento, insieme ad innumerevoli esempi possono essere trovati alla pagina <http://http://www.beej.us/guide/bgnet/> e su *The linux programming interface: A Linux and UNIX System Programming Handbook*, M. Kerrisk, Nostark, 2010.

### 2.2.2 Slave

Come già anticipato, nella modalità slave è possibile inviare i file al destinatario scelto, per far questo entrati nella slave mode la piattaforma apre un socket UDP sulla porta 1234 e invia un messaggio in broadcast sulla porta 1235 interrogando tutti i Link master in ascolto sulla rete locale.

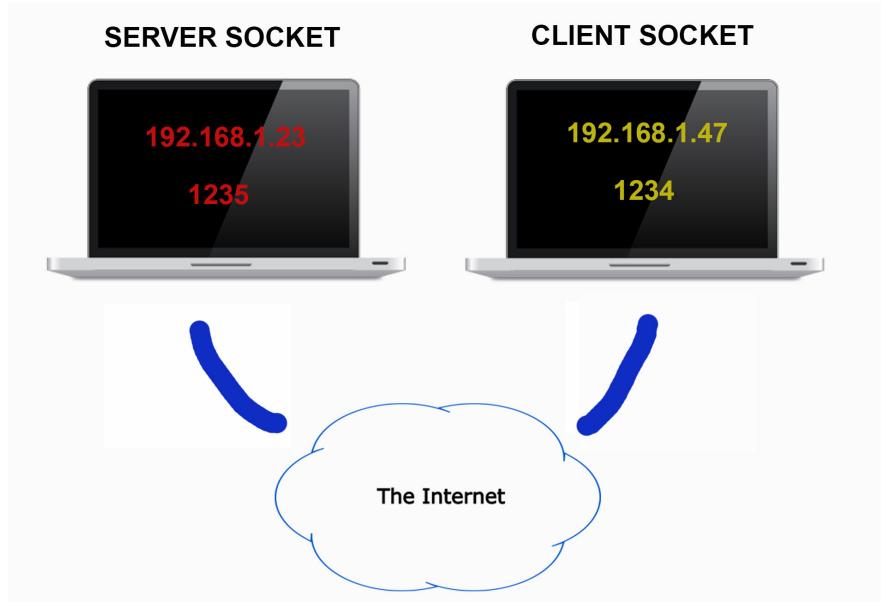


Figure 2.1: Princípio di funzionamento dei socket, la trasmissione è identificata da una copia di socket.

Viene, infatti, inviata la richiesta LINKAPP/CLNTRQT/SRVON?/ e si mette in ascolto delle eventuali risposte da parte dei master disponibili. Verificata la correttezza della risposta, dunque, tutti i master validi vengono messi in un array contenente il loro nome e l'indirizzo del loro socket. A questo punto l'applicazione, elencando i vari master disponibili nella rete, permette, in modo interattivo, di scegliere il destinatario e di inviare i file. La piattaforma si preoccupa, quindi, di preparare il file da trasmettere comprimendolo come specificato nei prossimi paragrafi. In particolar modo, conoscendo l'indirizzo del ricevente, lo slave apre un socket TCP sulla porta 2345, effettua una richiesta di connessione al master scelto e, inviando nei primi byte trasmessi il messaggio LINKAPP/SLVNAME/<nomeslave>/FNAME/<nomefiledainviare>, chiede se accetta di ricevere il file. Vengono dunque raccolte informazioni sul file da trasmettere e in particolare la dimensione del file, la quale viene inviata al master connesso. Il file viene dunque aperto in lettura e inviato allo slave senza caricarlo interamente in memoria centrale ma limitandosi a un buffer di 4 MB.

### 2.2.3 Master

Per quanto riguarda la modalità master, inizialmente viene aperto un socket UDP il quale si mette in ascolto sulla porta 1235 di eventuali richieste da parte degli slave. Ricevuta e validata la richiesta, il master, dunque, risponde al richiedente con il messaggio di servizio LINKAPP/SRVON/<nomemaster>. Quindi, dopo aver inviato le proprie informazioni, la piattaforma apre un socket TCP sulla porta 2346 e resta in attesa delle richieste di connessione. Stabilita la connessione, viene ricevuta la richiesta del file da trasmettere e una, volta validata, viene lasciata la libertà all'utente di accettare o meno lo scambio di file. In caso di esito negativo, il master invia il messaggio LINKAPP/SEND/NOTACCEPTED/ e si occupa della chiusura della connessione mentre, in caso contrario, si procede con la trasmissione dei dati rispondendo con LINKAPP/SEND/ACCEPTED/. A questo punto viene ricevuta la dimensione del file trasmesso e si continua a ricevere il file finché non è stato interamente ricevuto, ovvero finché il numero di byte letti in entrata non corrisponde alla dimensione del file.

