

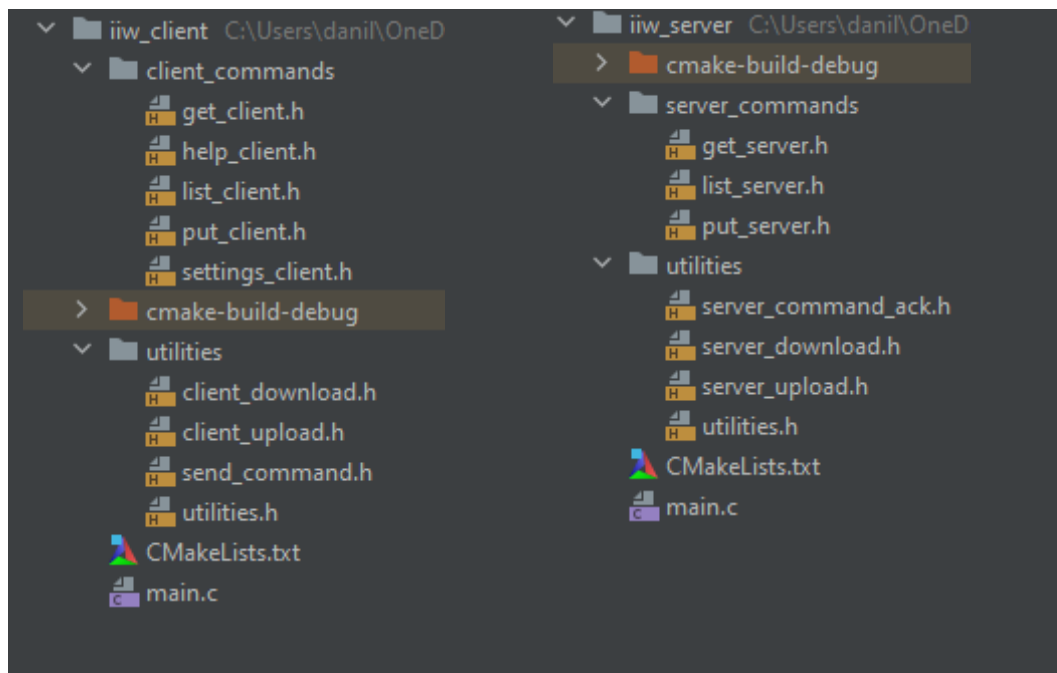
# Progetto B1

## Trasferimento file su UDP

D'Amico Danilo  
Ingegneria Internet e Web  
Anno Accademico 2019/2020

## Architettura del sistema e scelte progettuali effettuate

Il sistema si articola in due cartelle, *iiw\_client* ed *iiw\_server*, contenenti rispettivamente il codice necessario per eseguire il client e quello necessario per eseguire il server.



*Struttura del programma*

All'interno di queste due cartelle si trovano:

- un file *main.c* da compilare per eseguire il programma;
- una cartella *client\_commands/server\_commands* contenente i file di header dedicati ai singoli comandi eseguibili dall'utente;
- una cartella *utilities* contenente i file di header per la ricezione e l'invio dei dati ed un file *utilities.h* contenente le variabili globali ed altre funzioni utilizzate dagli altri file.

Si è optato per questa struttura con lo scopo di rendere quanto più modulare e leggibile l'articolazione del progetto.

I file *main.c* ricevono i comandi, li analizzano e li trasmettono ai file dei comandi omonimi, i quali chiameranno a loro volta le funzioni per la gestione della trasmissione necessarie alla loro esecuzione.

Quando un client invia un comando al server, invia in realtà una struct contenente tutte le costanti di comunicazione specificate dall'utente.

Il server, una volta ricevuto un comando sul socket d'ascolto, imita il funzionamento della funzione *accept()*, creando un socket connesso su cui gestirà la connessione con il client. Creato il nuovo socket, invia un pacchetto al client contenente la porta su cui gestirà la sua richiesta.

La connessione viene aperta tramite un handshake a due vie e chiusa mediante un handshake a tre vie.

