




















Progetto di Ingegneria di Internet e Web:  
Trasferimento file su UDP

Antonio Bernardini e Flavio Caporilli

# Indice

<b>1</b>	<b>Architettura del sistema e scelte progettuali</b>	<b>2</b>
1.1	Introduzione	2
1.1.1	Funzionalità del server 	2
1.1.2	Funzionalità del client 	3
1.1.3	Trasmissione affidabile 	3
1.2	Architettura del sistema 	4
1.2.1	Organizzazione del progetto	4
1.2.2	Protocollo Selective Repeat 	5
1.3	Scelte progettuali	6
1.3.1	Gestione della concorrenza tramite processi 	6
1.3.2	Bitmask 	7
1.3.3	Casi particolari 	7
1.3.4	Timeout di inattività 	7
1.3.5	Gestione degli errori e pacchetti persi 	8
1.3.6	Three-Way Handshake 	8
1.3.7	Progressive Bar 	9
<b>2</b>	<b>Descrizione dell'implementazione</b>	<b>10</b>
2.1	Implementazione del server	10
2.2	Implementazione del client	14
2.3	Implementazione del Three-Way Handshake	16
2.4	Implementazione del protocollo Selective Repeat	18
2.4.1	Implementazione del sender	18
2.4.2	Implementazione del receiver	22
2.5	Limitazioni riscontrate 	25
2.5.1	Calcolo del tempo di trasferimento 	25
2.5.2	Timeout sulla socket anziché sui singoli pacchetti 	26
<b>3</b>	<b>Manuale per l'uso </b>	<b>27</b>
3.1	Introduzione	27
3.1.1	Download e installazione	27
3.1.2	Disinstallazione	28
3.1.3	Configurazione del software	28
3.1.4	Esecuzione del software e comandi disponibili	29
3.2	Esempi di funzionamento	30
3.2.1	Esempio d'uso del comando <b>LIST</b>	30
3.2.2	Esempio d'uso del comando <b>GET</b>	32
3.2.3	Esempio d'uso del comando <b>PUT</b>	33

3.2.4	Esempio d'uso del comando <code>CLOSE</code>	34
<b>4</b>	<b>Valutazione delle prestazioni</b> 	<b>36</b>
4.1	Ambiente di test	36
4.1.1	Ambiente Linux	36
4.1.2	Ambiente MacOS ARM e Intel	36
4.2	Test effettuati	37
4.2.1	Test Timeout Adattivo	37
4.2.2	Test Timeout Statico	38
4.2.3	Test Timeout Statico  Timeout Adattivo	39
4.2.4	Test cumulazione degli errori	40
4.3	Test per l'integrità dei file trasferiti 	41
4.3.1	Esempio di funzionamento	41

# Capitolo 1

## Architettura del sistema e scelte progettuali

In questo capitolo vengono descritte le scelte progettuali e l'architettura del sistema implementato, con particolare enfasi sul protocollo di comunicazione.

### 1.1 Introduzione

Lo scopo del progetto è progettare ed implementare un'applicazione client-server per il trasferimento di file in linguaggio C, utilizzando l'API dei socket di Berkeley. L'applicazione dovrà impiegare il servizio di rete senza connessione, ossia il protocollo UDP (socket di tipo `SOCK_DGRAM`) per la trasmissione dei dati. Il software deve permettere:

- La connessione client-server senza autenticazione;
- La visualizzazione sul client dei file disponibili sul server tramite il comando `LIST` ;
- Il download di un file dal server tramite il comando `GET` ;
- L'upload di un file sul server tramite il comando `PUT` ;
- Il trasferimento di file in modo affidabile.

La comunicazione tra client e server deve avvenire tramite un opportuno protocollo. Il protocollo di comunicazione deve prevedere lo scambio di due tipi di messaggi:

- *messaggi di comando*: vengono inviati dal client al server per richiedere delle diverse operazioni;
- *messaggi di risposta*: vengono inviati dal server al client, in risposta ad un comando, con l'esito dell'operazione.

#### 1.1.1 Funzionalità del server

Il server, di tipo concorrente, deve fornire le seguenti funzionalità:

- L'invio del messaggio di risposta al comando `LIST` al client richiedente; il messaggio di risposta contiene la lista dei file, ovvero la lista dei nomi dei file disponibili per la condivisione;

- L'invio del messaggio di risposta al comando `GET` contenente il file richiesto, se presente, o un opportuno messaggio di errore;
- La ricezione di un messaggio `PUT` contenente il file da caricare sul server e l'invio di un messaggio di risposta con l'esito dell'operazione.

### 1.1.2 Funzionalità del client 👤

Il client, di tipo concorrente, deve fornire le seguenti funzionalità:

- L'invio del messaggio `LIST` per richiedere la lista dei nomi dei file disponibili;
- L'invio del messaggio `GET` per ottenere un file;
- La ricezione di un file, richiesta tramite il messaggio di `GET`, o la gestione dell'eventuale errore;
- L'invio del messaggio `PUT` per effettuare l'upload di un file sul server e la ricezione del messaggio di risposta con l'esito dell'operazione.

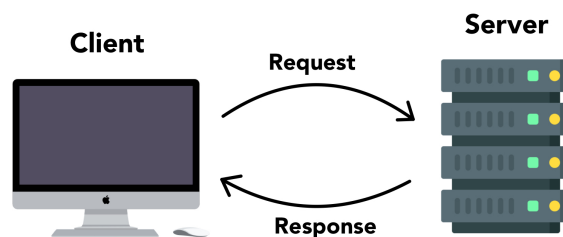


Figura 1.1: Schema client-server









### 1.1.3 Trasmissione affidabile 🤝

Lo scambio di messaggi avviene usando un servizio di comunicazione non affidabile. Al fine di garantire la corretta spedizione/ricezione dei messaggi e dei file sia i client che il server implementano a livello applicativo il protocollo *Selective Repeat* con finestra di spedizione `WINDOW_SIZE`. Per simulare la perdita dei messaggi in rete (evento alquanto improbabile in una rete locale per non parlare di quando client e server sono eseguiti sullo stesso host), si assume che ogni messaggio sia scartato dal mittente con probabilità `LOSS_PROBABILITY`. La dimensione della finestra di spedizione `WINDOW_SIZE`, la probabilità di perdita dei messaggi `LOSS_PROBABILITY`, e la durata del timeout `TIMEOUT`, sono tre costanti configurabili ed uguali per tutti i processi. Oltre all'uso di un timeout fisso (*static timeout*), deve essere possibile scegliere l'uso di un valore per il timeout adattativo (*adaptive timeout*) calcolato dinamicamente in base all'evoluzione dei ritardi di rete osservati. I client ed il server devono essere eseguiti nello spazio utente senza richiedere privilegi di root. Il server deve essere in ascolto su una porta di default (configurabile).








## 1.2 Architettura del sistema

### 1.2.1 Organizzazione del progetto

Il progetto S.P.Q.R. (*Selective Protocol for Quality and Reliability*) consta di diverse directory<sup>1</sup>, tra le più d'interesse troviamo:

-  `src/` : contiene i sorgenti del progetto, organizzato in file dedicati per lo sviluppo di specifiche funzionalità per il client e per il server;
-  `include/` : contiene gli header del progetto, organizzati per modularità e riutilizzo del codice;
-  `client-files/` e  `server-files/` : nelle quali troviamo i file che il client e il server si scambiano tra loro;
-  `tests/` : nella quale è presente lo script `integrity-consistency.py` che controlla l'integrità di ogni file inviato e/o ricevuto dal client e dal server;
-  `tests/network/` : nella quale è presente lo script `list.pcapng` che riporta il traffico di rete generato dal client e dal server durante l'esecuzione del comando `LIST` ;
-  `tests/performance/` : nella quale si trovano i grafici delle performance relativi al timeout statico, al timeout adattivo e alla cumulazione degli errori;
-  `docs/` : nella quale si trova il sito web del progetto.


Oltre alle directory citate, troviamo il  `Makefile` di particolare rilievo per automatizzare la compilazione del progetto. Entrando più nel dettaglio, nella directory `src/` troviamo i seguenti file:

-  `client.c` : rappresenta il punto d'ingresso del client ed in particolare si occupa di gestire l'argomento `IPv4` (cioè l'indirizzo IP del server a cui i client devono connettersi) tramite riga di comando, istanzia il gestore dei segnali e tenta l'avvio della connessione con il server;
-  `server.c` : rappresenta il punto d'ingresso del server, configura il socket principale, gestisce le connessioni in entrata e istanzia il gestore dei segnali;
-  `common.c` : contiene funzioni e strutture condivise tra client e server (es: gestione timeout, progress bar, ASCII art, gestione degli errori, simulazione perdita pacchetti, ...);
-  `protocol.c` : implementa il protocollo di trasferimento dati implementando il *Selective Repeat* ;
-  `spqr_client.c` : implementa la logica del client, l'invio/ricezione pacchetti, comandi utente `LIST` , `GET` , `PUT` e gestione del comando aggiuntivo `CLOSE` ;
-  `spqr_server.c` : implementa la logica del server, la gestione concorrente di client multipli e la bitmask per tracciare le connessioni attive.


---

<sup>1</sup>Si vuole specificare che la *working directory* `spqr/` è composta fondamentalmente dalle directory citate.

Mentre, per quanto riguarda la directory `include/`, troviamo i seguenti file:


- `stdc.h`: contiene tutte le librerie standard del linguaggio C e file header custom come `settings.h` e `common.h`;
- `settings.h`: contiene le configurazioni globali e i parametri del progetto (es: il numero di porta, il numero massimo di client supportati, il timeout statico, il timeout adattivo, la dimensione della finestra, la probabilità di perdita, ...);
- `common.h`: contiene i prototipi delle funzioni e dei messaggi custom condivisi tra client e server;
- `protocol.h`: definizione delle strutture utili per il protocollo di trasferimento dati in modo affidabile implementando tramite *Selective Repeat* ;
- `spqr_client.h`: prototipi delle funzioni specifiche del client (es: gestione delle connessioni (*Three-Way Handshake* con il server), gestione della terminazione con `ctrl+c`, ...);
- `spqr_server.h`: prototipi delle funzioni specifiche del server (es: supporto a client multipli via `fork()`, risposta ai comandi `LIST`, `GET`, `PUT` e gestione del comando aggiuntivo `CLOSE`, ...).

## 1.2.2 Protocollo Selective Repeat

L'architettura client-server progettata utilizza il protocollo di comunicazione non affidabile UDP per il trasferimento dei messaggi. Tuttavia, per garantire l'affidabilità nella trasmissione dei file tra client e server, è stato implementato il protocollo *Selective Repeat* . Questo protocollo ottimizza la gestione degli errori, riducendo al minimo le ritrasmissioni grazie alla selezione mirata dei pacchetti persi o corrotti, migliorando così l'efficienza complessiva del sistema.

### Funzionamento generale

Il protocollo *Selective Repeat* si basa sull'uso di una finestra scorrevole condivisa tra mittente e destinatario, consentendo l'invio e la ricezione di più pacchetti senza dover attendere una conferma immediata. La dimensione della finestra determina il numero massimo di pacchetti che possono essere “in volo” contemporaneamente. Ogni pacchetto trasmesso è identificato da un numero di sequenza univoco, che permette al destinatario di riordinare correttamente i dati ricevuti.

A differenza di altri protocolli di affidabilità, il *Selective Repeat*  utilizza ACK selettivi, ovvero conferme individuali per ogni pacchetto ricevuto, anche se fuori ordine. I pacchetti non ancora riordinabili vengono temporaneamente memorizzati in un buffer, in attesa di ricevere quelli mancanti.

### Comportamento del Mittente

Il mittente gestisce la trasmissione dei pacchetti rispettando la finestra di invio. Ogni pacchetto viene inviato con un timer individuale che controlla il tempo massimo di attesa per la conferma.

Quando il mittente riceve un ACK, il pacchetto corrispondente viene marcato come confermato e la finestra scorre, consentendo l'invio di nuovi dati. Se il timer di un pacchetto scade senza che sia stato ricevuto un ACK, il pacchetto viene ritrasmesso selettivamente, evitando di reinviare quelli già confermati.