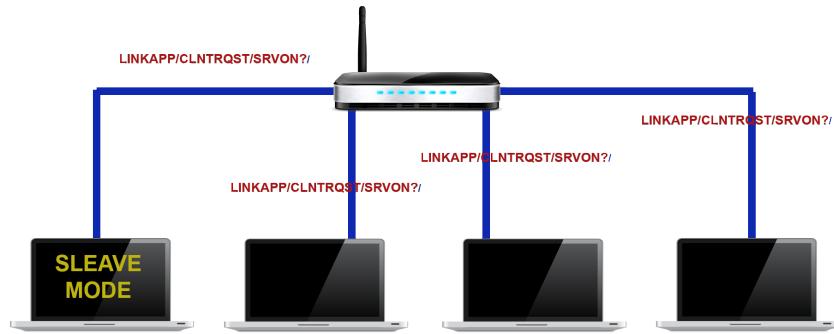


Figure 2.2: Lo slave invia un messaggio in broadcast per conoscere la rete in cui si trova.

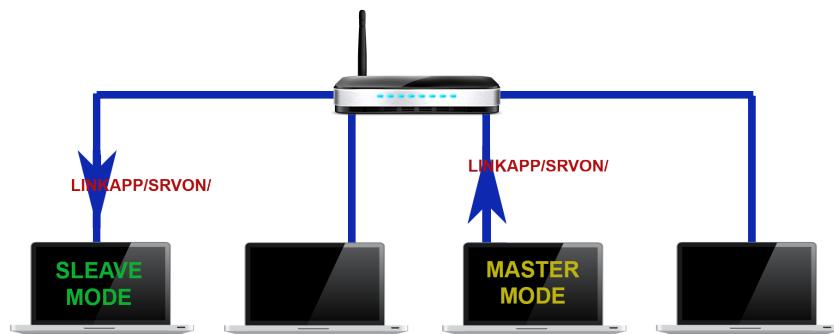


Figure 2.3: I master disponibili rispondono alla richiesta dello slave confermando di essere attivi.

#### 2.2.4 Compressione dati

Per semplicità e, affinchè fosse possibile comprimere, non solo un singolo file, ma anche una cartella con tutto il suo sottoalbero si è evitato di utilizzare `zlib` (solitamente utilizzato in C) ma il meccanismo di compressione dati è affidato al sistema operativo, in particolare si è scelto di effettuare delle chiamate al sistema per invocare `tar` applicazione che permette di comprimere file, cartelle e disponibile in tutti i sistemi Unix-like. Dopo svariate ricerche, si è scoperto che OS X presenta la versione `bsdtar`, differente dalla tradizionale GNU `tar`, ma comunque compatibile.

#### 2.2.5 Interfaccia utente

Tra gli obiettivi vi è quello di creare una interfaccia utente piuttosto semplice, pertanto, pur non essendoci una interfaccia grafica si è scelto di utilizzare dei comandi semplici e autoesplicativi attraverso il passaggio di parametri da riga di comando. Viene pertanto effettuato un parsing dei parametri utilizzando una versione minimizzata di Argp che permette di verificare la correttezza dei parametri passati e, in caso di errore, aiutare l'utente attraverso l' `--help`.

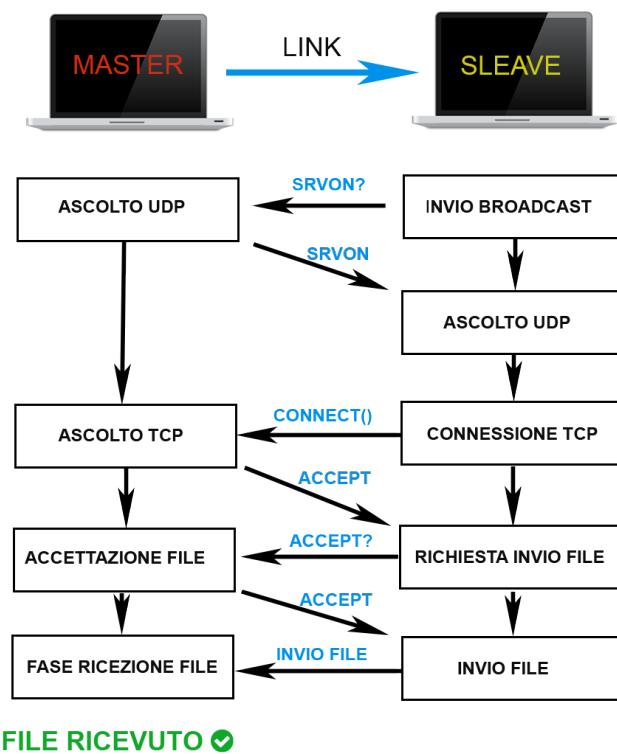


Figure 2.4: Schema del funzionamento della trasmissione dati tra master e slave.





