Trasferimento file su UDP
Go Back N

Progetto B1 - A.A. 2019/20

GIANMARCO MERLETTI
matricola 0240581

INTRODUZIONE

Presentazione del progetto

Il progetto consiste in una semplice applicazione di tipo *client-server* concorrente, che permette al client di scaricare file dal server attraverso un comando *get*, caricare dei propri files sul server stesso tramite un comando *put* e ottenere una lista di tutti i files presenti nel server con il comando *list*.

Questa applicazione per il trasferimento di file utilizza come protocollo di trasporto **UDP**, ovvero un protocollo che risulta essere di base non affidabile essendo senza connessione e privo di un meccanismo di riscontro. Lo scopo del progetto è stato quello di utilizzare UDP rendendolo un protocollo affidabile come TCP, attraverso un meccanismo di affidabilità basato sul protocollo Go-Back N, in modo da garantire una corretta trasmissione di file e messaggi tra client e server.

Protocollo **UDP**
Introduzione

Il protocollo UDP (User Datagram Protocol) consente alle applicazioni di scambiare messaggi singoli fornendo un livello di servizio minimo. Le sue caratteristiche principali sono:

- è un protocollo senza connessione;
- non supporta nessun tipo di meccanismo di riscontro/recupero in caso d'errore;
- implementa il servizio di checksum per verificare l'integrità dei dati;
- utilizzato per trasferimenti di dati semplici tra applicativi;
- risulta particolarmente adatto per applicazioni Real-Time grazie all'assenza di ritrasmissione, all'assenza di meccanismi di controllo di flusso e congestione e alla ridotta dimensione dell'header

Protocollo **UDP**

Differenze con TCP

Caratteristica	**Protocollo UDP**	**Protocollo TCP**
Descrizione	*Protocollo **semplice** ad alta velocità per il trasferimento di dati*	*Protocollo **completo** che permette di inviare dati in modo affidabile*
Connessione	***Connection-less*** *la trasmissione non necessita di nessun tipo di setup iniziale*	***Connection-oriented*** *la trasmissione richiede handshake iniziale*
Affidabilità	***Inaffidabile** e senza ACKs*	***Affidabile** con ACKs*
Ritrasmissione	***Non implementata***	***Implementata***
Controllo flusso dati	-	*Utilizzo di **finestra di invio scorrevole** e **algoritmi per il controllo di congestione***
Overhead	***Molto piccolo***	***Piccolo**, ma maggiore di UDP*
Velocità trasmissione	***Molto veloce***	***Veloce**, ma meno di UDP*
Dimensione dati idonea	***Bassa/media** quantità di dati*	***Media/alta** quantità di dati*

Protocollo **UDP**

Metodo Go-Back-N

Il metodo Go-Back N è uno dei modi più efficienti di effettuare una connessione, perché contrariamente al dovere aspettare che ogni frame invii il proprio riscontro tramite l'ACK, la connessione viene sfruttata più a lungo, inviando altri frame mentre aspetta i riscontri (invio a finestra scorrevole). In altre parole, durante il tempo che altrimenti sarebbe stato di attesa, vengono inviati frame aggiuntivi a prescindere dal risultato dell'invio precedente. Infatti, l'arrivo di un ACK relativo al frame che occupa l'estremo inferiore della finestra, genera la cancellazione di tale frame dal buffer di trasmissione e il conseguente "scorrimento" della finestra di una posizione in avanti, per permettere l'invio di un frame in più. Tuttavia, questo metodo può dare come risultato l'invio multiplo di frame già consegnati (nel caso in cui si perdesse un frame o un ACK). In questo caso il Go-Back-N ri-trasmette tutti i frame all'interno della finestra, anche quelli non necessari che dovranno essere scartati.

SCELTE DI IMPLEMENTAZIONE

Reliable UDP

Come rendere affidabile il protocollo UDP

Per rendere il protocollo UDP affidabile e garantire la corretta spedizione/ricezione dei messaggi, sono state implementate le seguenti funzionalità basate sul protocollo TCP (Go-Back-N):

- *Finestra di invio scorrevole* : permette l'invio di più trame consecutivamente senza attesa del riscontro
- *Sistema di riscontro corretta ricezione tramite ACK* : notifica la corretta ricezione di un frame
- *Timer per ogni frame inviato con timeout (fisso o adattativo)* : se scade il timeout prima dell'arrivo delle conferme, il trasmettitore ripete la trasmissione di tutti i frame non ancora confermati