## Comportamento del Destinatario

Il destinatario, una volta ricevuti i pacchetti, segue una logica basata su bufferizzazione e riordinamento:

- Accetta i pacchetti anche se ricevuti fuori ordine.
- I pacchetti non immediatamente utilizzabili vengono temporaneamente salvati in un buffer.
- Ogni pacchetto ricevuto genera un ACK, inviato al mittente indipendentemente dalla ricezione dei pacchetti precedenti.
- Se un pacchetto è già stato confermato in precedenza, eventuali ACK duplicati vengono ignorati dal mittente.
- Una volta ricevuti tutti i pacchetti fino a un certo numero di sequenza, i dati vengono consegnati all'applicazione in modo ordinato.

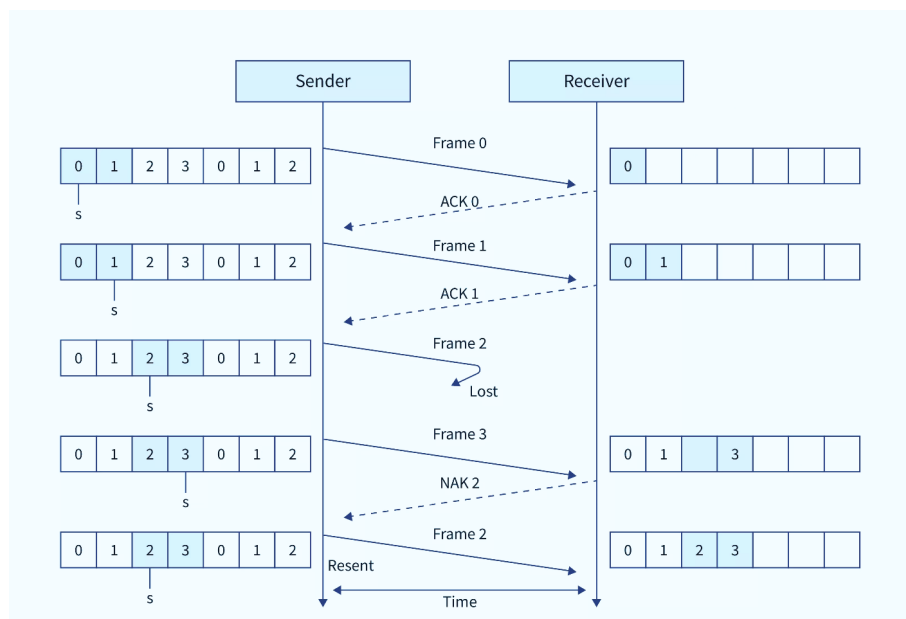


Figura 1.2: Funzionamento del protocollo *Selective Repeat* ✉.

## 1.3 Scelte progettuali

### 1.3.1 Gestione della concorrenza tramite processi 🔗

Il server è stato progettato come applicazione multi-processo in grado di gestire connessioni multiple. Ogni volta che un nuovo client si connette, il server crea un processo figlio dedicato esclusivamente a quella connessione. Questo approccio garantisce:



1. **Isolamento dalle sessioni:** un malfunzionamento di un client non influisce sugli altri. Questo rende il sistema più robusto e resiliente ai guasti;
2. **Semplicità gestionale:** evita complessità legate al multithreading, come race condition o deadlock.

### 1.3.2 Bitmask

Per tenere traccia degli slot disponibili, affinché un generico client possa connettersi al server, quest'ultimo fa uso di una bitmask. La bitmask è un array di bit, in cui ogni bit rappresenta lo stato di uno slot come mostrato in Fig. 1.3.

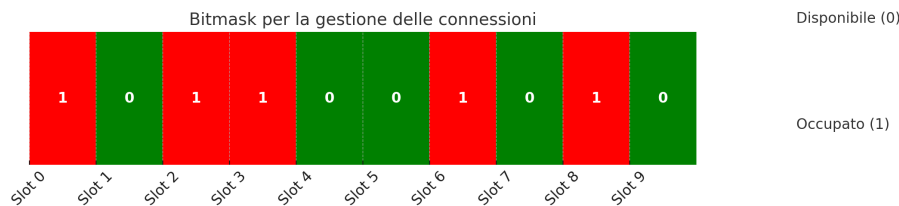


Figura 1.3: Rappresentazione grafica della bitmask.

dove ogni bit della bitmask rappresenta un generico client connesso o meno al server. Poiché si è scelto di poter accogliere al massimo 2727 client, la bitmask è stata dimensionata per contenere 2727 bit. Questo implica che la dimensione della memoria condivisa, creata con le API di System V IPC, è di 341 byte, calcolata come segue:

$$\frac{2727}{8} = 341 \text{ byte}$$

### 1.3.3 Casi particolari

Sia i client che il server sono dotati di handler per la gestione dei segnali, come `SIGINT` e `SIGQUIT`, garantendo una terminazione e una pulizia corretta delle risorse, sia nel caso in cui viene terminato prima il client, sia nel caso in cui viene terminato prima il server. Inoltre è stato gestito il caso in cui il server è down e un generico client tenta di connettersi. In questo caso il client termina l'esecuzione dopo tre tentativi di connessione falliti. Per ultimo è stato gestito il caso in cui il server è occupato, ovvero tutti gli slot sono occupati, in questo caso il server notifica il client e quest'ultimo termina l'esecuzione.

### 1.3.4 Timeout di inattività

L'applicazione prevede un meccanismo di timeout per la chiusura automatica delle connessioni inattive. Si è considerato un tempo di inattività di 3600 secondi (1 ora) come limite massimo di tempo per la connessione. Se un client non invia alcun messaggio al server per un periodo di tempo superiore a 1 ora, la connessione viene chiusa automaticamente dal server. Questo meccanismo previene la persistenza di connessioni “zombie” e garantisce un utilizzo efficiente delle risorse del server.

### 1.3.5 Gestione degli errori e pacchetti persi 🌟📦

L'applicazione interrompe il trasferimento se si verificano più di 25 errori consecutivi. Questo numero è stato scelto per evitare che il protocollo di trasferimento dati entri in uno stato di loop infinito, ad esempio a causa della perdita o corruzione dei pacchetti. Oltre questo limite, la quantità di dati corrotti renderebbe impossibile la ricostruzione fedele del file, compromettendone l'integrità e l'usabilità.

### 1.3.6 Three-Way Handshake 🤝

Per stabilire una connessione tra client e server, è stato implementato un *Three-Way Handshake* per garantire l'affidabilità e l'integrità della connessione. Il *Three-Way Handshake* è un protocollo di comunicazione a tre passaggi che consente a due host di stabilire una connessione TCP/IP. Il protocollo funziona come segue:

1. Il client invia un pacchetto di richiesta di connessione al server, contenente il flag `SYN` (synchronization request).
2. Il server risponde con un pacchetto di conferma, contenente il flag `SYN` e `ACK` (acknowledgment).
3. Il client invia un pacchetto di conferma al server, contenente il flag `ACK`.
4. La connessione è stabilita e i due host possono iniziare a scambiarsi dati.

#### Three-Way Handshake di apertura

Per la realizzazione del software S.P.Q.R. è stato implementato un *Three-Way Handshake* di apertura per stabilire la connessione tra client e server. Il client invia un pacchetto di richiesta di connessione al server, contenente il flag `SYN`. Il server risponde con un pacchetto di conferma, contenente il flag `SYN-ACK`. Infine, il client invia un pacchetto di conferma al server, contenente il flag `ACK`. La connessione è stabilita e i due host possono iniziare a scambiarsi dati.

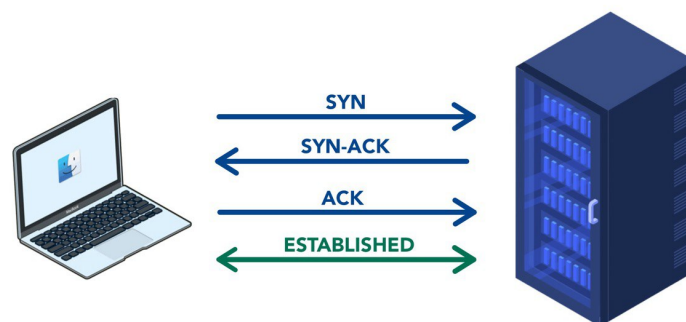


Figura 1.4: Three-Way Handshake d'apertura.

### Three-Way Handshake di chiusura

Lo stesso vale per il *Three-Way Handshake* di chiusura, che consente a client e server di terminare la connessione in modo sicuro. Il client invia un pacchetto di richiesta di chiusura al server, contenente il flag `FIN`. Il server risponde con un pacchetto di conferma, contenente il flag `FINACK` e invia un pacchetto di conferma al client, contenente il flag `FIN`. Il client risponde con un pacchetto di conferma al server, contenente il flag `ACK`. La connessione è chiusa e i due host possono liberare le risorse allocate.

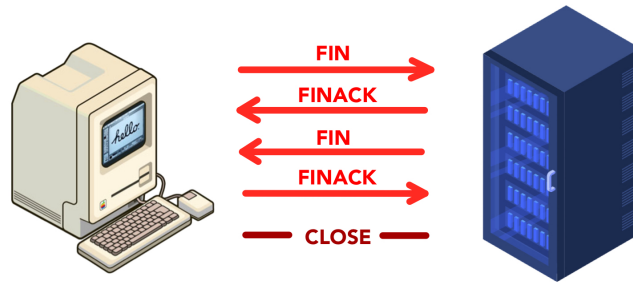


Figura 1.5: Three-Way Handshake di chiusura.

### 1.3.7 Progressive Bar 🚀

Per rendere il trasferimento dei file più interattivo e coinvolgente, è stata implementata una barra di avanzamento. La barra di avanzamento mostra la percentuale di completamento del trasferimento, aggiornata in tempo reale. Questo permette all'utente di monitorare il progresso del trasferimento e di avere un'idea chiara del tempo rimanente.

Figura 1.6: Animazione della progressive bar.

# Capitolo 2

## Descrizione dell'implementazione

In questo capitolo saranno illustrate nel dettaglio le scelte progettuali, con particolare enfasi sull'implementazione, e le soluzioni tecniche adottate nello sviluppo del software S.P.Q.R. (*Selective Protocol for Quality and Reliability*). Attraverso un'analisi strutturata, verranno presentati i componenti fondamentali del sistema, con particolare attenzione agli aspetti critici che ne garantiscono l'efficienza e l'affidabilità. Infine saranno analizzate le limitazioni riscontrate durante lo sviluppo.

### 2.1 Implementazione del server

Il server è il cuore del sistema, responsabile della gestione delle connessioni con i client, dell'elaborazione dei comandi e del trasferimento dei file. Di seguito viene riportato il codice sorgente del server attraverso il quale seguirà una spiegazione dettagliata delle funzioni principali.

```
1  #include "spqr_server.h"
2
3  int32_t main(void) {
4      setup_signal_handling(PARENT);
5
6      int32_t sockfd = server_create_socket(PORT - 1);
7
8      load_ascii_art();
9      puts("🌐 Server is listening for incoming connections.");
10
11     return server_manage_client_connections(sockfd);
12 }
```

Listing 2.1: `src/server.c`

Analizziamo la prima funzione, `setup_signal_handling()`, che si occupa di settare i gestori dei segnali per il processo padre del server. Si noti che quest'ultima può essere usata anche per il processo figlio, in base al parametro passato.

```
1  void setup_signal_handling(signal_handler_t type) {
2      struct sigaction sa;
3      sigset_t set;
4
5      memset(&sa, 0, sizeof(sa));
6      sa.sa_flags = 0;
7      spqr_assert(sigemptyset(&sa.sa_mask), "sigemptyset");
8
9      if (type == PARENT) {
10         sa.sa_handler = parent_server_signal_handler;
11         spqr_assert(sigaction(SIGINT, &sa, NULL), "sigaction");
12         spqr_assert(sigaction(SIGQUIT, &sa, NULL), "sigaction");
```

```

13
14     } else if (type == CHILD) {
15         sa.sa_handler = child_server_signal_handler;
16         spqr_assert(sigaction(SIGTERM, &sa, NULL), "sigaction");
17
18         // ? Block all signals except SIGTERM
19         spqr_assert(sigfillset(&set), "sigfillset");
20         spqr_assert(sigdelset(&set, SIGTERM), "sigdelset");
21     }
22
23     spqr_assert(sigprocmask(SIG_BLOCK, &set, NULL), "sigprocmask");
24 }