### 3. Sviluppo e implementazione

#### 3.1 Inizio dello sviluppo

Trattandosi di uno dei primi grossi progetti personali sviluppati, ed essendo C un linguaggio a me nuovo, la fase di implementazione del codice ha richiesto una parte sostanziale del tempo dedicato al progetto. Ciò è stato principalmente dovuto a errori di inesperienza nella programmazione C e alla non comprensione della documentazione relativa ai socket di Unix portando più volte alla disperazione.

#### 3.2 Implementazione del codice

In questa sezione vengono documentate le parti più complicate dell'implementazione adottata, in particolar modo come scoprire i master nella rete, il parsing dei parametri da linea di comando, la compressione dei dati e l'utilizzo dei socket per la loro trasmissione.

##### 3.2.1 Scoprire i master nella rete

Una delle funzionalità chiave della piattaforma è sicuramente la possibilità, dopo aver mandato un messaggio broadcast all'intera rete, di trovare i master nelle reti locali pur non conoscendone la struttura. Questa funzionalità è offerta dalla funzione `srvsInNet()` presente in `sleave.c`.

Alla funzione viene passato il file descriptor `udpClntSock` del socket UDP aperto e un array vuoto di `srv` contenente informazioni sui socket dei master trovati nella rete. All'interno della funzione viene inizializzato un server fittizio e, successivamente, si imposta il socket in modalità non-blocking, ovvero l'esecuzione del programma non viene interrotta in attesa di dati messaggi in ingresso, proprio per questo viene impostato un tempo massimo per la ricerca.

```
int srvsInNet( const int udpClntSock , srv *srvs ) {  
  
    // number of server  
    int nSrvs = 0;  
  
    struct sockaddr_in currentAddr;
```

```

currentAddr.sin_family = AF_INET;
currentAddr.sin_port = htons(UDP_SERVER_PORT);
currentAddr.sin_addr.s_addr = htonl(INADDR_ANY);

int currentAddrLen = sizeof(currentAddr);

printf("Waiting_for_master_response ...\\n");

// set socket non blocking
fcntl(udpClntSock, F_SETFL, O_NONBLOCK);
int err;

// buffer created
char buffer[SERVICE_BUFFER_SIZE];

int exceededTime = time(0) + SERVER_SEARCH_TIME;

A questo punto, finchè il numero di master trovati è minore del numero massimo di quelli rilevabili e non si è superato il tempo massimo di ricerca, si rimane in attesa di messaggi da parte dei server disponibili, viene verificata la validità ed eventualmente si aggiungono tutte le informazioni del master trovato, comprensivo di indirizzo reale, nell'array srvs. Si conclude restituendo il numero di master trovati.

while ((nSrvs <= MAX_NUMBER_SERVERS) && (time(0) < exceededTime)) {
    // cleaning buffer
    bzero(buffer, SERVICE_BUFFER_SIZE);

    // checking received message
    recvfrom(udpClntSock, buffer, SERVICE_BUFFER_SIZE, 0, (struct sockaddr *)
              &currentAddr, &currentAddrLen);
    err = errno;

    // handle error from recvfrom
    if ((err != EAGAIN) && (err != EWOULDBLOCK)) {
        printf("recv_returned_unrecoverable_error(errno=%d)\\n", err);
        return -1;
    }
    char *strName;
    // checking received message
    if (strncmp(buffer, VALID_SERVER_ON, 14) == 0) {

        // getting server name
        strtok(buffer, "/");
        strtok(NULL, "/");
        strName = strtok(NULL, "/");
        srv currentSrv;
        strcpy(currentSrv.name, strName);
        currentSrv.sockAddr = currentAddr;
        currentSrv.sockAddrLen = sizeof(currentSrv.sockAddr);

        srvs[nSrvs] = currentSrv;
        nSrvs++;
    }
}

```

```
    return nSrvs;
```

### 3.2.2 Connessione TCP

La connessione TCP permette di assicurarci una trasmissione dei dati affidabile e orientata alla connessione (pertanto ordinata). Vediamo quindi come il master apre il socket TCP e si metta in ascolto di eventuali richieste. Ciò è disponibile in `master.c` e in particolare nelle funzioni `masterMode()` e `openTcpSrv()`.