## Implementazione

### Pacchetti per la comunicazione

Client e Server comunicano mediante due tipi di pacchetti implementati come struct e definite nel file *utilities.h*: *connectionPacket*, utilizzato per stabilire la connessione, e *goBackNPacket*, utilizzato per scambiare dati.

```
// pacchetto utilizzato per stabilire la connessione
struct connectionPacket {
    int type; //-1: Errore
    // 0: Comando o ACK

    char data[MAXLINE];
    int serverPort; // porta su cui il server stabilisce la connessione
    int numberOfPackets;
    int windowSize;
    int lossRate;
    int timeout;
    int adaptiveTimeout;
};

// pacchetto utilizzato per trasmettere dati
struct goBackNPacket {
    int type; //-1: Errore
    // 0: ACK
    // 1: Dati
    // 2: Comando & ACK Comando

    int sequenceNumber;
    int dataLength;
    char data[MAXLINE];
};
```

Nello stabilire la connessione, il client comunica al server i parametri specificati dall'utente durante la compilazione, mentre il server specifica la porta su cui ha creato un socket di connessione. Nel caso di comando *get* o *list*, il server si premurerà anche di comunicare il numero di pacchetti da trasmettere, mentre per il comando *put* avverrà il contrario.

## Gestione del segnale SIGALRM

Ogni volta che durante l'esecuzione del software viene invocato il segnale *alarm*, verrà prima chiamata la funzione *setTimeout()*, la quale ritorna il valore scelto dall'utente se il timeout in uso è statico, altrimenti calcola il valore del timeout adattivo.

```
int setTimeout() {
    int currentTimeout;

    if (adaptiveTimeout == 0)
        return timeout;
    else {
        if (attempt == 0) {
            currentTimeout = 1;
        } else {
            currentTimeout = 1 * attempt * 2;
        }
        return currentTimeout;
    }
}
```

Quando il timeout scade, viene invocata la funzione *alarmHandler()*, che riconosce il fallimento dell'attuale tentativo ed imposta la variabile *alarmTrigger* su 1.

```
void alarmHandler(int signo) {
    attempt++;
    alarmTrigger = 1;
}
```

## Perdita di pacchetti

Per simulare la perdita di pacchetti, la funzione *sendto()* viene invocata in uno statement if che calcola, mediante *calculateLoss()*, la probabilità che il pacchetto in oggetto venga invece scartato.

```

// introduciamo una perdita pacchetti
if (calculateLoss()) {
    sendto(listeningSocket, packet, sizeof(struct connectionPacket), 0, (struct sockaddr *) &serverAddress,
           sizeof(serverAddress));
} else {
    printf("Pacchetto scartato.\n");
}

```

```

int calculateLoss() {
    int randomPercentage = rand() % 100;
    if (lossRate > randomPercentage)
        return 0; // il mittente perde il pacchetto
    else
        return 1; // il mittente invia il pacchetto
}

```

## Invio comandi e ricezione ACK

I comandi vengono inviati dal client al server mediante la funzione *sendCommand()* presente in *utilities/send\_command.h* e ricevuti dal server nella funzione *main.c*.

*sendCommand()* ha la responsabilità di inviare il comando al server finché non riceve risposta o non conclude che la connessione sia troppo instabile perché possa avvenire la comunicazione.

Ad inviare la conferma di ricezione comando è la funzione del server *sendCommandACK()*, presente in *utilities/server\_command\_ack.h*. L'affidabilità della comunicazione è stavolta affidata alla funzione di upload o download che la invoca, la quale la invocherà nuovamente se il suo timer di ricezione sarà scaduto e non avrà ricevuto nessun pacchetto dati dal client.

Entrambe le funzioni, quando chiamate, inseriscono il valore della variabile globale *numberOfPackets* nel campo omonimo del *connectionPacket* che devono inviare. Nel caso di *get* o *list* sarà il client a conoscere la dimensione effettiva del file che si appresta ad inviare,

pertanto il server estrarrà il valore inviato da *sendCommand()* e lo utilizzerà durante la connessione. Durante l'esecuzione del comando *put* avverrà l'opposto.