Reliable UDP

Tipologie di pacchetti implementati

- **DATA_PKT** : pacchetto di dati del file. È caratterizzato da:
  - *Tipologia:* il valore **1** indica un **pacchetto dati classico**, il valore **2** indica un **pacchetto di terminazione**;
  - *Numero di sequenza*
  - *Lunghezza*
  - *Contenuto dati*
- **ACK_PKT** : pacchetto di riscontro. È caratterizzato da:
  - *Tipologia:* il valore **3** indica un **pacchetto ACK classico**, il valore **4** indica un **pacchetto ACK di terminazione**;
  - *Numero di sequenza* relativo al DATA_PKT riscontrato;

Reliable UDP

Tipologie di timeout implementati

- **Timeout FISSO** : valore costante scelto dal client
- **Timeout ADATTATIVO**: valore calcolato dinamicamente in base ai ritardi presenti sulla rete; tale valore viene stabilito attraverso l'uso di tre variabili:
  - **RTT** (*Round Trip Time*), ovvero il il tempo che intercorre dalla trasmissione di un singolo pacchetto alla ricezione del suo riscontro;
  - **SRTT** (*Smoothed Round Trip Time*), ovvero la media esponenziale di tutti gli RTT riscontrati fino a un certo istante;
  - **RTO** (*Retrasmission Time Out*), ovvero il tempo di ritrasmissione che consiste nel valore da assegnare al timer

  Questi valori vengono utilizzati nelle seguenti formule:

  - $RTO(k) = SRTT(k) \cdot \beta$ (dove β è una costante configurabile di default pari a 2 che deve essere >1)
  - $SRTT(k) = (1-\alpha) \cdot RTT + \alpha \cdot SRTT(k-1)$ (dove α è una costante configurabile di default pari a 0.9)

Reliable UDP

Gestione della concorrenza

Per garantire la *concorrenza* lato client/server, è stato necessario implementare un sistema di ***fork*** strutturate nel seguente modo:

- il server, quando viene generato, crea una prima socket di tipo UDP associata ad una specifica porta e resta in attesa del comando da eseguire. Una volta ricevuta la richiesta, viene creata una seconda socket UDP e il relativo numero di porta viene inviato al client attraverso la prima socket; successivamente il server esegue una fork per far svolgere l'azione richiesta dal client al figlio (al quale è assegnata appunto la seconda socket), e torna in ascolto di nuove richiesta con il processo padre.
- il client crea una prima socket UDP per comunicare con il server, chiede all'utente l'azione da svolgere e invia la richiesta ricevuta al server. Successivamente, riceve dal server il numero di porta associata al socket del figlio del server, e crea una seconda socket con quei dati. A questo punto esegue una fork e comunica con il server per svolgere l'azione scelta dall'utente. Infine, il processo padre elimina gli eventuali zombie e torna in attesa dell'utente per nuove richieste.

FUNZIONALITÀ OFFERTE

- comando ***get***
- comando ***put***
- comando ***list***
- comando ***exit***

Reliable UDP

Comando get

get [nome_file] - comando usato dal client per scaricare il file specificato dal server

Implementazione server

1. Ricevuta la richiesta dal client, il server controlla se il file richiesto è presente o meno. Se il file non è presente, il server non invia nulla al client.

2. Il server apre il file specificato, ottiene la dimensione del file e calcola il numero di pacchetti necessari per inviare il file.

3. Viene impostato come parametro di timeout il valore espresso dal client in fase di richiesta.

4. Vengono inviati tutti i frame in sequenza secondo il valore della finestra di invio specificata dal client e vengono ricevuti i relativi pacchetti di ACK. Inoltre, per ogni invio viene calcolato il nuovo valore di RTT e nel caso fosse stato richiesto l'uso del timeout adattativo, viene aggiornato il valore del timer. In caso di timeout, vengono ritrasmessi i pacchetti presenti nella finestra di spedizione corrente.

5. Terminati i pacchetti, viene inviato un pacchetto di terminazione a notificare il completamento dell'invio del file e il server resta in attesa del relativo ACK di terminazione da parte del client per poter completare l'operazione e mettersi in attesa di nuove richieste.