```

Listing 2.2: src/spqr\_server.c

All'interno della funzione `setup_signal_handling()` vengono inizializzate le strutture necessarie per la gestione dei segnali, in particolare vengono settati i gestori dei segnali per i segnali `SIGINT` e `SIGQUIT` nel caso del processo padre, mentre per il processo figlio viene settato il gestore per il segnale `SIGTERM`. Di seguito sono riportate le implementazioni delle funzioni `parent_server_signal_handler()` e `child_server_signal_handler()`.

```

1 void parent_server_signal_handler(const int32_t signo) {
2     if (signo == SIGINT || signo == SIGQUIT) {
3
4         #ifdef DEBUG
5             printf("\n[DEBUG 🐛] Server terminated due to %s.\n", signo == SIGINT ? "SIGINT" :
6                 "SIGQUIT");
7         #endif
8
9         for (uint32_t i = 0; i < MAX_CLIENTS; ++i) {
10             if (child_pids[i] != INVALID_PID) {
11                 kill(child_pids[i], SIGTERM);
12             }
13         }
14
15         for (uint32_t i = 0; i < MAX_CLIENTS; ++i) {
16             if (child_pids[i] != INVALID_PID) {
17                 waitpid(child_pids[i], NULL, 0);
18                 #ifdef DEBUG
19                     printf("[DEBUG 🐛] Child process #%d terminated.\n", i);
20                 #endif
21             }
22
23             if (i == MAX_CLIENTS - 1) {
24                 spqr_assert(shmdt(shm_bitmask), "shmdt");
25                 spqr_assert(shmctl(shm_id, IPC_RMID, NULL), "shmctl");
26             }
27
28             child_pids[i] = INVALID_PID;
29         }
30
31         puts(SPQR_CLOSE_CONNECTION);
32         exit(EXIT_SUCCESS);
33     }
34 }
35
36 void child_server_signal_handler(const int32_t signo) {
37     if (signo == SIGTERM) {
38         char dummy[MAX_READ_LINE];
39         server_receive_packet(dummy, child_sockfd, MAX_READ_LINE);
40         memset(dummy, 0, MAX_READ_LINE);
41         server_close_connection_with_client(child_sockfd);
42         spqr_assert(close(child_sockfd), "close");
43         exit(EXIT_SUCCESS);
44     }
45 }

```

Listing 2.3: src/spqr\_server.c

La seconda funzione di interesse è `server_manage_client_connections()`. I punti salienti riguardanti tale funzione sono la stabilizzazione della connessione e l'assegnazione di una porta dedicata per ogni client, la gestione dei processi figli, la pulizia dei processi zombie, l'utilizzo della funzione `select()` per gestire le connessioni in arrivo, la gestione dei segnali e la gestione dei comandi inviati dai client.

```

1  int8_t server_manage_client_connections(const int32_t sockfd) {
2      uint16_t port;
3      char buffer[MAX_READ_LINE];
4      memset(buffer, 0, sizeof(buffer));
5
6      // ? Initialize shared memory and bitmask as before
7      key_t key = ftok(KEY_PATH, 's');
8      spqr_assert(key, "ftok");
9
10     shm_id = shmget(key, SHM_SIZE, IPC_CREAT | PERMS);
11     spqr_assert(shm_id, "shmget");
12
13     shm_bitmask = (uint8_t *)shmat(shm_id, NULL, 0);
14     spqr_assert_shm(shm_bitmask, "shmat");
15     memset(shm_bitmask, FREE, SHM_SIZE);
16
17     memset(child_pids, INVALID_PID, sizeof(child_pids));
18
19     // ? Set up the file descriptor set.
20     fd_set read_fds, master_fds;
21     int32_t max_fd = sockfd;
22
23     FD_ZERO(&master_fds);
24     FD_SET(sockfd, &master_fds);
25
26     // ? Main server loop.
27     while (true) {
28         read_fds = master_fds;
29
30         // ? Set timeout for select
31         struct timeval timeout;
32         timeout.tv_sec = 1;
33         timeout.tv_usec = 0;
34
35         int32_t ready = select(max_fd + 1, &read_fds, NULL, NULL, &timeout);
36         if (ready < 0) {
37             if (errno == EINTR) continue; // ? Handle interruption by signal.
38             HANDLE_ERROR("select");
39             break;
40         }
41
42         // ? Check for new connections on the main socket.
43         if (FD_ISSET(sockfd, &read_fds)) {
44             if (server_open_connection_with_client(sockfd)) {
45                 uint32_t no_client = 0;
46                 for (; no_client < MAX_CLIENTS; ++no_client) {
47                     if (!get_bit(no_client)) {
48                         set_bit(no_client, BUSY);
49                         break;
50                     }
51                 }
52
53                 if (no_client == MAX_CLIENTS - 1) {
54                     server_send_packet(SPQR_SERVER_BUSY, sockfd);
55                     continue;
56                 }
57
58                 port = PORT + no_client;
59                 memset(buffer, 0, sizeof(buffer));
60                 spqr_assert(snprintf(buffer, sizeof(buffer), "%d", port), "snprintf");
61                 server_send_packet(buffer, sockfd);
62                 printf("\nThe client #%d is connected on port %d.\n", no_client, port);
63
64                 pid_t pid = fork();
65                 spqr_assert(pid, "fork");

```

```

66
67         if (pid == 0) { // ? Child process.
68             spqr_assert(close(sockfd), "close"); // ? Close the parent's socket
because it is not needed.
69             setup_signal_handling(CHILD);
70             init_packet_loss_simulator();
71
72             child_sockfd = server_create_socket(port);
73             int32_t status = server_manage_client_commands(child_sockfd);
74
75             set_bit(no_client, FREE);
76             printf("Closed connection for client #%d on port %d with exit code %d
.\n", no_client, port, status);
77
78             exit(status); // ? Close the child process.
79
80         } else { // ? Parent process.
81             child_pids[no_client] = pid;
82         }
83     }
84 }
85
86 // ? Check for zombie processes and clean them up.
87 int32_t status;
88 pid_t wpid;
89 while ((wpid = waitpid(INVALID_PID, &status, WNOHANG)) > 0) {
90     // ? Handle terminated child process.
91     for (uint32_t i = 0; i < MAX_CLIENTS; ++i) {
92         if (child_pids[i] == wpid) {
93             child_pids[i] = INVALID_PID;
94             set_bit(i, FREE);
95             break;
96         }
97     }
98 }
99 }
100
101 return EXIT_SUCCESS;
102 }

```

Listing 2.4: src/spqr\_server.c

Inoltre, come già accennato nel capitolo precedente, si è implementata una bitmask tramite memoria condivisa per tenere traccia delle connessioni attive. In particolare le funzioni `set_bit()` e `get_bit()` sono fondamentali per la gestione della bitmask.

```

1 void set_bit(const uint32_t client_id, const bitmask_t state) {
2     int32_t byte_index = client_id / CHAR_BIT;
3     int32_t bit_index = client_id % CHAR_BIT;
4     if (state == BUSY) {
5         shm_bitmask[byte_index] |= (1 << bit_index); // ? Set the bit to 1
6     } else {
7         shm_bitmask[byte_index] &= ~(1 << bit_index); // ? Set the bit to 0
8     }
9 }
10
11 int8_t get_bit(const uint32_t client_id) {
12     int32_t byte_index = client_id / CHAR_BIT;
13     int32_t bit_index = client_id % CHAR_BIT;
14     return (shm_bitmask[byte_index] >> bit_index) & 1;
15 }

```

Listing 2.5: src/spqr\_server.c

Infine, una volta che un client si connette al server, viene creato un processo figlio per gestire la connessione tramite la funzione `server_manage_client_commands()`. Quest'ultima ha il compito di gestire i comandi inviati dal client, in particolare i comandi `GET` e `PUT` per il trasferimento dei file e i comandi `LIST` e `CLOSE` per la visualizzazione dei file e la chiusura della connessione.

## 2.2 Implementazione del client

Il client è il componente che si occupa di stabilire la connessione con il server, inviare i comandi e ricevere/inviare i file richiesti. Di seguito viene riportato il codice sorgente del client attraverso il quale seguirà una spiegazione dettagliata delle funzioni principali.

```
1 #include "spqr_client.h"
2
3 int32_t main(int32_t argc, const char **argv) {
4     if (argc != 2) {
5         fprintf(stderr, "Usage: %s <IPv4>\n", *argv);
6         return EXIT_FAILURE;
7     }
8
9     setup_signal_handling();
10
11     init_packet_loss_simulator();
12
13     server_response = malloc(MAX_READ_LINE);
14     spqr_assert_ptr(server_response, "malloc");
15     memset(server_response, 0, MAX_READ_LINE);
16
17     if (!client_establish_connection_with_server(argv[1])) {
18         spqr_free(&server_response);
19         spqr_assert(close(sockfd), "close");
20         return EXIT_FAILURE;
21     }
22
23     load_ascii_art();
24     printf("🌐 Connection established with the server.\n");
25     printf(INFO, WINDOW_SIZE, LOSS_PROBABILITY, TIMEOUT, ADAPTIVE ? "true" : "false");
26
27     return client_manage_server_commands();
28 }
```

Listing 2.6: src/client.c

Di particolare interesse è la funzione `setup_signal_handling()` che si occupa di settare i gestori dei segnali per il client.

```
1 void setup_signal_handling(void) {
2     struct sigaction sa;
3     sigset_t set;
4
5     memset(&sa, 0, sizeof(sa));
6     sa.sa_handler = client_signal_handler;
7     sa.sa_flags = 0;
8     spqr_assert(sigemptyset(&sa.sa_mask), "sigemptyset");
9     spqr_assert(sigaction(SIGINT, &sa, NULL), "sigaction");
10    spqr_assert(sigaction(SIGQUIT, &sa, NULL), "sigaction");
11    spqr_assert(sigaction(SIGALRM, &sa, NULL), "sigaction");
12
13    // ? Block all signals except SIGINT, SIGQUIT, SIGALRM.
14    spqr_assert(sigfillset(&set), "sigfillset");
15    spqr_assert(sigdelset(&set, SIGINT), "sigdelset");
16    spqr_assert(sigdelset(&set, SIGQUIT), "sigdelset");
17    spqr_assert(sigdelset(&set, SIGALRM), "sigdelset");
18    spqr_assert(sigprocmask(SIG_BLOCK, &set, NULL), "sigprocmask");
19 }
```

Listing 2.7: src/spqr\_client.c

dove `client_signal_handler()` è la funzione che si occupa di gestire i segnali inviati al client.

```
1 void client_signal_handler(const int32_t sig) {
2     if (sig == SIGINT) {
3         #ifdef DEBUG
4             puts("\n SPQR_SESSION_STOPPED);
5         #endif
6         client_send_packet(feedback.EXIT, sockfd);
7     }
8 }
```



```

7     client_close_connection_with_server(sockfd);
8
9     } else if (sig == SIGQUIT) {
10    #ifdef DEBUG
11        puts("\n" SPQR_SESSION_STOPPED);
12    #endif
13        client_send_packet(feedback.EXIT, sockfd);
14
15    } else if (sig == SIGALRM) {
16        puts("\n" SPQR_TIMEOUT_SESSION);
17
18        client_send_packet(feedback.EXIT, sockfd);
19        client_close_connection_with_server(sockfd);
20    }
21
22    spqr_free(&server_response);
23    spqr_free(&client_pathname);
24    spqr_assert(close(sockfd), "close");
25
26    exit(EXIT_SUCCESS);
27 }

```

Listing 2.8: src/spqr\_client.c

Un'altra funzione di interesse è `client_establish_connection_with_server()`, la quale si occupa di stabilire la connessione con il server.

```

1  bool client_establish_connection_with_server(const char *ip) {
2      int8_t connection_attempts = CONNECTION_ATTEMPTS;
3
4      sockfd = client_create_socket(ip, PORT - 1);
5      spqr_assert(sockfd, "client_create_socket");
6
7      set_seconds_timeout(sockfd, CONNECTION_ATTEMPTS);
8      while (connection_attempts > 0) {
9          if (!client_open_connection_with_server(sockfd)) {
10             printf(COLOR_RED "⚠ Connection refused (remaining attempts: %d)\n", --
connection_attempts);
11             continue;
12         }
13         set_seconds_timeout(sockfd, 0);
14
15         client_receive_packet(server_response, sockfd, MAX_READ_LINE);
16         if (strcmp(server_response, SPQR_SERVER_BUSY) == 0) {
17             puts(SPQR_SERVER_BUSY);
18             return false;
19         }
20
21         uint16_t port;
22         if (!to_uint16(server_response, &port)) {
23             #ifdef DEBUG
24                 HANDLE_ERROR("to_uint16");
25             #endif
26             return false;
27         }
28
29         spqr_assert(close(sockfd), "close");
30         sockfd = client_create_socket(ip, port);
31         break;
32     }
33     set_seconds_timeout(sockfd, 0);
34
35     if (connection_attempts == 0) {
36         puts(COLOR_RESET SPQR_CONNECTION_REFUSED);
37         return false;
38     }
39
40     puts(COLOR_RESET);
41     return true;
42 }

```

Listing 2.9: src/spqr\_client.c

In particolare essa gestisce i tentativi di connessione e il caso in cui il server è down. Infatti, in quest'ultimo caso il client tenterà di connettersi nuovamente per un numero di volte definito dalla costante `CONNECTION_ATTEMPTS` che per default è settata a 3. Per fare ciò è usato un timeout, tramite la funzione `set_seconds_timeout()`, per evitare che il client rimanga bloccato in attesa di una risposta dal server. Quindi, se il server è down, il client stamperà un messaggio di errore e terminerà la connessione. Infine, la funzione `client_manage_server_commands()` è responsabile della gestione dei comandi (`GET`, `PUT`, `LIST` e `CLOSE`) inviati dal server al client.