## Download ed Upload dei file

La gestione del caricamento e scaricamento dei file è gestita dalle funzioni *clientDownload()*, *clientUpload()*, *serverDownload()* e *serverUpload()*.

Quando il client ha bisogno di scaricare dei dati dal server invoca la funzione *clientDownload()*, che ha la responsabilità di ricevere pacchetti fino a *numberOfPackets*, per poi invocare *closeClientDownloadConnection*, la quale ha la responsabilità di chiudere la connessione.

```
int clientDownload(int sockfd, struct sockaddr_in address/*indirizzo su cui ricevo pacchetti*/,
                  struct sockaddr_in connectedSocketAddress/*indirizzo cui inviare ACK*/, int fileDescriptor) {

    if (numberOfPackets == 0) {
        return 1;
    }

    struct goBackNPacket packet;

    socklen_t receiveAddressLength = sizeof(address);
    int lastPacketReceived = -1;

    while (1) {
        memset(&packet, 0, sizeof(packet));
        // attende finché non arriva un pacchetto
        recvfrom(sockfd, &packet, sizeof(packet), 0, (struct sockaddr *) &address, &receiveAddressLength);
        printf("ricevuto.\n");
        if (ntohl(packet.sequenceNumber) == (lastPacketReceived + 1)) {
            lastPacketReceived++;
            write(fileDescriptor, packet.data, ntohl(packet.dataLength));
        }

        // ACK
        memset(&packet, 0, sizeof(packet));
        packet.type = htonl(0);
        packet.sequenceNumber = htonl(lastPacketReceived);

        if (ntohl(packet.sequenceNumber) == (numberOfPackets - 1)) { // ho ricevuto l'ultimo pacchetto
            printf("Ho ricevuto l'ultimo pacchetto.\n");
            closeClientDownloadConnection(sockfd, packet, connectedSocketAddress, address);
            return 1;
        }
        if (calculateLoss()) {
            printf("Invio conferma del pacchetto %d di %d.\n", lastPacketReceived + 1, numberOfPackets);
            sendto(sockfd, &packet, sizeof(packet), 0, (struct sockaddr *) &connectedSocketAddress,
                  sizeof(connectedSocketAddress));
        } else {
            printf("Pacchetto scartato.\n");
        }
    }
}
```