Inizialmente nella funzione `masterMode()` viene inizializzata la struttura `sockaddr_in` contenente l'indirizzo del socket TCP del master, in particolare gli si assegna la famiglia `AF_INET` che permette lo scambio di dati in rete, la porta `TCP_SERVER_PORT = 2346` e gli si assegna l'interfaccia di loopbak (si imposta in ascolto in tutte le interfacce di rete disponibili). Successivamente si crea il socket di tipo `SOCK_STREAM` effettuando i vari controlli sull'avvenuta creazione:

```
// server TCP address
struct sockaddr_in tcpSrvSockAddr;
memset(&tcpSrvSockAddr, 0, sizeof(tcpSrvSockAddr));

tcpSrvSockAddr.sin_family = AF_INET;
tcpSrvSockAddr.sin_port = htons(TCP_SERVER_PORT);
tcpSrvSockAddr.sin_addr.s_addr = htonl(INADDR_ANY);

int tcpSrvSockAddrLen = sizeof(tcpSrvSockAddr);

// client TCP address
struct sockaddr_in tcpClntSockAddr;

int tcpClntSockAddrLen = sizeof(tcpClntSockAddr);

// create TCP server socket
int tcpSrvSock = socket (AF_INET, SOCK_STREAM, 0);

if (tcpSrvSock < 0) {
    perror("\nCannot create TCP server socket:");
    return -1;
}
```

Nella parte successiva, invece il socket viene bindato, ovvero si associa il socket a uno specifico indirizzo assegnato, in questo caso a `tcpSrvSockAddr`, e vi è una chiamata alla funzione `openTcpSrv()` passando come parametri il socket creato, il suo indirizzo comprensivo di dimensione, l'indirizzo e la dimensione del socket dello slave e infine la connessione accettata.

```
if (bind(tcpSrvSock, (struct sockaddr *) &tcpSrvSockAddr, tcpClntSockAddrLen) <
0) {
    perror("\nCannot bind TCP server socket:");
    return -1;
}

// create socket for connection
int connectionSock;
int tcpSrvIsOpen = openTcpSrv(tcpSrvSock, (struct sockaddr_in *)&tcpSrvSockAddr,
tcpSrvSockAddrLen, (struct sockaddr_in *)&tcpClntSockAddr, tcpClntSockAddrLen,
&connectionSock);
```

Nella funzione `openTcpSrv()` pertanto si pone il socket in ascolto imponendogli una backlog di 5, ovvero limitando il numero di connessioni in sospeso nella coda del socket. Successivamente, le richieste di connessione vengono accettate mediante la funzione `accept()` e vengono salvate in `connectionSock` permettendo dunque di iniziare la trasmissione dei dati.

```
int openTcpSrv(const int tcpSrvSock, struct sockaddr_in *tcpSrvSockAddr, const
    int tcpSrvSockAddrLen, struct sockaddr_in *tcpClntSockAddr, int
    tcpClntSockAddrLen, int *connectionSock) {

    printf("\nStarting_TCP_server...\n");
    fflush(stdout);
    // start listening with queue size of 5
    if(listen(tcpSrvSock, 5) < 0) {
        printf("Error_starting_listening\n");
        // exit(1);
    }

    // search for a connection request
    int request;
    while (1) {

        printf("ConnectionSock:_%d\n", connectionSock);
        // accept connection with sleave
        request = accept(tcpSrvSock, (struct sockaddr *) &
            tcpClntSockAddr, &tcpClntSockAddrLen);

        printf("ConnectionSock_after_accept:_%d\n", request);

        *connectionSock = request;

        if(connectionSock < 0) {
            perror("Connection_with_sleave_not_accepted:");
            return -1;
        }

        return 0;
    }
}
```

### 3.2.3 Ricezione del file

Sicuramente la ricezione del file rappresenta il cuore della piattaforma realizzata in quanto permette di raggiungere l'obiettivo primario, ovvero la trasmissione dei dati da un host a un altro. E' pertanto qui riportata la documentazione relativa alla funzione `receiveFile()` presente in `master.c`.

Dopo che l'utente ha accettato il file il master invia un messaggio allo slave comunicandogli di poter iniziare la trasmissione; viene ricevuta la dimensione del file da ricevere e successivamente, si apre, con lo stesso nome del file in ricezione, un file in modalità di scrittura.

```
int receiveFile (const int *connectionSock, const char *fileToReceive) {

    char buffer[DATA_BUFFER_SIZE];

    send(connectionSock, SEND_ACCEPTED, sizeof(SEND_ACCEPTED), 0);
```

```

printf("Server_response:%s\n", SEND_ACCEPTED);

// creating service buffer for receiving file size
char serviceBuffer[SERVICE_BUFFER_SIZE];
memset(&serviceBuffer, 0, sizeof(serviceBuffer));

// receiving buffer size
recv(connectionSock, serviceBuffer, sizeof(serviceBuffer), 0);
printf("buffer:%s\n", serviceBuffer);

// transform buffer into a int
int size = atoi(serviceBuffer);
printf("File_size_to_receive:%d\n", size);

// opening file
FILE *fpOutput;
fpOutput = fopen(fileToReceive, "w");

if(fpOutput == NULL) {
    printf("NONO\n");
}

```

Dunque si inizia a ricevere il file e si continua a leggere nel socket finchè i byte letti non corrispondono alla effettiva dimensione del file.

```

// byte read
int offset = 0;
int n;

// receiving message
while( n = read(connectionSock, buffer, sizeof(buffer)) > 0 && offset < size) {
    fwrite(buffer, DATA_BUFFER_SIZE, 1, fpOutput);
    // printf("%s", buffer);
    offset = offset + n;
}

return 0;

```

### 3.2.4 Argp e il parsing dei comandi

Grazie all'utilizzo di una piccola libreria basata su Argp di GNU è stato possibile effettuare, in modo elegante e sicuro, il parsing dei parametri da linea di comando, in particolare, esso, infatti, ci permette di costringere l'utente all'utilizzo dei comandi designati. Analizziamo nei dettagli la sua implementazione presente in [link.c](#).