## 2.3 Implementazione del Three-Way Handshake

Il *Three-Way Handshake* di apertura, il cui funzionamento è stato descritto nel capitolo precedente, è stato implementato tramite le funzioni `server_open_connection_with_client()` e `client_open_connection_with_server()` per il server e il client rispettivamente.

```
1 bool server_open_connection_with_client(const int32_t sockfd) {
2     char buffer[MAX_READ_LINE];
3     memset(buffer, 0, MAX_READ_LINE);
4
5     server_receive_packet(buffer, sockfd, MAX_READ_LINE);
6     if (strcmp(buffer, udp.SYN)) {
7         #ifdef DEBUG
8             HANDLE_ERROR(udp.SYN);
9         #endif
10        return false;
11    }
12
13    server_send_packet(udp.SYNACK, sockfd);
14
15    server_receive_packet(buffer, sockfd, MAX_READ_LINE);
16    if (strcmp(buffer, udp.ACK)) {
17        #ifdef DEBUG
18            HANDLE_ERROR(udp.ACK);
19        #endif
20        return false;
21    }
22
23    return true;
24 }
```

Listing 2.10: `src/spqr_server.c`

```
1 bool client_open_connection_with_server(const int32_t sockfd) {
2     char buffer[MAX_READ_LINE];
3     memset(buffer, 0, MAX_READ_LINE);
4
5     client_send_packet(udp.SYN, sockfd);
6
7     client_receive_packet(buffer, sockfd, MAX_READ_LINE);
8     if (strcmp(buffer, udp.SYNACK)) {
9         #ifdef DEBUG
10            HANDLE_ERROR(udp.SYNACK);
11        #endif
12        return false;
13    }
14
15    client_send_packet(udp.ACK, sockfd);
16
17    return true;
18 }
```

Listing 2.11: `src/spqr_client.c`

Mentre il *Three-Way Handshake* di chiusura è stato implementato tramite le funzioni `server_close_connection_with_client()` e `client_close_connection_with_server()` per il server e il client rispettivamente.

```

1  int8_t server_close_connection_with_client(const int32_t sockfd) {
2      char buffer[MAX_READ_LINE];
3      memset(buffer, 0, MAX_READ_LINE);
4
5      server_receive_packet(buffer, sockfd, MAX_READ_LINE);
6      if (strcmp(buffer, udp.FIN)) {
7          #ifdef DEBUG
8              HANDLE_ERROR(udp.FIN);
9          #endif
10         return TRANSFER_ERROR;
11     }
12
13     server_send_packet(udp.FINACK, sockfd);
14     server_send_packet(udp.FIN, sockfd);
15
16     server_receive_packet(buffer, sockfd, MAX_READ_LINE);
17     if (strcmp(buffer, udp.FINACK)) {
18         #ifdef DEBUG
19             HANDLE_ERROR("FINACK");
20         #endif
21         return TRANSFER_ERROR;
22     }
23
24     return EXIT_SUCCESS;
25 }

```

Listing 2.12: `src/spqr_server.c`

```

1  int8_t client_close_connection_with_server(const int32_t sockfd) {
2      char buffer[MAX_READ_LINE];
3      memset(buffer, 0, MAX_READ_LINE);
4
5      client_send_packet(udp.FIN, sockfd);
6
7      client_receive_packet(buffer, sockfd, MAX_READ_LINE);
8      if (strcmp(buffer, udp.FINACK)) {
9          #ifdef DEBUG
10             HANDLE_ERROR(udp.FINACK);
11         #endif
12         return TRANSFER_ERROR;
13     }
14
15     client_receive_packet(buffer, sockfd, MAX_READ_LINE);
16     if (strcmp(buffer, udp.FIN)) {
17         #ifdef DEBUG
18             HANDLE_ERROR(udp.FIN);
19         #endif
20         return TRANSFER_ERROR;
21     }
22
23     client_send_packet(udp.FINACK, sockfd);
24     puts(SPQR_CLOSE_CONNECTION);
25     return EXIT_SUCCESS;
26 }

```

Listing 2.13: `src/spqr_client.c`

Infine si riportano le implementazioni dettagliate delle funzioni `server_receive_packet()`, `server_send_packet()`, `client_send_packet()` e `client_receive_packet()`.

```

1  char *server_receive_packet(char *msg, const int32_t sockfd, const uint64_t size) {
2      memset(msg, 0, size);
3      spqr_assert(recvfrom(sockfd, msg, size, MSG_WAITALL, (struct sockaddr *) &client_addr,
4                          &client_len), "recvfrom");
5      return msg;
6  }

```

```

7 void server_send_packet(const char *msg, const int32_t sockfd) {
8     spqr_assert(sendto(sockfd, msg, strlen(msg), 0, (struct sockaddr *) &client_addr,
9         client_len), "sendto");

```

Listing 2.14: src/spqr\_server.c

```

1 void client_send_packet(const char *msg, const int32_t sockfd) {
2     spqr_assert(sendto(sockfd, msg, strlen(msg), 0, (struct sockaddr *) &server_addr,
3         server_len), "sendto");
4 }
5 char *client_receive_packet(char *msg, const int32_t sockfd, const uint64_t size) {
6     memset(msg, 0, size);
7     int64_t n = recvfrom(sockfd, msg, size, MSG_WAITALL, (struct sockaddr *) &server_addr,
8         &server_len);
9     if (n < 0) {
10         if (errno == EWOULDBLOCK || errno == EAGAIN) {
11             return NULL;
12         } else {
13             #ifdef DEBUG
14                 HANDLE_ERROR("recvfrom");
15             #endif
16         }
17     }
18     return msg;
19 }

```

Listing 2.15: src/spqr\_client.c

## 2.4 Implementazione del protocollo Selective Repeat

Per l'implementazione del protocollo *Selective Repeat* sono state implementate due strutture dati, `packet_t` e `ack_packet_t`, per rappresentare i pacchetti e i pacchetti di ack rispettivamente.

```

1 #define PACKET_SIZE 1500 // ? bytes
2
3 typedef struct {
4     int8_t payload[PACKET_SIZE]; // ? Data payload
5     int32_t seq_num; // ? Sequence number
6     int32_t no_packets_to_send; // ? Total number of packets to send
7     int64_t no_bytes_to_send; // ? Total number of bytes to send
8     int64_t size; // ? Size of the packet
9     bool sent; // ? Sent flag
10    bool received; // ? Received flag
11    bool ack; // ? Acknowledged flag
12 } packet_t;
13
14 typedef struct {
15     int32_t seq_num; // ? Sequence number
16     int64_t size; // ? Size of the packet
17     int64_t write_byte; // ? Number of bytes written
18 } ack_packet_t;

```

Listing 2.16: src/protocol.h

### 2.4.1 Implementazione del sender

Il sender è responsabile dell'invio dei pacchetti e della gestione degli ack ricevuti. In questo caso particolare sono state implementate tre funzioni principali: `main_sender()`, `send_data_to_receiver()` e `wait_ack()`. La funzione `main_sender()` si occupa di configurare

il timeout del socket, di aprire e leggere il file da inviare, di calcolare la dimensione del file e il numero di pacchetti da inviare, di inizializzare la finestra di invio, di coordinare l'invio dei pacchetti e la gestione degli ACK. Inoltre si occupa di calcolare le statistiche di throughput e di stamparle a fine trasmissione. Infine gestisce gli errori e il fallimento del trasferimento.

```

1  int8_t main_sender(int32_t sockfd, struct sockaddr_in *sender_addr, char *pathname) {
2      socklen_t len = sizeof(*sender_addr);
3      int32_t seq_num = 0;
4      int64_t n;
5      uint16_t max_errors = 0;
6      uint64_t counter = 0;
7      int32_t no_packets_to_send = 0;
8
9      // ? Set the timeout of the socket.
10     set_timeout(sockfd, TIMEOUT);
11
12     // ? Open the file in read-binary mode.
13     FILE *file = fopen(pathname, "rb");
14     if (file == NULL) {
15         HANDLE_ERROR("fopen");
16         return TRANSFER_ERROR;
17     }
18
19     // ? Calculate the size of the file.
20     struct stat st;
21     if (fstat(fileno(file), &st) != 0) {
22         HANDLE_ERROR("fstat");
23         fclose(file);
24         return TRANSFER_ERROR;
25     }
26     int64_t size_file = st.st_size;
27
28     // ? Calculate the number of packets to send.
29     no_packets_to_send = (size_file % PACKET_SIZE) ? (size_file / PACKET_SIZE) + 1 :
        size_file / PACKET_SIZE;
30
31     // ? Create the initial window.
32     uint32_t window_size = (no_packets_to_send < WINDOW_SIZE) ? no_packets_to_send :
        WINDOW_SIZE;
33     packet_t window[WINDOW_SIZE] = { 0 };
34
35     // ? Fill the window with the packets.
36     for (uint32_t i = 0; i < window_size; ++i) {
37         packet_t packet = { 0 };
38
39         uint64_t num_read = fread(packet.payload, 1, PACKET_SIZE, file);
40         if (num_read == 0) { break; }
41
42         counter += num_read;
43         packet.seq_num = seq_num;
44         packet.no_packets_to_send = no_packets_to_send;
45         packet.no_bytes_to_send = counter;
46         packet.size = num_read;
47         packet.sent = false;
48         packet.ack = false;
49         packet.received = false;
50         window[i] = packet;
51
52         seq_num = (seq_num + 1) % window_size;
53     }
54
55     #ifdef IS_CLIENT
56         struct timeval end, start;
57         spqr_assert(gettimeofday(&start, NULL), "gettimeofday");
58     #endif
59
60     // ? Start sending the packets.
61     send_data_to_receiver(sockfd, sender_addr, window, window_size);
62     wait_ack(sockfd, sender_addr, len, window, window_size, file, size_file, &max_errors,
        &counter, &no_packets_to_send);

```

```

63
64 #ifdef IS_CLIENT
65     spqr_assert(gettimeofday(&end, NULL), "gettimeofday");
66 #endif
67
68     int64_t ret = 0;
69     while (true) {
70         ack_packet_t new_ack = { 0 };
71         n = recvfrom(sockfd, &new_ack, sizeof(new_ack), 0, (struct sockaddr *) sender_addr
72         , &len);
73         if (n < 0) { break; }
74         ret = new_ack.write_byte;
75     }
76
77     spqr_assert(fclose(file), "fclose");
78     set_timeout(sockfd, 0);
79
80     if (ret == WRITE_BYTE_ERROR) { return TRANSFER_ERROR; }
81
82 #ifdef IS_CLIENT
83     double time = end.tv_sec - start.tv_sec + (double)(end.tv_usec - start.tv_usec) / 1e6;
84     double throughput = (time != 0.0) ? (size_file / time) / SPQR_KB : 0.0; // [kB/s]
85 #endif
86
87     if (max_errors >= MAX_ERRORS) {
88         #ifdef IS_CLIENT
89             printf("\n\n" FILE_TRANSFER_FAILED, time, throughput);
90         #endif
91
92         packet_t packet = { 0 };
93         packet.seq_num = SEQ_NUM_ERROR;
94
95         n = sendto(sockfd, &packet, sizeof(packet), 0, (struct sockaddr *) sender_addr,
96         len);
97         if (n < 0) { HANDLE_ERROR("sendto"); }
98
99         return TRANSFER_ERROR;
100     }
101
102 #ifdef IS_CLIENT
103     printf("\n\n" FILE_TRANSFER_COMPLETED, time, throughput);
104 #endif
105
106     return EXIT_SUCCESS;
107 }