Ad inviare i file al client è la funzione del server *serverDownload()*. Se la variabile *alarmTrigger* è un numero positivo, invia pacchetti fino a saturare la finestra di spedizione o fino a quando non raggiunge la fine del file interessato, dopodiché si pone in attesa avviando un timeout. Nel caso in cui il timeout scada senza che sia arrivato alcun pacchetto, conclude che il client non abbia ricevuto il primo dei pacchetti inviati e pertanto torna nella sezione di invio pacchetti. Nel caso sia arrivato un pacchetto, invece, si assicura che sia un ack di pacchetto che non ha ancora ricevuto, aggiorna di conseguenza la variabile *lastPacketReceived* e, nel caso tutti i pacchetti inviati siano stati confermati, invia nuovi pacchetti. Una volta che tutti i pacchetti inviati siano stati confermati invoca *closeServerDownloadConnection()* per porre fine alla connessione.

```
int serverDownload(int fd, int sockfd, struct sockaddr_in address /*indirizzo cui inviare dati*/, int connectedSocket,
                  struct sockaddr_in connectedSocketAddress /* indirizzo su cui ricevere dati*/) {
    struct goBackNPacket packet;
    socklen_t addressLength = sizeof(connectedSocketAddress);
    int lastPacketReceived = -1;
    ssize_t bytesRead;
    int sequenceNumberReceived;
    int currentTimeout;

    attempt = 0;

    sendCommandACK(connectedSocketAddress, address, sockfd);

    alarmTrigger = 1;
    sequenceBase = 0;

    while ((lastPacketReceived < (numberOfPackets - 1)) && (attempt < MAX_ATTEMPTS)) {

        // invio dati
        send_packets:

        if (alarmTrigger) {
            alarmTrigger = 0;
            // invia pacchetti fino a riempire la finestra
            for (int i = 0; i < windowSize; i++) {
                if (sequenceBase + i <
                    numberOfPackets) { // controllo di non andare oltre il numero previsto di pacchetti
                    memset(&packet, 0, sizeof(packet));

                    packet.type = htonl(1); // Dati
                    packet.sequenceNumber = htonl(sequenceBase + i);
                    lseek(fd, (sequenceBase + i) * MAXLINE, SEEK_SET);
                    bytesRead = read(fd, packet.data, MAXLINE);
                    packet.dataLength = htonl(bytesRead);

                    // introduciamo una perdita pacchetti
                    if (calculateLoss()) {
                        sendto(sockfd, &packet, sizeof(packet), 0, (struct sockaddr *) &address, sizeof(address));
                    } else {
                        printf("Pacchetto %d di %d scartato.\n", sequenceBase + i + 1, numberOfPackets);
                    }
                }
            }
        }
    }
}
```



```

// ricezione ACK
waitACK:

currentTimeout = setTimeout();
alarm(currentTimeout);

memset(&packet, 0, sizeof(packet));
while (recvfrom(connectionSocket, &packet, sizeof(packet), MSG_DONTWAIT,
    (struct sockaddr *) &connectionSocketAddress, &addressLength) <= 0) { // attesa risposta
    if (alarmTrigger) { // alarmTrigger è 1 se è scaduto il timeout
        alarm(0);
        if (attempt < MAX_ATTEMPTS) {
            sequenceBase = lastPacketReceived + 1; // inizio a inviare dall'ultimo pacchetto ricevuto

            // se non è mai arrivato un ack reinviemo la conferma del comando:
            if (lastPacketReceived == -1)
                sendCommandACK(connectionSocketAddress, address, sockfd);

            goto send_packets;
        } else {
            printf("La connessione con il client è instabile. L'operazione è stata annullata.\n");
            return -1;
        }
    }
}

sequenceNumberReceived = ntohs(packet.sequenceNumber);

//ACK di risposta ricevuto
if (ntohl(packet.type) == 0 && sequenceNumberReceived > lastPacketReceived) {
    alarm(0);
    attempt = 0;
    lastPacketReceived = ntohs(packet.sequenceNumber);

    if ((lastPacketReceived + 1) ==
        (sequenceBase + windowSize)) { // tutti i pacchetti inviati sono stati confermati
        sequenceBase = lastPacketReceived +
            1; // Dobbiamo iniziare ad inviare dal pacchetto successivo all'ultimo ricevuto
    }
}

```

```

//ACK di risposta ricevuto
if (ntohl(packet.type) == 0 && sequenceNumberReceived > lastPacketReceived) {
    alarm(0);
    attempt = 0;
    lastPacketReceived = ntohs(packet.sequenceNumber);

    if ((lastPacketReceived + 1) ==
        (sequenceBase + windowSize)) { // tutti i pacchetti inviati sono stati confermati
        sequenceBase = lastPacketReceived +
            1; // Dobbiamo iniziare ad inviare dal pacchetto successivo all'ultimo ricevuto
    }

    alarmTrigger = 1;
} else { // abbiamo ricevuto qualcosa ma non è un ack successivo
    goto waitACK;
}

// messaggio di conferma di fine comunicazione
memset(&packet, 0, sizeof(packet));
packet.type = htonl(0);
packet.sequenceNumber = htonl(numberOfPackets);

closeServerDownloadConnection(sockfd, packet, connectionSocketAddress, address);
return 1;
}

```

La logica di funzionamento dell'upload di un file è speculare a quella del download.