Reliable UDP

Comando get

get [nome_file] - comando usato dal client per scaricare il file specificato dal server

Codice server

```c
/*************** comando GET ***************/
if ( (strcmp(cmd, "get") == 0) && (name != '\0') )  {

    printf("SERVER [%d]: Comando GET per il file -> %s (loss_rate = %.2f)\n", getpid(), name, loss_rate);

    FILE *file;
    long length;

    /* ottengo percorso file completo */
    struct stat st;
    char path[1000];
    getcwd(path, sizeof(path));
    strcat(path, "/files/");
    if (stat(path, &st) < 0) {
        mkdir(path, 0700);
    }
    strcat(path, name);

    /* apro il file */
    file = fopen(path, "rb");
    if (file != NULL) {
        fseek(file, 0, SEEK_END);
        length = ftell(file);
        fseek(file, 0, SEEK_SET);

    } else {
        printf("ERRORE: File not found!\n");
        continue;
    }

    /* calcolo numero di pacchetti */
    int num_packets = length / (BUF_DIM);
    if ((length % (BUF_DIM)) != 0) {
        num_packets++;
    }
    printf("SERVER [%d]: Numero di pacchetti da inviare = %d\n", getpid(), num_packets);

    struct timeval RTT_start[num_packets];
    struct timeval RTT_end[num_packets];

    /* imposto il timer iniziale */
    RTO.tv_sec = 0;
    RTO.tv_usec = timeout_ms*1000;
    setsockopt(c_sockfd, SOL_SOCKET, SO_RCVTIMEO,(char*)&RTO, sizeof(RTO));

    int base = -1;          /* controllo ACK ricevuto */
    int pkt_number = 0;     /* controllo pacchetto inviato */
    int noFinalACK = 1;     /* controllo ricezione ACK di terminazione */

    while (noFinalACK) {

        while (pkt_number <= num_packets && (pkt_number - base) <= winsize) {
            struct DATA_PKT dataPacket;

            if (pkt_number == num_packets) {
                /* raggiunta la fine, creo il pacchetto di terminazione */
                dataPacket = createTerminalPacket(pkt_number, 0);
                printf("SERVER [%d]: Invio pacchetto di terminazione...\n", getpid());
            } else {
                /* creo il pacchetto */
                char pkt_data[BUF_DIM];
                int pkt_len;
                fseek(file, pkt_number*(BUF_DIM), SEEK_SET);
                pkt_len = fread(pkt_data, 1, BUF_DIM, file);

                dataPacket = createDataPacket(pkt_number, pkt_len, pkt_data);
                printf("SERVER [%d]: Invio pacchetto %d...\n", getpid(), pkt_number);
            }

            /* invio il pacchetto */
            if (sendto(c_sockfd, &dataPacket, sizeof(dataPacket), 0,(struct sockaddr *)&c_clnt_addr, c_addr_len) < 0) {
                printf("Errore in sendto()\n");
                exit(-1);
            }
            gettimeofday(&RTT_start[pkt_number], NULL);
            pkt_number++;
        }

        /* ricevo gli ACK per i pacchetti inviati */
        struct ACK_PKT ack;
        while ((c_recv_len = recvfrom(c_sockfd, &ack, sizeof(ack), 0, (struct sockaddr *)&c_clnt_addr, &c_addr_len)) < 0) {

            /* in caso di timeout, reinvio i pacchetti fino a esaurimento della finestra */
            pkt_number = base + 1;
            printf("\n!!! TIMEOUT per pacchetto %d !!!\n", pkt_number);

            if (tries >= MAX_TRIES) {
                printf("SERVER [%d]: Connessione con il client persa.\n". getpid());
                exit(0);
            }

            while (pkt_number <= num_packets && (pkt_number - base) <= winsize) {
                struct DATA_PKT dataPacket;

                if (pkt_number == num_packets) {
                    /* raggiunta la fine, creo il pacchetto di terminazione */
                    dataPacket = createTerminalPacket(pkt_number, 0);
                    printf("SERVER [%d]: Invio pacchetto di terminazione...\n", getpid());
                } else {
                    /* creo il pacchetto */
                    char pkt_data[BUF_DIM];
                    int pkt_len;
                    fseek(file, pkt_number*(BUF_DIM), SEEK_SET);
                    pkt_len = fread(pkt_data, 1, BUF_DIM, file);

                    dataPacket = createDataPacket(pkt_number, pkt_len, pkt_data);
                    printf("SERVER [%d]: Invio pacchetto %d...\n", getpid(), pkt_number);
                }

                /* invio il pacchetto */
                if (sendto(c_sockfd, &dataPacket, sizeof(dataPacket), 0,(struct sockaddr *)&c_clnt_addr, c_addr_len) < 0) {
                    printf("Errore in sendto()\n");
                    exit(-1);
                }
                gettimeofday(&RTT_start[pkt_number], NULL);
                pkt_number++;
            }
            tries++;

        }

        if (ack.type != 4) {
            printf("\nSERVER [%d]: ACK ricevuto per pacchetto %d\n", getpid(), ack.ack_no);

            gettimeofday(&RTT_end[ack.ack_no], NULL);
            timersub(&RTT_end[ack.ack_no], &RTT_start[ack.ack_no], &RTT);
            SRTT = calculateSRTT(SRTT, RTT, (float)ALPHA);
            RTO = calculateRTO(SRTT, (float)BETA);
            /* set timer */
            if (T_OUT == 'Y') {
                setsockopt(c_sockfd, SOL_SOCKET, SO_RCVTIMEO,(char*)&RTO, sizeof(RTO));
                printf("SERVER [%d]: Nuovo valore di timeout = %ld ms\n", getpid(), (RTO.tv_sec)*1000 + (RTO.tv_usec)/1000);
            }

            if(ack.ack_no > base){
                base = ack.ack_no;
            }
        } else {
            printf("\nSERVER [%d]: ACK di terminazione ricevuto\n", getpid());
            noFinalACK = 0;
        }

        tries = 0;
    }

    fclose(file);

    /* reset timer */
    setsockopt(c_sockfd, SOL_SOCKET, SO_RCVTIMEO,(char*)&reset, sizeof(reset));

    printf("\n\n*************************************\n");
    printf("*********    FILE INVIATO    *********\n");
    printf("*************************************\n\n");
}
```