```

Listing 2.17: src/protocol.c

La funzione `send_data_to_receiver()` si occupa dell'invio effettivo dei pacchetti al receiver, verificando quali pacchetti nella finestra devono essere inviati. Inoltre simula l'eventuale perdita di pacchetti tramite l'ausilio della funzione `can_send_packet()`.

```

1 // ? This function is used to simulate the loss of a packet.
2 // ? The function returns true if the packet can be sent, false otherwise.
3 bool can_send_packet(void) {
4     return (rand() % 100 + 1) > LOSS_PROBABILITY;
5 }

```

Listing 2.18: src/common.c

Infine marca i pacchetti come inviati e verifica il completamento del trasferimento.

```

1 void send_data_to_receiver(int32_t sockfd, struct sockaddr_in *sender_addr, packet_t *
2     window, uint32_t window_size) {
3     int64_t n;
4     for (uint32_t i = 0; i < window_size; ++i) {
5         if (window[i].ack == false) { // ? Check if the packet has been acknowledged.
6             if (can_send_packet()) { // ? Check if the packet is lost.
7                 n = sendto(sockfd, &window[i], sizeof(window[i]), 0, (struct sockaddr *)
8                 sender_addr, sizeof(*sender_addr));
9                 if (n < 0) {

```

```

8             HANDLE_ERROR("sendto");
9             continue;
10        }
11        window[i].sent = true;
12        if (window[i].size == 0 || n == 0) { break; } // ? Check if the file has
        been completely sent.
13    }
14    }
15 }
16 }

```

Listing 2.19: src/protocol.c

Infine la funzione `wait_ack()` attende e gestisce gli ACK dal receiver, implementa il meccanismo di timeout e ritrasmissione, gestisce il conteggio degli errori, aggiorna la finestra scorrevole, carica nuovi pacchetti quando necessario, implementa il controllo di flusso e gestisce il timeout adattivo se abilitato.

```

1 void wait_ack(int32_t sockfd, struct sockaddr_in *sender_addr, socklen_t len, packet_t *
    window, uint32_t window_size, FILE *file, int64_t size_file, uint16_t *max_errors,
    uint64_t *counter, int32_t *no_packets_to_send) {
2     int64_t n;
3     int32_t ack_num;
4     int64_t last_ack_confirmed_bytes = 0;
5     uint64_t num_read;
6     uint32_t i = 0;
7     ack_packet_t new_ack = { 0 };
8
9     #ifdef IS_CLIENT
10        printf("\n");
11    #endif
12
13    while (true) {
14        wait_for_ack:
15        n = recvfrom(sockfd, &new_ack, sizeof(new_ack), 0, (struct sockaddr *) sender_addr
        , &len); // ? Wait for the acknowledgment from the receiver.
16        if (n < 0) { // ? If the acknowledgment is not received.
17            (*max_errors)++;
18            if (*max_errors >= MAX_ERRORS) { return; }
19            if (ADAPTIVE) { increase_timeout(sockfd); }
20
21            // ? Resend the packets that have not been acknowledged.
22            for (uint32_t j = 0; j < window_size; ++j) {
23                if (window[j].ack == false || window[j].no_bytes_to_send >
        last_ack_confirmed_bytes) {
24                    if (can_send_packet()) { // ? Check if the packet is lost.
25                        n = sendto(sockfd, &window[j], sizeof(window[j]), MSG_CONFIRM, (
        struct sockaddr *) sender_addr, sizeof(*sender_addr));
26                        if (n < 0) {
27                            HANDLE_ERROR("sendto");
28                        } else {
29                            window[j].sent = true;
30                        }
31                    }
32                }
33            }
34            goto wait_for_ack;
35        }
36
37        *max_errors = 0;
38
39        // ? Check if the acknowledgment is received.
40        if (new_ack.seq_num == SEQ_NUM_ERROR || new_ack.write_byte > size_file || new_ack.
        write_byte == WRITE_BYTE_ERROR) { return; }
41        if (ADAPTIVE) { decrease_timeout(sockfd); }
42
43        ack_num = new_ack.seq_num;
44        if (new_ack.size > last_ack_confirmed_bytes) { last_ack_confirmed_bytes = new_ack.
        size; }
45
46        // ? If the acknowledgment is received, I skip to the next packet.
47        if (window[ack_num].ack == true) {

```

```

48         continue;
49     } else {
50         window[ack_num].ack = true;
51     }
52     #ifdef IS_CLIENT
53         print_progress(size_file, new_ack.write_byte);
54     #endif
55     while (window[i].ack && *no_packets_to_send > 0) {
56         packet_t packet = { 0 };
57         --(*no_packets_to_send);
58         num_read = fread(packet.payload, 1, PACKET_SIZE, file);
59         if (num_read == 0) { break; }
60         *counter += num_read;
61         packet.seq_num = i;
62         packet.no_packets_to_send = *no_packets_to_send;
63         packet.no_bytes_to_send = *counter;
64         packet.size = num_read;
65         packet.sent = false;
66         packet.ack = false;
67         packet.received = false;
68         window[i] = packet;
69         if (can_send_packet()) { // ? Check if the packet is lost.
70             n = sendto(sockfd, &window[i], sizeof(window[i]), 0, (struct sockaddr
*) sender_addr, sizeof(*sender_addr));
71             if (n < 0) {
72                 HANDLE_ERROR("sendto");
73             } else {
74                 window[i].sent = true;
75             }
76             if (window[i].size == 0 || n == 0) { break; } // ? Check if the file has
been completely sent.
77             i = (i + 1) % window_size; // ? Move to the next packet.
78         }
79     }
80 }
81 }
82 }
83 }
84 }

```

Listing 2.20: src/protocol.c

## 2.4.2 Implementazione del receiver

Il receiver è responsabile della ricezione dei pacchetti e dell'invio degli ACK al sender. In questo caso particolare sono state implementate due funzioni principali: `main_receiver()` e `receive_data_from_sender()`. La funzione `main_receiver()` si occupa di gestire la ricezione del file, di inizializzare la finestra di ricezione e di aprire il file in modalità di scrittura binaria. Successivamente si occupa di inviare il primo ACK al sender per iniziare il trasferimento e di gestire il calcolo del throughput (solo in modalità client). In caso di errori, elimina il file incompleto e invia l'ACK finale al termine della ricezione.

```

1  int8_t main_receiver(const int32_t sockfd, struct sockaddr_in *receiver_addr, const char *
    pathname) {
2      socklen_t receiver_len = sizeof(*receiver_addr);
3      int64_t n;
4      int64_t size_received = 0;
5      packet_t window[WINDOW_SIZE] = { 0 };
6
7      // ? Open the file in write-binary mode.
8      FILE *file = fopen(pathname, "wb");
9      if (file == NULL) {

```



```

10     HANDLE_ERROR("fopen");
11     return TRANSFER_ERROR;
12 }
13
14 // ? Send the first acknowledgment to the sender.
15 n = sendto(sockfd, feedback.START_TRANSFER, strlen(feedback.START_TRANSFER), 0, (
    struct sockaddr *) receiver_addr, receiver_len);
16 if (n < 0) {
17     HANDLE_ERROR("sendto");
18     fclose(file);
19     return TRANSFER_ERROR;
20 }
21
22 // ? Initialize the window.
23 for (int32_t i = 0, seq_num = 0; i < WINDOW_SIZE; ++i) {
24     window[i].seq_num = seq_num++;
25     window[i].received = false;
26     window[i].sent = false;
27     window[i].ack = false;
28     window[i].no_bytes_to_send = 0;
29     window[i].size = 0;
30 }
31
32 #ifdef IS_CLIENT
33     struct timeval end, start;
34     spqr_assert(gettimeofday(&start, NULL), "gettimeofday");
35 #endif
36
37 // ? Receive the data from the sender.
38 int8_t ret = receive_data_from_sender(sockfd, receiver_addr, receiver_len, window, &
    size_received, file);
39 spqr_assert(fclose(file), "fclose");
40
41 #ifdef IS_CLIENT
42     spqr_assert(gettimeofday(&end, NULL), "gettimeofday");
43     double time = end.tv_sec - start.tv_sec + (double)(end.tv_usec - start.tv_usec) / 1e6;
44     double throughput = (time != 0.0) ? (size_received / time) / SPQR_KB : 0.0; // [kB/s]
45 #endif
46
47 if (n < 0 || ret == TRANSFER_ERROR) {
48     #ifdef IS_CLIENT
49         printf("\n\n" FILE_TRANSFER_FAILED, time, throughput);
50     #endif
51
52     // ? Delete the file.
53     spqr_assert(remove(pathname), "remove");
54
55     #ifdef DEBUG
56         puts(SPQR_FILENAME_SUCCESSFULLY_DELETED);
57     #endif
58     return ret;
59 }
60
61 // ? Send the last acknowledgment to the sender.
62 for (int32_t i = 0; i < MAX_RETRIES; ++i) {
63     ack_packet_t ack = { 0 };
64     ack.seq_num = SEQ_NUM_ERROR;
65     ack.write_byte = size_received;
66     n = sendto(sockfd, &ack, sizeof(ack), 0, (struct sockaddr *) receiver_addr,
    receiver_len);
67     if (n < 0) { HANDLE_ERROR("sendto"); }
68 }
69
70 #ifdef IS_CLIENT
71     printf("\n\n" FILE_TRANSFER_COMPLETED, time, throughput);
72 #endif
73
74     return EXIT_SUCCESS;
75 }

```

Listing 2.21: src/protocol.c

La funzione `receive_data_from_sender()` si occupa della ricezione effettiva dei pacchetti dal sender, della verifica della sequenza dei pacchetti ricevuti e del controllo di eventuali pacchetti duplicati. Inoltre scrive i dati sul file in modo ordinato, invia gli ACK per i pacchetti ricevuti correttamente, gestisce la finestra scorrevole (lato receiver) e mostra la progress bar (solo in modalità client).

```

1  int8_t receive_data_from_sender(const int32_t sockfd, struct sockaddr_in *receiver_addr,
2      socklen_t receiver_len, packet_t *window, int64_t *size_received, FILE *file) {
3      int64_t n;
4      bool is_first = true;
5      int32_t seq_num;
6      int64_t no_bytes_to_send;
7      int32_t no_packets_to_receive = 0;
8      #ifdef IS_CLIENT
9          printf("\n");
10     #endif
11
12     uint32_t i = 0;
13
14     do {
15         packet_t new_packet = { 0 };
16
17     retry:
18         n = recvfrom(sockfd, &new_packet, sizeof(new_packet), 0, (struct sockaddr *)
19             receiver_addr, &receiver_len);
20         if (n < 0) { HANDLE_ERROR("recvfrom"); }
21
22         if (new_packet.seq_num == SEQ_NUM_ERROR) { return TRANSFER_ERROR; }
23
24         seq_num = new_packet.seq_num;
25         if (seq_num >= WINDOW_SIZE) {
26             ack_packet_t error_ack = { 0 };
27             error_ack.seq_num = SEQ_NUM_ERROR;
28             n = sendto(sockfd, &error_ack, sizeof(error_ack), 0, (struct sockaddr *)
29                 receiver_addr, receiver_len);
30             continue;
31         }
32
33         no_bytes_to_send = new_packet.no_bytes_to_send;
34
35         // ? If it is the first packet, then set the number of packets to receive.
36         if (is_first) {
37             no_packets_to_receive = new_packet.no_packets_to_send;
38             is_first = !is_first;
39         }
40
41         // ? If the packet has been received then send an acknowledgment.
42         // ? In particular, if the packet has already been received,
43         // ? the size of the packet received is less than the size of
44         // ? the packet to receive, or the number of bytes to send is
45         // ? less than or equal to the size received, then send an
46         // ? acknowledgment.
47         if (window[seq_num].received || (*size_received < window[seq_num].no_bytes_to_send
48             ) || (no_bytes_to_send <= *size_received)) {
49             ack_packet_t ack = { 0 };
50             ack.seq_num = seq_num;
51             ack.size = no_bytes_to_send;
52             ack.write_byte = *size_received;
53
54             // ? Simulate the packet loss.
55             if (can_send_packet()) {
56                 n = sendto(sockfd, &ack, sizeof(ack), 0, (struct sockaddr *) receiver_addr
57                     , receiver_len);
58                 if (n < 0) { HANDLE_ERROR("sendto"); }
59             }
60             goto retry;
61         }
62
63         // ? If the packet has not been received, then store it in the window.