## Comando List

Il comando *list* è implementato come un caso particolare del comando *get*. Per conservare i dati ricevuti dal server, il client, nella funzione *listClient()*, crea un file temporaneo da consegnare a *clientDownload()*. Una volta che il file è stato riempito, ne stampa il contenuto a schermo e lo cancella chiudendolo.

```
void listClient(int listeningSocket, struct sockaddr_in serverAddress) {
    char buffer[MAXLINE] = {0};
    char list[] = "list ";
    struct connectionPacket *connectionData = sendCommand(listeningSocket, serverAddress, list);

    if (connectionData == NULL)
        return;

    //creo un file temporaneo in cui verrà scritto il contenuto ricevuto
    char template[] = "/tmp/list_client_temp_file_XXXXXX";
    int fileDescriptor = mkstemp(template);

    struct sockaddr_in connectedSocketAddress = calculateConnectedSocketAddress(serverAddress, *connectionData);
    numberOfPackets = ntohs(connectionData->numberOfPackets);

    if (clientDownload(listeningSocket, serverAddress/*indirizzo su cui ricevo pacchetti*/,
        connectedSocketAddress/*indirizzo cui inviare ACK*/, fileDescriptor)) {

        // stampiamo il contenuto del file temporaneo
        lseek(fileDescriptor, 0, SEEK_SET);

        printf("Elenco dei file disponibili sul server:\n");
        while (read(fileDescriptor, buffer, MAXLINE)) {
            printf("%s", buffer);
            memset(buffer, 0x0, MAXLINE);
        }
    } else {
        printf("non è stato possibile completare la richiesta.\n");
    }

    close(fileDescriptor);
}
```

Lato server, *listServer()* chiama il comando *ls* in un terminale creato da *popen()* per ottenere il contenuto della cartella indicata a tempo di compilazione per i file del server. Il file risultante da *popen()*, tuttavia, è una pipe, il che non consente di riportare all'indietro l'indice di lettura nel caso sia necessario reinviare dei pacchetti. Per ovviare a questo problema viene creato un file temporaneo con *mkstemp()* in cui copiare il contenuto della pipe.

```

void listServer(char *directory, int sockfd, struct sockaddr_in address) {
    static char buffer[MAXLINE] = {0};
    size_t readLength;
    char error[] = "errore nella lettura dei file disponibili.";

    chdir(directory);

    // apro un terminale in cui eseguire il comando ls e scrivendo l'output in filesList
    // popen crea una pipe, pertanto dovrò copiarne i contenuti in un file temporaneo se voglio usare fseek()
    FILE *filesList = popen("ls", "r");

    if (filesList == NULL) {
        fprintf(stderr, "%s\n", error);
        sendLastPacket(error, sockfd, address);
        return;
    }

    // copio filesList in un file temporaneo
    char template[] = "/tmp/list_server_temp_file_XXXXXX";
    int fileDescriptor = mkstemp(template);

    while ((readLength = fread(buffer, 1, MAXLINE, filesList)) != 0) {
        write(fileDescriptor, buffer, readLength);
        memset(buffer, 0x0, MAXLINE);
    }

    // calcolo numberOfPackets
    numberOfPackets = howManyPackets(fileDescriptor);

    // creo il socket connesso e salvo l'indirizzo della porta su cui è in ascolto per comunicarlo al client
    int connectedSocket = createConnectionSocket();
    struct sockaddr_in connectedSocketAddress = extractConnectionSocketAddress(connectedSocket);

    // trasferimento dati
    serverDownload(fileDescriptor, sockfd, address /*indirizzo cui inviare dati*/, connectedSocket,
        connectedSocketAddress /* indirizzo su cui ricevere dati*/);
}

```

## Limitazioni riscontrate

Sebbene la connessione tra client e server utilizzi il protocollo GoBackN per la trasmissione e ricezione dei dati, i due pacchetti per inviare il comando iniziale e ricevere la porta del socket connesso si comportano secondo un modello Stop and Wait.

## Piattaforma software usata per lo sviluppo ed il testing

Il progetto è stato svolto in ambiente Linux su Manjaro KDE Plasma 20.2, utilizzando come IDE per lo sviluppo ed il testing CLion 20.2.4 con una licenza Educational garantita dall'email universitaria.