Argp ci fornisce la struttura `argp_option` con la quale andiamo a specificare ogni parametro utilizzato

```

struct argp_option {
    char *name;
    char key;
    char *arg;
    int flags;
    char *doc;
    int group;
};

```

In cui:

`name` rappresenta il nome dell'opzione;  
`key` è la chiave che viene passata al parser per riconoscere l'opzione;  
`arg` è l'argomento dell'opzione (se ce ne sono);  
`flags` rappresenta i flag dell'opzione;  
`doc` è una stringa di documentazione per l'opzione.

E inoltre ci mette a disposizione la struttura `argp_state` dove vengono salvate le informazioni relative allo stato del parsing corrente.

Pertanto andiamo a definire le opzioni che accettiamo da linea di comando, ovvero:

```
static struct argp_option options[] = {
    {"verbose", 'v', 0, 0, "Produce_verbose_output" },
    {"listen", 'l', 0, 0, "Start_listening" },
    {"send", 's', "<FILENAME>", 0, "Send_file" },
    {"setname", 'n', "<NEWNAME>", 0, "Set_user_name" },
    {"getname", 'g', 0, 0, "Get_user_name" },
    {0}
};
```

Dove abbiamo definito:

`verbose` stampa i vari passaggi dell'operazione eseguita (attualmente di default);  
`listen` avvia la master mode;  
`send` seguito dall'argomento `<filename>` attiva la slave mode e invia il file avente nome `filename`;  
`setname` seguito dall'argomento `<newname>` imposta il nuovo nome utente `newname`;  
`getname` stampa sullo schermo il proprio nome utente.

Successivamente ci occupiamo di definire la struttura argomenti che verrà passata ad Argp e viene utilizzata dal main per comunicare con la funzione `argp_opt` che definiremo nel prossimo passaggio:

```
struct arguments {
    char *args[2]; /* arg1 & arg2 */
    int listen, verbose, getName;
    char *sendFile;
    char *newName;
};
```

Quindi abbiamo la funzione che, un parametro alla volta, verificare la correttezza di quanto passato e, in caso di riconoscimento setta le variabili corrispondenti, altrimenti fissa lo stato su `ARGP_ERR_UNKNOWN`. In caso, invece in cui il numero di parametri passati sia superiore rispetto a quelli dovuti verrà fissato lo stato `ARGP_KEY_ARG` mentre terminato il parsing `ARGP_KEY_END`

```
static error_t parse_opt (int key, char *arg, struct argp_state *state) {
    /* Get the input argument from argp_parse, which we
     * know is a pointer to our arguments structure. */
    struct arguments *arguments = state->input;

    switch (key) {
        case 'l':
            arguments->listen = 1;
            break;
        case 'v':
            arguments->verbose = 1;
```

```

        break;
    case 'g':
        arguments->getName = 1;
        break;
    case 'n':
        arguments->newName = arg;
        break;
    case 's':
        arguments->sendFile = arg;
        break;

    case ARGP_KEY_ARG:
        if (state->maxlen >= 2) {
            argp_usage(state);
        }

        arguments->args[state->maxlen] = arg;
        break;
    case ARGP_KEY_END:
        if (state->maxlen < 2) {
            argp_usage(state);
        }
        break;
    default:
        return ARGP_ERR_UNKNOWN;
}

return 0;
}

```

Andiamo dunque a costruire la struttura da passare ad argp per il parsing

```
static struct argp argp = { options, parse_opt, args_doc, doc };
```

In cui `options`, `parse_opt` sono stati già definiti mentre `args_doc` e `doc` sono delle stringhe contenenti rispettivamente informazioni sui parametri da passare e sull'applicazione.