```

```

61     window[seq_num] = new_packet;
62     window[seq_num].received = true;
63
64     // ? Process all available contiguous packets.
65     int32_t process_count = 0;
66     int32_t max_iterations = WINDOW_SIZE;
67
68     while (window[i].received && process_count++ < max_iterations) {
69         packet_t packet = window[i];
70
71         *size_received = packet.no_bytes_to_send;
72         int64_t num_write = fwrite(&packet.payload, 1, packet.size, file);
73         if (num_write != packet.size) {
74             HANDLE_ERROR("fwrite");
75             return WRITE_BYTE_ERROR;
76         }
77
78         #ifdef IS_CLIENT
79             static uint32_t packets_processed = 0;
80             packets_processed++;
81             print_progress(no_packets_to_receive + packets_processed, packets_processed);
82         #endif
83
84         no_packets_to_receive--;
85         window[i].received = false;
86         window[i].ack = false;
87         i = (i + 1) % WINDOW_SIZE;
88     }
89
90     ack_packet_t ack = { 0 };
91     ack.seq_num = seq_num;
92     ack.size = no_bytes_to_send;
93     ack.write_byte = *size_received;
94
95     if (can_send_packet()) { // ? Check if the packet is lost.
96         n = sendto(sockfd, &ack, sizeof(ack), 0, (struct sockaddr *) receiver_addr,
97 receiver_len);
98         if (n < 0) { HANDLE_ERROR("sendto"); }
99         window[seq_num].ack = true;
100     }
101 } while (no_packets_to_receive > 0);
102
103 return EXIT_SUCCESS;
104 }

```

Listing 2.22: src/protocol.c

## 2.5 Limitazioni riscontrate

Durante lo sviluppo del progetto sono state riscontrate due principali limitazioni legate al calcolo del tempo di trasferimento e alla gestione dei timeout.

### 2.5.1 Calcolo del tempo di trasferimento

Per il calcolo del tempo di trasferimento è stata utilizzata la funzione `gettimeofday()`. La scelta di utilizzare tale funzione per misurare il tempo totale di trasferimento anziché la nota formula:

$$\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT}$$

è dovuta alla semplicità e immediatezza, a discapito della precisione. Questo approccio calcola il tempo effettivo trascorso tra l'inizio e la fine della trasmissione, senza adattarsi alle fluttuazioni di rete.

### 2.5.2 Timeout sulla socket anziché sui singoli pacchetti

Nell'implementazione viene impostato un timeout a livello di socket e non sui singoli pacchetti, come il protocollo *Selective Repeat* prevede. Questo significa che l'intera socket “scade” se non riceve dati entro un certo periodo, anziché gestire timer individuali per ogni pacchetto. Questo approccio è stato scelto per la sua semplicità e per evitare la complessità aggiuntiva di gestire timer per ogni pacchetto.

# Capitolo 3

## Manuale per l'uso

In questo capitolo viene presentato il manuale per l'uso del software S.P.Q.R. (*Selective Protocol for Quality and Reliability*) nel quale vengono riportati i principali esempi di funzionamento, con l'obiettivo di illustrare il comportamento del server e dei client in risposta ai comandi inviati.

### 3.1 Introduzione

Per facilitare la distribuzione e l'utilizzo del software, è stato redatto un manuale d'uso che descrive le funzionalità principali del progetto. Il manuale è suddiviso in sezioni, ognuna delle quali descrive un aspetto specifico del software. In particolare, quest'ultimo fornisce informazioni su come installare e configurare il software, con un focus particolare sulle funzionalità offerte dal server e dai client. Inoltre vengono forniti esempi di utilizzo dei comandi principali, al fine di illustrare il comportamento del sistema in risposta alle richieste degli utenti. Infine è possibile contribuire al progetto, segnalando eventuali bug o suggerendo nuove funzionalità da implementare, attraverso il [repository GitHub ufficiale](#).

#### 3.1.1 Download e installazione

Per installare il software S.P.Q.R. è necessario scaricare il codice sorgente. Per fare ciò, è possibile clonare il repository GitHub del progetto, eseguendo il seguente comando da terminale:

```
git clone https://github.com/AntonioBerna/spqr.git
```

al termine del download, è possibile accedere alla directory del progetto, eseguendo il comando:

```
cd spqr/
```

All'interno di quest'ultima è presente il file `Makefile`, che permette di compilare il codice sorgente e generare i file eseguibili. In particolare, sono disponibili i seguenti comandi:

```
make          # Compilazione del codice sorgente in release mode
make debug    # Compilazione del codice sorgente in debug mode
make clean    # Pulizia dei file generati dalla compilazione
```

La compilazione in release mode è consigliata per l'utilizzo del software in produzione, mentre la compilazione in debug mode è utile per il testing e il debugging del codice, in quanto abilita la stampa di messaggi di log aggiuntivi. Una volta che il codice sorgente

è stato compilato con successo viene generata una directory `bin/`, all'interno della quale sono presenti i file eseguibili `spqr-server` e `spqr-client`. In particolare, se tutto è andato a buon fine, si ottiene il seguente output:

```
Build in release mode completed. Run with ./bin/spqr-server and ./bin/spqr-client <IPv4>
```

## Installazione avanzata

Per impostazione predefinita, il software S.P.Q.R. viene compilato utilizzando il compilatore `clang` con alcune opzioni di compilazione predefinite, che per semplicità sono riportate di seguito:

```
CC=clang
CFLAGS=-Wall -Wextra -Werror -pedantic
```

Lo stesso discorso vale per la directory di output, che viene impostata di default come segue:

```
BINARY_DIR=bin/
```

Tuttavia, è possibile personalizzare le opzioni di compilazione e la directory di output, modificando il file `Makefile` e aggiungendo le opzioni desiderate, tenendo presente che il software è stato testato con successo utilizzando il compilatore `clang` e le opzioni di compilazione predefinite.

### 3.1.2 Disinstallazione

Per disinstallare definitivamente il software S.P.Q.R. è sufficiente eliminare la directory del progetto, eseguendo il seguente comando da terminale:

```
rm -r spqr/
```

Invece, per pulire i file generati dalla compilazione e mantenere solo i file sorgente, è possibile eseguire il comando:

```
make clean
```

### 3.1.3 Configurazione del software

Il software S.P.Q.R. è stato progettato per essere configurabile, in modo da adattarsi alle esigenze degli utenti, infatti è possibile configurarlo modificando i parametri presenti nel file `include/settings.h`. In particolare è possibile configurare la porta di ascolto del server:

```
#define PORT 6969
```

scegliendo un valore compreso tra `1024` e `65535`. Inoltre è possibile configurare il percorso delle directory del server e del client, avendo cura di modificare non solo il nome della directory, ma anche di aggiungere il carattere `/` alla fine del percorso:

```
#define SERVER_FILES_PATH "server-files/"
#define CLIENT_FILES_PATH "client-files/"
```

Per impostazione predefinita, quando si scarica il codice sorgente del progetto, le directory `server-files/` e `client-files/` sono presenti nella directory principale del progetto. Tuttavia se si decide di modificare il nome delle directory o il percorso, è necessario assicurarsi che le directory esistano e siano accessibili in lettura e scrittura. È possibile configurare il simbolo che identifica i pacchetti:

```
#define PACKAGE_EMOJI "📦"
```

il timeout statico, il timeout adattivo, la dimensione della finestra, la probabilità di perdita dei pacchetti, il numero massimo di errori prima della chiusura della connessione, il timeout di connessione e il numero di tentativi di connessione:

```
#define TIMEOUT 8000 // ? Configurable Static Timeout (8 ms)
#define LOSS_PROBABILITY 25 // ? Range [0, 80]
#define WINDOW_SIZE 32 // ? Range [8, 128]
#define ADAPTIVE true // ? Configurable Adaptive Timeout
#define TIME_UNIT 4000 // ? 4 ms
#define MAX_TIMEOUT 80000 // ? 80 ms
#define MIN_TIMEOUT 8000 // ? 8 ms
#define MAX_ERRORS 25 // ? Maximum errors before closing the connection
#define TIMEOUT_CONNECTION 3600 // ? 1 hour
#define CONNECTION_ATTEMPTS 3 // ? Number of connection attempts
```

### 3.1.4 Esecuzione del software e comandi disponibili

Una volta che il software è stato compilato in release mode con successo, è possibile eseguire il server e i client, utilizzando la seguente procedura. Innanzitutto è necessario avviare il server utilizzando il seguente comando:

```
./bin/spqr-server
```

ottenendo:

```

      _ _ _ _ _ 
     /   \   /   \   /   \   /   \  
    /____\ /____\ /____\ /____\ 
    |elective|protocol for|quality and|reliability
    Developed by Antonio Bernardini & Flavio Caporilli
    🌐 Server is listening for incoming connections.
```

Il server per impostazione predefinita utilizza l'indirizzo IPv4 `127.0.0.1` (localhost) e la porta `6968`, finchè rimane in attesa di connessioni da parte dei client. La porta `6968` è dedicata all'attesa della connessione da parte di un client, mentre la porta `6969` viene utilizzata per il trasferimento dei file e per lo scambio di messaggi tra il server e il primo client connesso. Di conseguenza, ogni client che si collega riceverà una porta specifica: il secondo client sarà assegnato alla porta `6970`, il terzo alla `6971` e così via. Quando un client si connette al server con successo, il server stampa il seguente messaggio:

```
The client #0 is connected on port 6969.
```

Mentre, per avviare un client è necessario eseguire il seguente comando:

```
./bin/spqr-client <IPv4>
```

ottenendo:

```
S    D    A    L  
|____/elective |__|protocol for \_\_\quality and |_|\eliability
```





Developed by Antonio Bernardini & Flavio Caporilli

Connection established with the server.

dove `<IPv4>` è l'indirizzo IPv4 del server, ad esempio `127.0.0.1` (localhost), per connettersi al server in locale, oppure `10.2.2.15` per connettersi ad una macchina virtuale in rete. A differenza del server, i client forniscono i parametri di configurazione utilizzati per la connessione:

CONFIGURATION PARAMETERS	
WINDOW_SIZE	= 32
LOSS_PROBABILITY	= 25%
TIMEOUT	= 8000
ADAPTIVE	= true

ed inoltre forniscono un prompt interattivo, che permette di inviare i comandi al server:

Command	Description
 LIST	List the files available on the server
 GET	Download a file from the server
 PUT	Upload a file to the server
 CLOSE	Close the connection

```
[client@spqr ~]$
```

Si noti la presenza del comando `close`, che permette di chiudere la connessione tra il server e il client. In particolare il client viene terminato correttamente:

```
[client@spqr ~]$ close
```

```
Bye bye! 🙋
```

mentre il server rimane in attesa di connessioni da parte di altri client, annotando nella memoria condivisa che si è liberata una posizione, utilizzando il seguente messaggio:

```
Closed connection for client #0 on port 6969 with exit code 0.
```

Infine si noti la presenza degli altri comandi `LIST`, `GET` e `PUT`, che permettono di listare i file disponibili sul server, scaricare un file dal server e caricare un file sul server, rispettivamente.

## 3.2 Esempi di funzionamento

### 3.2.1 Esempio d'uso del comando `LIST`

Una volta che la connessione client-server è stata stabilita con successo, è possibile utilizzare il comando `LIST` per ottenere la lista dei file disponibili sul server:

```
[client@spqr ~]$ list
```

```
List of files availables on the server:
```

```
📄 Progetto2324.pdf
📄 README.md
📄 GuidaC.txt
```





pacchetti SYN, SYN-ACK e ACK, come mostrato in Fig. 3.1.