## Esempi di funzionamento

I seguenti esempi sono screenshot catturati durante l'esecuzione dei comandi disponibili all'utente. Per le operazioni *put* e *get* sono stati inseriti solo i messaggi iniziali e finali di trasmissione dei pacchetti per migliorare la leggibilità.

```
/home/danilo/Scrivania/iw_client/cmake-build-debug/iw_client 127.0.0.1 /ho
Inserisci un comando o digita 'help' per la lista dei comandi disponibili.
help

'list'          l'elenco dei file disponibili sul server.
'get nomefile'  scarica dal server il file indicato.
'put nomefile'  carica sul server il file indicato.
'shutdown'      termina l'esecuzione del client
'settings'      imposta le variabili configurabili
Inserisci un comando o digita 'help' per la lista dei comandi disponibili.
list
Ho ricevuto l'ultimo pacchetto.
Elenco dei file disponibili sul server:
Introduzione al Web.pdf
Programmazione Socket 1.pdf
Programmazione Socket 2.pdf
test
Inserisci un comando o digita 'help' per la lista dei comandi disponibili.
|
```

*Esecuzione dei comandi "help" "list"*

```
test
Inserisci un comando o digita 'help' per la lista dei comandi disponibili.
get test
Invio conferma del pacchetto 1 di 94.
Invio conferma del pacchetto 2 di 94.
Invio conferma del pacchetto 3 di 94.
Invio conferma del pacchetto 4 di 94.
Invio conferma del pacchetto 5 di 94.
Invio conferma del pacchetto 6 di 94.
Invio conferma del pacchetto 7 di 94.
Invio conferma del pacchetto 8 di 94.
Invio conferma del pacchetto 9 di 94.
Invio conferma del pacchetto 10 di 94.
Invio conferma del pacchetto 11 di 94.
Invio conferma del pacchetto 12 di 94.
Invio conferma del pacchetto 13 di 94.
Invio conferma del pacchetto 14 di 94.
Invio conferma del pacchetto 15 di 94.
Invio conferma del pacchetto 16 di 94.
Invio conferma del pacchetto 17 di 94.
Invio conferma del pacchetto 18 di 94.
Invio conferma del pacchetto 19 di 94.
Invio conferma del pacchetto 20 di 94.
```

*Prima parte dell'esecuzione del comando "get"*

```
Invio conferma del pacchetto 80 di 94.
Invio conferma del pacchetto 81 di 94.
Invio conferma del pacchetto 82 di 94.
Invio conferma del pacchetto 83 di 94.
Invio conferma del pacchetto 84 di 94.
Invio conferma del pacchetto 85 di 94.
Invio conferma del pacchetto 86 di 94.
Invio conferma del pacchetto 87 di 94.
Invio conferma del pacchetto 88 di 94.
Invio conferma del pacchetto 89 di 94.
Invio conferma del pacchetto 90 di 94.
Invio conferma del pacchetto 91 di 94.
Invio conferma del pacchetto 92 di 94.
Invio conferma del pacchetto 93 di 94.
Ho ricevuto l'ultimo pacchetto.
Download completato con successo.
Inserisci un comando o digita 'help' per la lista dei comandi disponibili.
|
```

*Seconda parte dell'esecuzione del comando "get"*

```
/home/danilo/Scrivania/iw_client/cmake-build-debug/iw_client 127.0.0.1 /t
Inserisci un comando o digita 'help' per la lista dei comandi disponibili.
put test
pacchetti: 94
Invio il pacchetto 1 di 94.
Invio il pacchetto 2 di 94.
Invio il pacchetto 3 di 94.
Invio il pacchetto 4 di 94.
Invio il pacchetto 5 di 94.
Invio il pacchetto 6 di 94.
Invio il pacchetto 7 di 94.
Invio il pacchetto 8 di 94.
Invio il pacchetto 9 di 94.
Invio il pacchetto 10 di 94.
Invio il pacchetto 11 di 94.
Invio il pacchetto 12 di 94.
Invio il pacchetto 13 di 94.
Invio il pacchetto 14 di 94.
Invio il pacchetto 15 di 94.
Invio il pacchetto 16 di 94.
Invio il pacchetto 17 di 94.
Invio il pacchetto 18 di 94.
Invio il pacchetto 19 di 94.
Invio il pacchetto 20 di 94.
```