Reliable UDP

Comando get

get [nome_file] - comando usato dal client per scaricare il file specificato dal server

Implementazione client

1. Il server non trasferisce alcun messaggio se il nome file è NULL o non è presente.

2. Vengono richiesti i seguenti parametri: dimensione della finestra di spedizione, la probabilità di perdita dei messaggi, la durata del timeout fisso e se si vuole l'utilizzo di un timer adattativo.

3. Il client crea il file, riceve tutti i pacchetti in sequenza, scrive il contenuto sul file appena creato e quindi notifica al server la corretta ricezione tramite gli ACK corrispondenti. Nel caso in cui il client riceva un pacchetto fuori sequenza, lo ignora. Inoltre, se il client riceve un pacchetto che per la probabilità di perdita dei messaggi deve essere "perso", lo scarta.

4. Quando riceve il pacchetto di terminazione dal server, invia il relativo ACK di terminazione, completando la ricezione e chiudendo il relativo processo figlio.

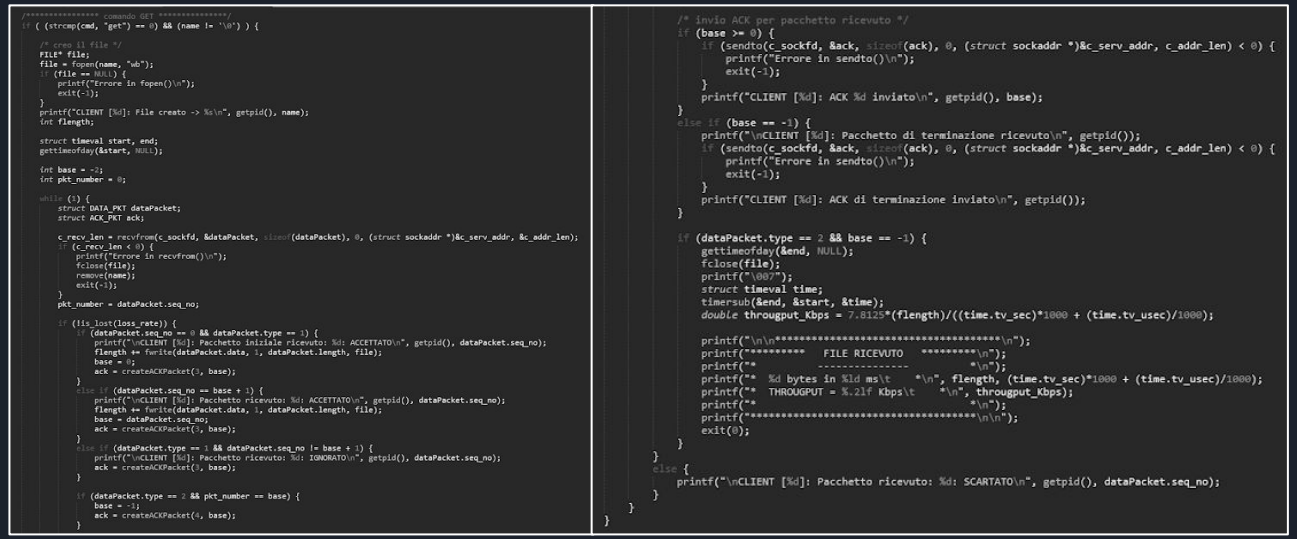
5. In seguito alla ricezione, vengono stampate a schermo le informazioni relative alla dimensione del file ricevuto [*bytes*], il tempo impiegato per il trasferimento [*millisec*] e il relativo throughput [*kilobit/sec*].