The screenshot shows the Wireshark network protocol analyzer interface. The main window displays a list of 12 captured packets. The packet list pane shows the following details:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	45	54377 → 6968 Len=3
2	0.000171722	127.0.0.1	127.0.0.1	UDP	48	6968 → 54377 Len=6
3	0.000292295	127.0.0.1	127.0.0.1	UDP	45	54377 → 6968 Len=3
4	0.000405929	127.0.0.1	127.0.0.1	UDP	46	6968 → 54377 Len=4
5	10.902612284	127.0.0.1	127.0.0.1	UDP	45	39403 → 6969 Len=4
6	10.903163009	127.0.0.1	127.0.0.1	UDP	278	6969 → 39403 Len=236
7	13.758661460	127.0.0.1	127.0.0.1	UDP	47	39403 → 6969 Len=5
8	13.758656996	127.0.0.1	127.0.0.1	UDP	47	6969 → 39403 Len=5
9	13.758968821	127.0.0.1	127.0.0.1	UDP	45	39403 → 6969 Len=3
10	13.759140348	127.0.0.1	127.0.0.1	UDP	48	6969 → 39403 Len=6
11	13.759170670	127.0.0.1	127.0.0.1	UDP	45	6969 → 39403 Len=3
12	13.759233175	127.0.0.1	127.0.0.1	UDP	48	39403 → 6969 Len=6

The packet details pane is currently empty, and the packet bytes pane shows hex and ASCII data. The status bar at the bottom indicates 12 packets, 3 selected (25.00%), and the default profile.

SYN , SYN-ACK e ACK

la porta `6969` dal server al client, come mostrato in Fig. 3.2.

The image displays the Wireshark network protocol analyzer interface. The top menu bar includes File, Edit, View, Go, Capture, Analyze, Statistics, Telephony, Wireless, Tools, and Help. Below the menu is a toolbar with icons for file operations, packet selection, and analysis. The main packet list pane shows a table of captured packets:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	45	54377 → 6968 Len=3
2	0.000171722	127.0.0.1	127.0.0.1	UDP	48	6968 → 54377 Len=6
3	0.000292295	127.0.0.1	127.0.0.1	UDP	45	54377 → 6968 Len=3
4	0.000485929	127.0.0.1	127.0.0.1	UDP	48	6968 → 54377 Len=4
5	10.902812284	127.0.0.1	127.0.0.1	UDP	46	39403 → 6969 Len=4
6	10.903163009	127.0.0.1	127.0.0.1	UDP	278	6969 → 39403 Len=236
7	13.758661460	127.0.0.1	127.0.0.1	UDP	47	39403 → 6969 Len=5
8	13.758856996	127.0.0.1	127.0.0.1	UDP	47	6969 → 39403 Len=5
9	13.758988821	127.0.0.1	127.0.0.1	UDP	45	39403 → 6969 Len=3
10	13.759140348	127.0.0.1	127.0.0.1	UDP	48	6969 → 39403 Len=6
11	13.759170670	127.0.0.1	127.0.0.1	UDP	45	6969 → 39403 Len=3
12	13.759233175	127.0.0.1	127.0.0.1	UDP	48	39403 → 6969 Len=6

The bottom pane shows the details of the selected packet (No. 4). It includes the following information:

- Frame 4: 46 bytes on wire (368 bit)
- Ethernet II, Src: 00:00:00:00:00:00, Dst: 00:00:00:00:00:00
- User Datagram Protocol, Src Port: 39403, Dst Port: 6969
- Data (4 bytes): 00 00 00 00

The status bar at the bottom indicates the current filter is 'list.pcapng', the number of packets is 12, and the profile is Default.

inviata dal server al client

Il server risponde con la lista dei file disponibili, come mostrato in Fig. 3.3.

The image shows a Wireshark packet capture. The top pane displays a list of packets. The bottom pane shows the details of the selected packet (No. 10).

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	45	54377 → 6968 Lemn3
2	0.000011722	127.0.0.1	127.0.0.1	UDP	48	6968 → 54377 Lemn3
3	0.000252295	127.0.0.1	127.0.0.1	UDP	45	54377 → 6968 Lemn3
4	0.000405929	127.0.0.1	127.0.0.1	UDP	48	6968 → 54377 Lemn3
5	0.000621284	127.0.0.1	127.0.0.1	UDP	48	39403 → 6969 Lemn3
6	0.000631300	127.0.0.1	127.0.0.1	UDP	22	6969 → 39403 Lemn3
7	13.758601460	127.0.0.1	127.0.0.1	UDP	47	39403 → 6969 Lemn5
8	13.758856996	127.0.0.1	127.0.0.1	UDP	47	6969 → 39403 Lemn5
9	13.758908821	127.0.0.1	127.0.0.1	UDP	45	39403 → 6969 Lemn5
10	13.759140348	127.0.0.1	127.0.0.1	UDP	48	6969 → 39403 Lemn5
11	13.759170670	127.0.0.1	127.0.0.1	UDP	45	6969 → 39403 Lemn5
12	13.759233175	127.0.0.1	127.0.0.1	UDP	48	39403 → 6969 Lemn5

The bottom pane shows the details of the selected packet (No. 10). The packet is a UDP packet from 127.0.0.1 to 127.0.0.1, port 6969 to 39403. The packet length is 48 bytes. The packet contains a single byte of data, which is the character 'E'.

Figura 3.3: Lista dei file disponibili inviata dal server al client

Infine viene chiusa la connessione tra il client e il server. In particolare viene usato il comando `CLOSE` per chiudere la connessione, tramite i pacchetti `FIN`, `FIN-ACK`, `FIN` e `FIN-ACK`, come mostrato in Fig. 3.4.

The screenshot displays the Wireshark network protocol analyzer interface. The main window shows a packet list table with the following data:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.860609000	127.0.0.1	127.0.0.1	UDP	45	54377 → 6968 Len=3
2	0.00017122	127.0.0.1	127.0.0.1	UDP	48	6968 → 54377 Len=6
3	0.000292295	127.0.0.1	127.0.0.1	UDP	45	54377 → 6968 Len=3
4	0.000405929	127.0.0.1	127.0.0.1	UDP	48	6968 → 54377 Len=4
5	10.902812284	127.0.0.1	127.0.0.1	UDP	48	39403 → 6968 Len=4
6	10.903163009	127.0.0.1	127.0.0.1	UDP	278	6968 → 39403 Len=236
7	11.758031309	127.0.0.1	127.0.0.1	UDP	47	6968 → 39403 Len=5
8	13.758856996	127.0.0.1	127.0.0.1	UDP	47	6968 → 39403 Len=5
9	13.758988821	127.0.0.1	127.0.0.1	UDP	45	39403 → 6968 Len=3
10	13.759140348	127.0.0.1	127.0.0.1	UDP	48	6968 → 39403 Len=6
11	13.759170670	127.0.0.1	127.0.0.1	UDP	45	6968 → 39403 Len=3
12	13.759233175	127.0.0.1	127.0.0.1	UDP	48	39403 → 6968 Len=6

The status bar at the bottom indicates the file is 'list.pcapng', there are 12 packets, 0 are selected (50.0%), and the profile is 'Default'.

Figura 3.4: Three-Way Handshake: CLOSE, FIN, FIN-ACK, FIN e FIN-ACK

### 3.2.2 Esempio d'uso del comando GET

Utilizzando il comando `GET` è possibile effettuare il download di un file dal server. Per usare tale comando è richiesto l'utilizzo del comando `LIST` per ottenere la lista dei file

```
? Filename not found.
```

Se il file richiesto è presente nella directory `server-files/` e la probabilità di perdita dei pacchetti è del 20%, il file viene scaricato con successo:

FILE TRANSFER COMPLETED	
TIME	= 2.429695 sec
THROUGHPUT	= 825.52 KB/s

Se il file richiesto è presente nella directory `server-files/` e la probabilità di perdita dei pacchetti è dell'80%, il file non viene scaricato:

FILE TRANSFER FAILED	
TIME	= 18.794660 sec
THROUGHPUT	= 8.18 KB/s

Utilizzando il comando `PUT` è possibile effettuare l'upload di un file sul server. Nel caso in cui il file non sia presente sul client, viene stampato il seguente messaggio:

Mentre se il file è già presente sul server, viene stampato il seguente messaggio:

Se il file richiesto è presente nella directory `client-files/` e la probabilità di perdita dei pacchetti è del 20%, il file viene caricato con successo:

FILE TRANSFER COMPLETED	
TIME	= 2.149967 sec
THROUGHPUT	= 932.93 KB/s

FILE TRANSFER FAILED	
TIME	= 17.066263 sec
THROUGHPUT	= 117.53 KB/s

```
[client@spqr ~]$ close
```

```
Bye bye! 🙋
```

e il server stampa il seguente messaggio:

```
Closed connection for client #0 on port 6969 with exit code 0.
```

Viceversa, se il server viene chiuso con `ctrl+c`, esso comunica a tutti i client connessi che la connessione è stata chiusa, quindi solo nel momento in cui un client tenta di comunicare con il server (inviando un comando) si ottiene il seguente messaggio:

```
[client@spqr ~]$ list
```

```
🔴 Session terminated by server.
```

```
Bye bye! 🙋
```

Quando tutti i client si sono disconnessi, il server si chiude correttamente con il seguente messaggio:

```
Bye bye! 🙋
```

# Capitolo 4

## Valutazione delle prestazioni

In questo capitolo verranno descritti i test effettuati per valutare le prestazioni del sistema. In modo particolare, verranno analizzati i tempi di trasferimento di un file di dimensione fissa, tra due host connessi sulla stessa rete, al variare di alcune configurazioni del sistema. Infine verranno presentati i risultati ottenuti e le considerazioni finali.

### 4.1 Ambiente di test

Nonostante il software S.P.Q.R. sia *cross-platform (Unix-based)*, i test delle performance che sono illustrati in questo capitolo sono stati effettuati su un ThinkPad T480 con le seguenti specifiche:

- Processore Intel Core i7-8550U
- 32GB di RAM DDR4
- Sistema Operativo Linux Manjaro v25.0.0

#### 4.1.1 Ambiente Linux

Il software è stato sviluppato e testato utilizzando i seguenti strumenti:

- `clang` v19.1.7
- Valgrind v3.24.0
- Visual Studio Code v1.97.2
- Wireshark v4.4.3

dove `clang` è stato utilizzato come compilatore, `valgrind` per il controllo dei memory leak e Visual Studio Code come ambiente di sviluppo.

#### 4.1.2 Ambiente MacOS ARM e Intel

Inoltre il software è stato testato su un MacBook Air del 2020 con le seguenti specifiche:

- Processore Apple M1

- 8GB di RAM DDR4
- MacOS Sequoia

e su un MacBook Pro del 2015 con le seguenti specifiche:

- Processore Intel Core i5
- 16GB di RAM
- MacOS Monterey

dove per entrambi si è utilizzata l'ultima versione disponibile del compilatore `clang`, ovvero la versione `v16.0.0`. Infine si è scelto di non includere i risultati dei test svolti in ambiente Apple, in quanto risultano essere simili a quelli ottenuti su Linux e quindi non fornirebbero alcun valore aggiunto.

## 4.2 Test effettuati

Per valutare le prestazioni del sistema sono stati effettuati diversi test di trasferimento utilizzando il file di testo chiamato `GuidaC.txt` avente dimensione fissa pari a 2.1MB.

### 4.2.1 Test Timeout Adattivo

Durante l'esecuzione di questo test sono stati utilizzati dei valori della probabilità di perdita nell'intervallo  $[0\%, 80\%]$ , una dimensione della finestra nell'intervallo  $[8, 128]$  e un timeout adattivo impostato nel range  $[8000\mu s, 80000\mu s]$ . Mettendo insieme tutti i risultati ottenuti dai test è stato possibile costruire la seguente tabella:

Tabella 4.1: Throughput in funzione della finestra e della probabilità di perdita.