*Prima parte dell'esecuzione del comando "put"*

```
Invio il pacchetto 80 di 94.
Invio il pacchetto 81 di 94.
Invio il pacchetto 82 di 94.
Invio il pacchetto 83 di 94.
Invio il pacchetto 84 di 94.
Invio il pacchetto 85 di 94.
Invio il pacchetto 86 di 94.
Invio il pacchetto 87 di 94.
Invio il pacchetto 88 di 94.
Invio il pacchetto 89 di 94.
Invio il pacchetto 90 di 94.
Invio il pacchetto 91 di 94.
Invio il pacchetto 92 di 94.
Invio il pacchetto 93 di 94.
Invio il pacchetto 94 di 94.
Upload completato con successo.
Inserisci un comando o digita 'help' per la lista dei comandi disponibili.
|
```

*Seconda parte dell'esecuzione del comando "put"*

```
Process completed with 00000000.  
Inserisci un comando o digita 'help' per la lista dei comandi disponibili.  
settings  
  
'windowSize <integer>'      imposta il valore della finestra di spedizione.  
'lossRate <integer>'        imposta la probabilità di perdita pacchetto.  
'timeout <integer>'         imposta la durata del timeout.  
'adaptiveTimeout <integer>' attiva il timeout adattivo (1) o disattivalo (0)  
adaptiveTimeout 1  
Inserisci un comando o digita 'help' per la lista dei comandi disponibili.  
shutdown  
Il client sta per essere terminato.  
  
Process finished with exit code 0
```

*Esecuzione dei comandi “settings” e “shutdown”*

## Valutazione delle prestazioni

Per i seguenti test è stato utilizzato il pdf di presentazione del corso, un file da 94KB che il software divide in 94 pacchetti da 1KB ciascuno. Ogni test è stato ripetuto per cinque volte. Si riportano sia i risultati delle singole prove che la loro media.

I test sono stati condotti sul comando *get*, in quanto *list* non è che un suo sottocaso nel quale il pacchetto da scaricare è uno solo mentre *put* opera scambiando il ruolo del client e del server.

1. WindowSize: 1, LossRate: 0, Timeout: 1

- 0.007821
- 0.007634
- 0.008543
- 0.007676
- 0.009314

Media = 0.008197

2. WindowSize: 1, LossRate: 10, Timeout: 1

- 15.004044
- 20.004823
- 18.003952
- 18.004048
- 13.003933

Media = 16.80416

3. WindowSize: 1, LossRate: 10, Timeout: 10

- 200.002922
- 230.002951



- 220.004630
- 240.002983
- 140.004206

Media = 206.0035384

4. WindowSize: 10, LossRate: 0, Timeout: 1

- 0.012126
- 0.011583
- 0.010469
- 0.004657
- 0.010292

Media = 0.098254

Per i seguenti due test si è optato per una probabilità di perdita dei pacchetti di 30 invece che di 10 poiché quest'ultima opzione non produceva errori di trasmissione durante la transazione.