Nella funzione `main()`, pertanto, dopo aver inizializzato la struttura `arguments`, effettuiamo la chiamata al parser di Argp

```
argp_parse(&argp, argc, argv, 0, 0, &arguments);
```

### 3.2.5 Chiamate di sistema

Un'altra parte interessante del codice sono sicuramente le chiamate a sistema che legano questa piattaforma ai sistemi operativi Unix-like. In particolare sono state effettuate due principali chiamate:

**Compressione** : per effettuare la compressione del file/cartella da inviare è stata effettuata una chiamata al sistema attraverso la funzione `system()` presente nella `stdio.h` la quale ha permesso di chiedere al sistema operativo dell'host di eseguire il comando desiderato. Avendo scelto di utilizzare `tar`, il comando `tarCommand` non è altro che la stringa "`tar -zcf <filename>.tar.gz <filename>`"

```
system(tarCommand);
```

**Verifica esistenza file** : per verificare l'esistenza del file/cartella sono stati utilizzati i comandi della bash [ -f <nomefile> ]/[ -d <nomecartella> ] ma, affinchè il risultato di questa chiamata fosse poi computabile è stato necessario aprire una pipe con la shell nella funzione `exists()`

Inizialmente, nella funzione `exists()`, è stata modificata la chiamata al sistema in base al tipo di file da inviare, infatti, nel caso in cui sia necessario effettivamente un file la chiamata al sistema dovrà essere [ -f <nomefile> ] mentre, nel caso in cui si debba inviare una cartella, dovrà essere [ -d <nomefile> ]. Pertanto il comando di sistema che verifica l'esistenza del file/cartella viene costruito in modo tale che la shell restituisca il carattere '1' in caso di esito positivo, '0 altrimenti'.

```
char exists(const char *fileName, char c) {

    if (c != 'f' && c != 'd') {
        printf("Argument not valid");
        return 0;
    }

    // construction of system call for verifying if file exists
    char existsCommand[EXISTS_COMMAND_SIZE];
    memset(&existsCommand, 0, sizeof(char) * EXISTS_COMMAND_SIZE);

    if (c == 'f') {
        strcpy(existsCommand, "[ -f ");
    } else {
        strcpy(existsCommand, "[ -d ");
    }

    strcat(existsCommand, fileName);
    strcat(existsCommand, "] ]&& echo 1 || echo 0");
}
```

Quindi, per poter utilizzare il risultato di tale chiama, vi è la necessità di aprire una pipe con la shell, effettuare un forking e reindirizzare l'output su un I/O stream attraverso la funzione `popen`.

```
// open pipe with unix shell
FILE *fp;
fp = popen(existsCommand, "r");

if (fp == NULL) {
printf("Failed to run command\n");
exit(1);
}

char result = fgetc(fp);
```



## 4. Valutazione e collaudo

### 4.1 Testing sul campo

La fase di sviluppo dell'applicazione è stata un susseguirsi di implementazione e testing delle nuove funzionalità. In particolare, sfruttando il fatto che master e slave lavorano con socket bindati a porte differenti, è stato possibile testare la piattaforma semplicemente utilizzando in parallelo due shell di comando in cui da una parte si attiva la master mode, mentre dall'altra la slave mode.

Riportiamo di seguito il risultato dei test principali eseguiti, comprensivi di messaggi utili per la comprensione di quanto l'applicazione sta realmente facendo, a sinistra abbiamo il master, mentre a destra lo slave:

```
MacBook-Pro-di-Davide-3:master  
davidetalon$ link --listen
```

Waiting **for** sleave ...

```
MacBook-Pro-di-Davide-3:sleave  
davidetalon$ link --send canzone.mp3
```

Mentre dal lato master viene aperto il socket Udp in ascolto per eventuali richieste, dal lato slave vengono inviati i pacchetti broadcast e si stabiliscono i server attivi all'interno della rete, dando la possibilità all'utente di scegliere il destinatario.

```
MacBook-Pro-di-Davide-3:master  
davidetalon$ link --listen
```

Waiting **for** sleave ... UDP request:  
LINKAPP/CLNTRQT/SRVON?  
Server response: LINKAPP/SRVON/ Tollo  
Starting TCP server...  
ConnectionSock: 1514626604

```
MacBook-Pro-di-Davide-3:sleave  
davidetalon$ link --send canzone.mp3  
Broadcast sent: LINKAPP/CLNTRQT/SRVON?  
Waiting for master response ...  
1 masters found  
0. Tollo  
Scegliere un master valido:
```

Ricevuta la richiesta da parte dello slave il master attiva il socket TCP e resta in attesa di eventuali connessioni, dall'altra parte lo slave si connette al master ed invia l'header contenente il proprio username e il nome del file da inviare.

```
MacBook-Pro-di-Davide-3:master
davidetalon$ link --listen

Waiting for sleave... UDP request:
LINKAPP/CLNTRQT/SRVON?
Server response: LINKAPP/SRVON/ Tacco
Starting TCP server...
ConnectionSock: 1514626604
ConnectionSock after accept: 6
Connection established
ConnectionSock: 6
Header: LINKAPP/SLVNAME/ Tacco /FNAME/
canzone.mp3.tar.gz/
Accept file canzone.mp3.tar.gz from
Tacco? (Y/N)
```

```
MacBook-Pro-di-Davide-3:sleave
davidetalon$ link --send canzone.mp3
Broadcast sent: LINKAPP/CLNTRQT/SRVON?
Waiting for master response...
1 masters found

0. Tacco
Scegliere un master valido: 0
Connnetcting to Tacco...
Connection enstabilished.
tarCommand command: tar -zcf canzone.mp3
.tar.gz canzone.mp3
Header sent: LINKAPP/SLVNAME/ Tacco /FNAME
/canzone.mp3.tar.gz/
Header size: 336
```