Timeout Adattivo: 8000 $\mu s$ – 80000 $\mu s$		Dimensione della Finestra				
		8	16	32	64	128
Probabilità di Perdita	0%	34625.34	35602.67	34971.78	32822.13	16168.72
	5%	1670.61	2394.67	3653.44	6061.48	7704.30
	10%	965.88	1546.64	2318.92	3808.67	5816.49
	15%	695.77	1078.24	1831.08	2836.82	4606.03
	20%	521.22	827.01	1404.64	2348.13	3766.66
	25%	373.66	622.92	993.31	1827.67	2970.95
	30%	312.59	500.43	842.40	1495.68	2421.00
	40%	187.29	339.71	518.07	1037.10	1778.96
	60%	12.99	41.44	135.48	312.31	589.83
	80%	2.88	4.59	7.49	17.72	54.92

Valori espressi in kB/s

Successivamente, grazie ad uno script automatizzato in Python, è stato possibile costruire il grafico mostrato in Fig. 4.1.

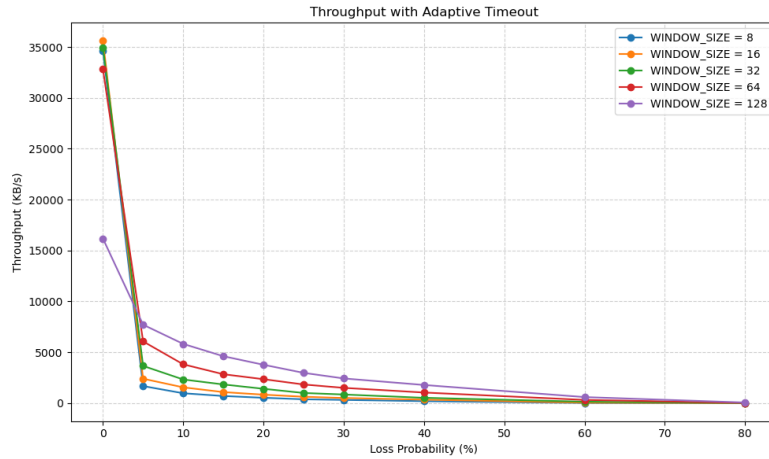


Figura 4.1: Throughput in funzione della finestra e della probabilità di perdita.

#### 4.2.2 Test Timeout Statico

Durante l'esecuzione di questo test sono stati utilizzati dei valori della probabilità di perdita nell'intervallo  $[0\%, 90\%]$ , una dimensione della finestra `WINDOW_SIZE` fissa a `32` e un timeout statico impostato nel range  $[4000\mu s, 80000\mu s]$ . Mettendo insieme tutti i risultati ottenuti dai test è stato possibile costruire la seguente tabella:

Tabella 4.2: Throughput in funzione del timeout statico e della probabilità di perdita.

Dim. finestra pari a 32		Timeout Statico (in $\mu s$ )					
		4000	8000	16000	32000	64000	80000
Probabilità di Perdita	0%	39653.63	32443.26	28928.45	34412.50	42966.73	36106.52
	5%	4727.65	3576.49	2490.50	1277.62	835.57	685.13
	10%	3500.83	2502.25	1768.88	909.38	525.79	425.64
	15%	2714.11	1914.10	1191.13	655.45	348.69	283.20
	20%	2206.86	1512.28	892.03	585.37	267.25	254.48
	25%	1694.11	1068.69	690.16	382.73	218.91	164.52
	30%	1304.02	926.47	562.95	309.86	170.61	132.60
	40%	830.81	578.49	355.41	204.92	108.62	88.31
	60%	324.29	221.31	133.52	75.07	39.99	32.43
	80%	75.57	50.82	30.73	17.16	9.09	7.36
	90%	18.94	12.70	7.62	4.25	2.25	1.35

Valori espressi in kB/s

Successivamente, grazie ad uno script automatizzato in Python, è stato possibile costruire il grafico mostrato in Fig. 4.2.



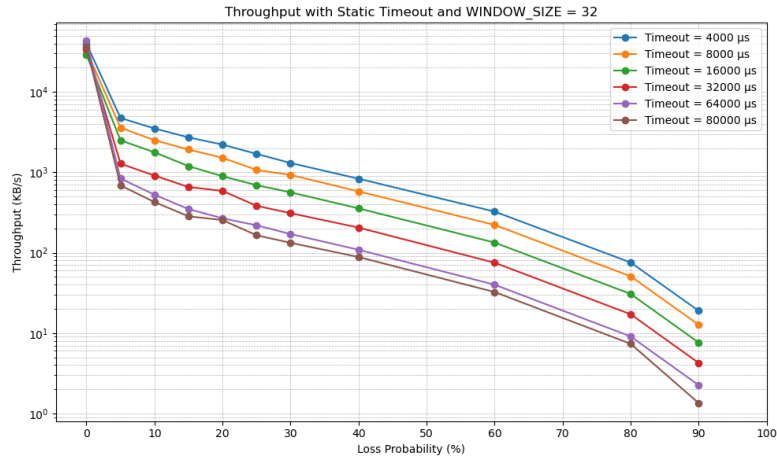


Figura 4.2: Throughput in funzione del timeout statico e della probabilità di perdita.

### 4.2.3 Test Timeout Statico vs Timeout Adattivo

Durante l'esecuzione di questo test sono stati riutilizzati i precedenti valori calcolati per il timeout statico e il timeout adattivo e sono stati messi a confronto. Inoltre è stata aggiunta una colonna relativa alla cumulazione di errore per quanto riguarda il timeout adattivo. Mettendo insieme tutti i risultati ottenuti dai test è stato possibile costruire la seguente tabella:

Tabella 4.3: Confronto throughput fra timeout statico e timeout adattivo.

Dim. finestra pari a 32		Cumulazione Errori	Timeout Adattivo	Timeout Statici (in $\mu s$ )		
				8000	32000	80000
Probabilità di Perdita	0%	0	34971.78	32443.26	34412.50	36106.52
	5%	34 – 45	3653.44	3576.49	1277.62	685.13
	10%	53 – 62	2318.92	2502.25	909.38	425.64
	15%	73 – 88	1831.08	1914.10	655.45	283.20
	20%	89 – 108	1404.64	1512.28	585.37	254.48
	25%	128 – 138	993.31	1068.69	382.73	164.52
	30%	164 – 181	842.40	926.47	309.86	132.60
	40%	257 – 280	518.07	578.49	204.92	88.31
	60%	693 – 715	135.48	221.31	75.07	32.43
	80%	2980 – 3082	7.49	50.82	17.16	7.36

Valori espressi in kB/s

Successivamente, grazie ad uno script automatizzato in Python, è stato possibile costruire il grafico mostrato in Fig. 4.3.

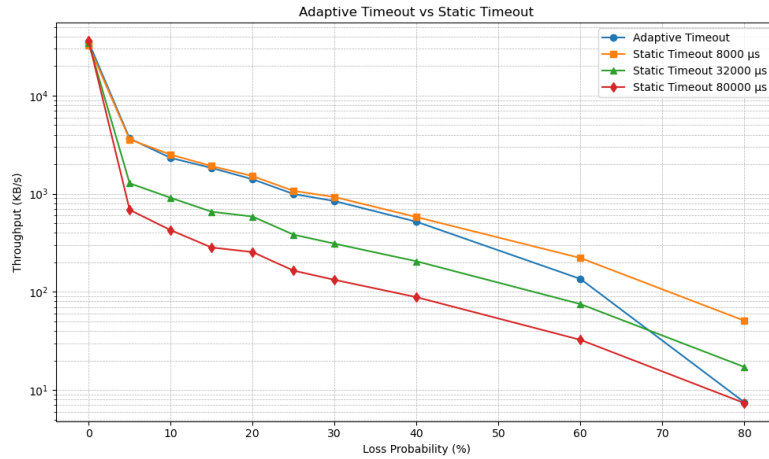


Figura 4.3: Confronto throughput fra timeout statico e timeout adattivo.

#### 4.2.4 Test cumulazione degli errori

Per questo test sono stati utilizzati i valori della cumulazione di errore calcolati per il timeout statico e il timeout adattivo e sono stati messi a confronto. Questi errori vengono memorizzati tramite la variabile `max_errors` che tiene traccia del numero massimo di errori consecutivi che si possono verificare durante il trasferimento di un file. Mettendo insieme tutti i risultati ottenuti dai test è stato possibile costruire la seguente tabella:

Tabella 4.4: Cumulazione degli errori in funzione della probabilità di perdita.

Comulazione Errori		Dimensione della Finestra				
		8	16	32	64	128
Probabilità di Perdita	0%	0	0	0	0	6
	5%	85 – 99	56 – 67	34 – 35	24 – 30	13 – 20
	10%	127 – 155	86 – 92	53 – 62	32 – 42	23 – 25
	15%	184 – 209	122 – 153	73 – 88	43 – 51	26 – 30
	20%	240 – 279	169 – 189	89 – 108	62 – 69	32 – 39
	25%	331 – 348	207 – 222	128 – 138	74 – 82	42 – 48
	30%	444 – 472	285 – 297	164 – 181	98 – 116	55 – 61
	40%	681 – 742	453 – 477	257 – 280	151 – 163	87 – 92
	60%	1889 – 1932	1202 – 1230	693 – 715	374 – 421	140 – 218
	80%	7963 – 8548	5028 – 5507	2980 – 3082	1682 – 1844	899 – 1150

Successivamente, grazie ad uno script automatizzato in Python, è stato possibile costruire i grafici mostrati in Fig. 4.4 e Fig. 4.5, dove per la cumulazione degli errori nel caso del timeout adattivo è stata considerata la colonna della Tabella 4.3.

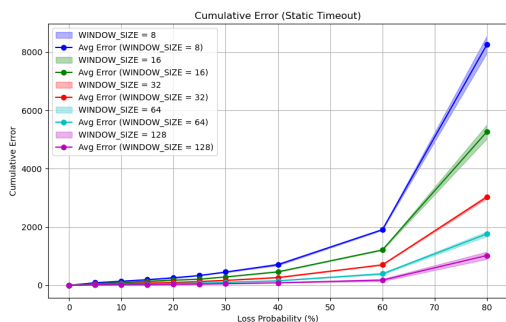


Figura 4.4: Comulazione degli errori per il timeout statico.

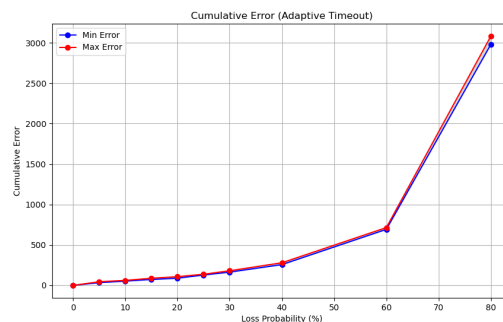


Figura 4.5: Comulazione degli errori per il timeout adattivo.

## 4.3 Test per l'integrità dei file trasferiti 🦆

Infine è stato svolto un test per verificare l'integrità dei file trasferiti. Per fare ciò è stato creato uno script in Python (presente nella directory `tests/`) che confronta il file originale e quello trasferito, verificando che siano identici. Prima di tutto, il programma calcola l'hash dei file utilizzando un algoritmo di hashing sicuro, come SHA-256, che permette di rilevare rapidamente eventuali differenze nel contenuto. L'hash viene calcolato leggendo il file a blocchi per ottimizzare l'efficienza anche con file di grandi dimensioni. Se gli hash di due file sono identici, il programma esegue un ulteriore confronto byte per byte per essere completamente sicuro che i file siano uguali. Il funzionamento si estende a due directory specificate dall'utente, denominate di default con `client-files/` e `server-files/`. Il programma analizza il contenuto di entrambe le directory, confrontando i file con lo stesso nome. Se due file hanno lo stesso nome ma contenuti differenti, il programma segnala la differenza, evidenziandola in rosso nel terminale. Se, invece, i file sono uguali, viene indicato un messaggio in verde, confermando che i file sono identici. Inoltre, il programma elenca anche i file che sono presenti solo in una delle due directory, fornendo un riepilogo completo delle differenze.

### 4.3.1 Esempio di funzionamento

Per esemplificare il funzionamento dello script, può essere usato il file di testo `GuidaC.txt`. Immaginando che questo file sia stato inviato dal cliente al server (o viceversa). Per eseguire lo script è sufficiente accedere alla root directory del progetto (ovvero `spqr/`) e lanciare il seguente comando:

```
python tests/integrity-consistency.py
```

Un esempio di output dello script è il seguente:

```
The files "GuidaC.txt" inside "./client-files/" and "./server-files/" are the same.
```