5. WindowSize: 10, LossRate: 30, Timeout: 1

- 6.003066
- 5.005652
- 4.002648
- 2.009660
- 3.007909

Media = 4.005787

6. WindowSize: 10, LossRate: 30, Timeout: 10

- 40.007803
- 0.007898
- 20.007791

- 10.004071

- 40.002909

Media = 22.006094

7. WindowSize: 1, LossRate: 10, Timeout: adattivo

- 17.059431

- 19.004064

- 24.006812

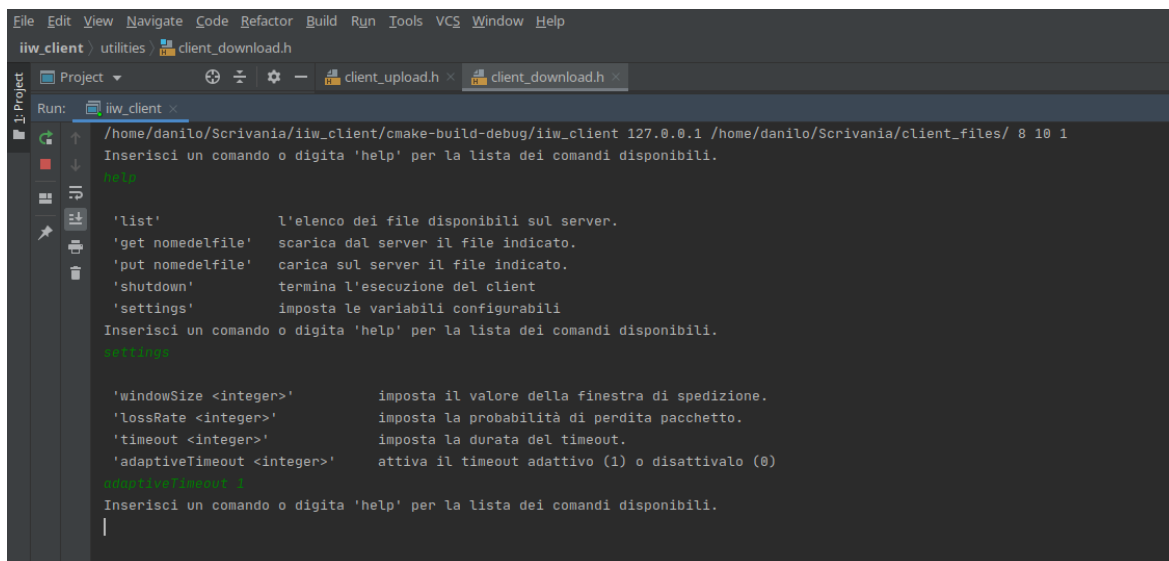
- 14.003588

- 19.004569

Media = 18.615692

## Manuale d'uso e d'installazione

Il progetto è diviso in due cartelle principali: *iiw\_client*, contenente il client, ed *iiw\_server*, contenente il server. È necessario eseguire il file *main.c* di entrambe le cartelle per poter eseguire il software.



```
File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
iiw_client utilities client_download.h
Project
Run: iiw_client x
/home/danilo/Scrivania/iiw_client/cmake-build-debug/iiw_client 127.0.0.1 /home/danilo/Scrivania/client_files/ 8 10 1
Inserisci un comando o digita 'help' per la lista dei comandi disponibili.
help
'list'          l'elenco dei file disponibili sul server.
'get nomefile'  scarica dal server il file indicato.
'put nomefile'  carica sul server il file indicato.
'shutdown'      termina l'esecuzione del client
'settings'      imposta le variabili configurabili
Inserisci un comando o digita 'help' per la lista dei comandi disponibili.
settings
'windowSize <integer>'  imposta il valore della finestra di spedizione.
'lossRate <integer>'    imposta la probabilità di perdita pacchetto.
'timeout <integer>'     imposta la durata del timeout.
'adaptiveTimeout <integer>'  attiva il timeout adattivo (1) o disattivalo (0)
adaptiveTimeout 1
Inserisci un comando o digita 'help' per la lista dei comandi disponibili.
|
```

*Il programma contiene comandi che ne spiegano l'utilizzo*

Segue l'elenco degli argomenti da immettere contestualmente all'esecuzione dei due file.

Per il file *main.c* contenuto nella cartella *iiw\_client*:

- Indirizzo IP del server;
- Fullpath della cartella con i file del client;
- Dimensione della finestra di spedizione;
- Probabilità di perdere i messaggi;
- Timeout.

Per il file *main.c* contenuto nella cartella *iiw\_server*:

- Fullpath della cartella con i file del server.

Una volta che il client è stato avviato è possibile specificare ulteriori impostazioni con il comando *settings*. Con questo comando è possibile cambiare il valore delle variabili finestra

di spedizione, probabilità di perdere i messaggi e timeout definiti in precedenza. In *settings* è anche possibile attivare un timeout adattativo per la comunicazione con il comando *adaptiveTimeout 1*.

Una volta che la comunicazione ha avuto inizio il client si occuperà di comunicare le variabili specificate al server contestualmente all'invio del comando digitato dall'utente.

Digitando il comando *help*, dal client sarà possibile vedere un elenco dei comandi disponibili accompagnati da una breve descrizione.