Grazie all'header precedentemente trasmesso il master ha le informazioni necessarie per chiedere all'utente se desidera accettare il file in ingresso, nella master mode, invece, una volta che il file è stato accettato, viene inviato con successo.

```
MacBook-Pro-di-Davide-3:sleave davidetalon$ link --send canzone.mp3
Broadcast sent: LINKAPP/CLNTRQT/SRVON?
Waiting for master response...
1 masters found

0. Tacco
Scegliere un master valido: 0
Connnetcting to Tacco...
Connection enstabilished.
tarCommand command: tar -zcf canzone.mp3.tar.gz canzone.mp3
Header sent: LINKAPP/SLVNAME/ Tacco /FNAME/canzone.mp3.tar.gz/
Header size: 336
Server response: LINKAPP/SEND/ACCEPTED/
Opening file canzone.mp3.tar.gz ...
size: 7210143
Remove command: rm -r canzone.mp3.tar.gz
File successfully sent.
```

Infine notiamo come il file sia realmente giunto a destinazione e sia valido, infatti è possibile estrarre il file.

```
MacBook-Pro-di-Davide-3:master davidetalon$ link --listen

Waiting for sleave... UDP request: LINKAPP/CLNTRQT/SRVON?
Server response: LINKAPP/SRVON/ Tacco
Starting TCP server...
```

```
ConnectionSock: 1514626604
ConnectionSock after accept: 6
Connection enstabilished
ConnectionSock: 6
Header: LINKAPP/SLVNAME/ Tacco /FNAME/ canzone.mp3.tar.gz /
Accept file canzone.mp3.tar.gz from Tacco? (Y/N) y

Receiving file canzone.mp3.tar.gz from Tacco ...
Server response:LINKAPP/SEND/ACCEPTED/
buffer: 7210143
File size to receive: 7210143
MacBook-Pro-di-Davide-3:master davide@talon$ ls
canzone.mp3.tar.gz
MacBook-Pro-di-Davide-3:master davide@talon$ tar -xzf canzone.mp3.tar.gz
MacBook-Pro-di-Davide-3:master davide@talon$ ls
canzone canzone.mp3.tar.gz
```

## 4.2 Valutazione e bug riscontrati

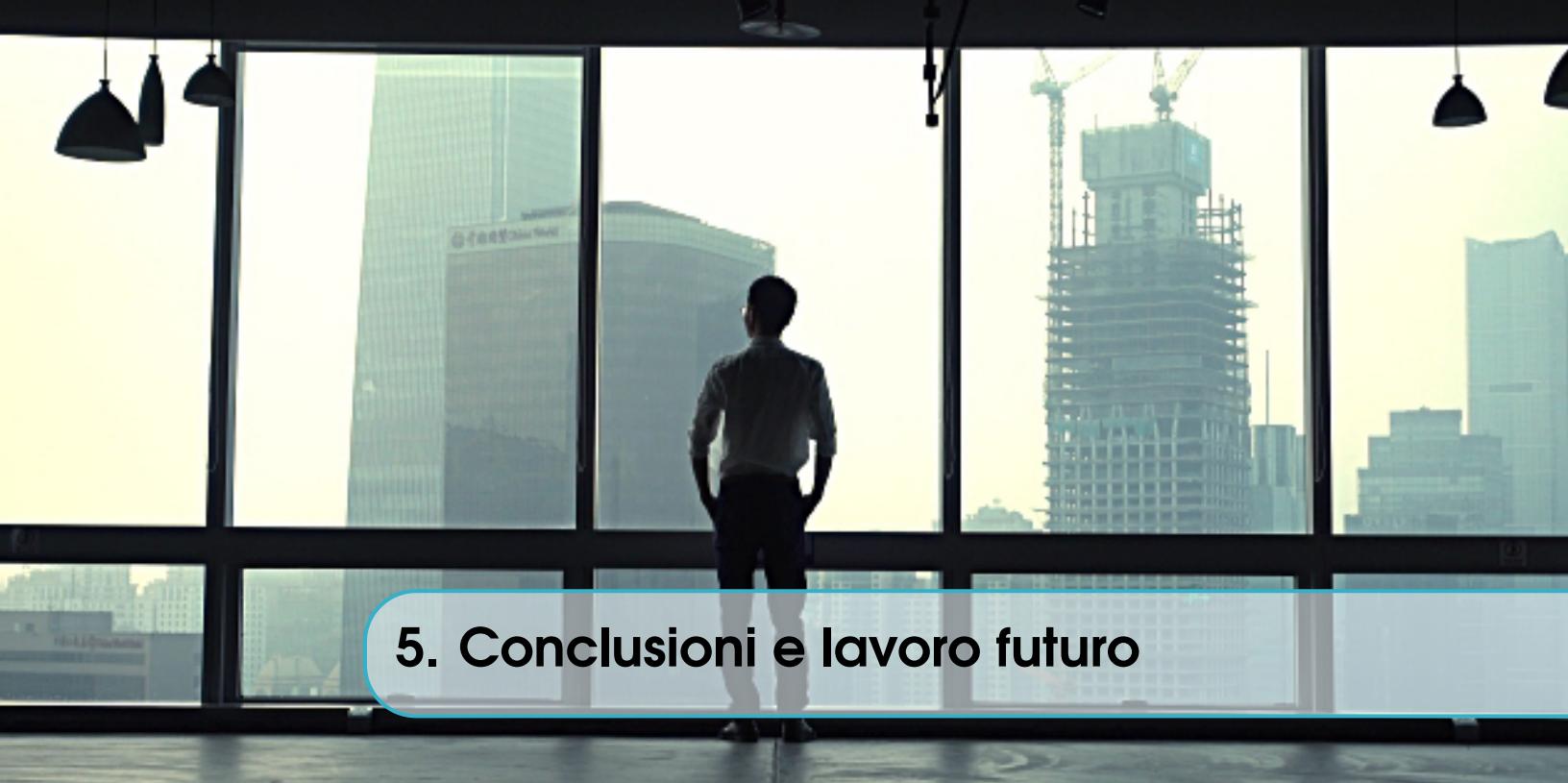
Osservando l'output dei terminali relativi allo slave e al master è possibile confermare la correttezza dei risultati e verificare il soddisfamento delle specifiche che avevamo prefissato come obiettivo. Tuttavia sono stati riscontrati alcuni bug che compaiono in determinate condizioni:

**Unrecoverable error** : alcune volte capita che, nel socket UDP dello slave, la `recvfrom()` dia "Unrecoverable error" `errno = 9 Bad file number` (motivo non ancora identificato);

**File name invalid encoding** : nella trasmissione dati da un sistema operativo OS X ad Ubuntu il nome del file assume una codifica non valida, tale problema è dovuto alla differente codifica dei caratteri nei due sistemi operativi;

**Address already in use** : lo slave tentando di connettersi al server (TCP) utilizza un indirizzo già in uso.





## 5. Conclusioni e lavoro futuro

### 5.1 Lavoro futuro

L'applicazione si trova tuttora a uno stato embrionale, infatti, possono essere fatte numerose interessanti modifiche che permettono di estenderne le funzionalità e di allargarne l'utenza. Alcune idee sono sicuramente:

**Correzione bugs** : risolvere i problemi attualmente riscontrati

**Porting su Windows** : uno dei prossimi obiettivi sarà sicuramente quello di rendere disponibile l'applicazione anche per utenti Windows permettendo dunque di raggiungere la gran parte del mercato. Ciò comporta un carico di lavoro abbastanza elevato,in quanto, non solo dobbiamo adattare i socket a Winsock ma dobbiamo adattare le varie chiamate del sistema, in particolare:

- la verifica dell'esistenza o meno del file da inviare
- la compressione dei dati da inviare, infatti Windows non fornisce una libreria per la compressione di dati, contrariamente a quanto avviene nei sistemi Unix-like

**Introduzione crittografia end-to-end** : è possibile introdurre una crittografia E2EE per rendere la trasmissione dei file sicura e impenetrabile contro i malintenzionati

**Più formati di compressione** : è inoltre doveroso implementare più sistemi di compressione per lasciare all'utente la possibilità di scegliere il formato di compressione desiderato

Si augura di poter continuare a lavorare sul progetto in quanto le soddisfazioni e le conoscenze tecniche aumentano in maniera esponenziale.

### 5.2 Conclusioni

Ci si può ritenere soddisfatti di quanto realizzato in questo progetto in quanto, oltre ad aver permesso di risolvere un problema quotidiano, ha dato l'occasione di accrescere le personali conoscenze sulla programmazione e in particolare sui principi fondamentali delle trasmissioni di dati. Durante la fase di progettazione e di implementazione sono state affrontate problematiche mai incontrate prima (un po' per le novità del linguaggio un po' per generale inesperienza) come ad esempio il parsing efficiente dei comandi da riga di comando o la necessità, in una trasmissione dati, di tener traccia di

quanto effettivamente inviato e ricevuto.

Inoltre confrontarsi con la documentazione scritta da altri programmati ha permesso di comprendere quanto sia importante, non solo commentare il codice, ma anche redarre una documentazione chiara e completa.

Si ringrazia in particolar modo <http://stackoverflow.com/> e tutta la sua community per tutti i preziosi consigli suggeritimi.

*Davide Talon*