Reliable UDP

Comando get

get [nome_file] - comando usato dal client per scaricare il file specificato dal server

Codice client



```
/*************** comando GET ***************/
if ( (strcmp(cmd, "get") == 0) && (name != '\0') ) {

    /* crea il file */
    FILE* file;
    file = fopen(name, "wb");
    if (file == NULL) {
        printf("Errore in fopen()\n");
        exit(-1);
    }
    printf("CLIENT [%d]: File creato -> %s\n", getpid(), name);
    int flength;

    struct timeval start, end;
    gettimeofday(&start, NULL);

    int base = -2;
    int pkt_number = 0;

    while (1) {
        struct DATA_PKT dataPacket;
        struct ACK_PKT ack;

        c_recv_len = recvfrom(c_sockfd, &dataPacket, sizeof(dataPacket), 0, (struct sockaddr *)&c_serv_addr, &c_addr_len);
        if (c_recv_len < 0) {
            printf("Errore in recvFrom()\n");
            fclose(file);
            remove(name);
            exit(-1);
        }
        pkt_number = dataPacket.seq_no;

        if (!is_lost(loss_rate)) {
            if (dataPacket.seq_no == 0 && dataPacket.type == 1) {
                printf("\nCLIENT [%d]: Pacchetto iniziale ricevuto: %d: ACCETTATO\n", getpid(), dataPacket.seq_no);
                flength += fwrite(dataPacket.data, 1, dataPacket.length, file);
                base = 0;
                ack = createACKPacket(0, base);
            }
            else if (dataPacket.seq_no == base + 1) {
                printf("\nCLIENT [%d]: Pacchetto ricevuto: %d: ACCETTATO\n", getpid(), dataPacket.seq_no);
                flength += fwrite(dataPacket.data, 1, dataPacket.length, file);
                base = dataPacket.seq_no;
                ack = createACKPacket(0, base);
            }
            else if (dataPacket.type == 1 && dataPacket.seq_no != base + 1) {
                printf("\nCLIENT [%d]: Pacchetto ricevuto: %d: IGNORATO\n", getpid(), dataPacket.seq_no);
                ack = createACKPacket(0, base);
            }

            if (dataPacket.type == 2 && pkt_number == base) {
                base = -1;
                ack = createACKPacket(4, base);
            }
            /* invio ACK per pacchetto ricevuto */
            if (base >= 0) {
                if (sendto(c_sockfd, &ack, sizeof(ack), 0, (struct sockaddr *)&c_serv_addr, c_addr_len) < 0) {
                    printf("Errore in sendto()\n");
                    exit(-1);
                }
                printf("CLIENT [%d]: ACK %d inviato\n", getpid(), base);
            }
            else if (base == -1) {
                printf("\nCLIENT [%d]: Pacchetto di terminazione ricevuto\n", getpid());
                if (sendto(c_sockfd, &ack, sizeof(ack), 0, (struct sockaddr *)&c_serv_addr, c_addr_len) < 0) {
                    printf("Errore in sendto()\n");
                    exit(-1);
                }
                printf("CLIENT [%d]: ACK di terminazione inviato\n", getpid());
            }

            if (dataPacket.type == 2 && base == -1) {
                gettimeofday(&end, NULL);
                fclose(file);
                printf("\007");
                struct timeval time;
                timersub(&end, &start, &time);
                double throughput_Kbps = 7.8125*(flength)/((time.tv_sec)*1000 + (time.tv_usec)/1000);

                printf("\n\n*************************************\n");
                printf("*********   FILE RICEVUTO   *********\n");
                printf("*            ---------------        *\n");
                printf("*  %d bytes in %ld ms\t  *\n", flength, (time.tv_sec)*1000 + (time.tv_usec)/1000);
                printf("*  THROUGPUT = %.2lf Kbps\t  *\n", throughput_Kbps);
                printf("*                                   *\n");
                printf("*************************************\n\n");
                exit(0);
            }
        }
        else {
            printf("\nCLIENT [%d]: Pacchetto ricevuto: %d: SCARTATO\n", getpid(), dataPacket.seq_no);
        }
    }
}
```

Reliable UDP

Comando put

put [nome_file] - comando usato dal client per caricare file sul server

Implementazione server/client

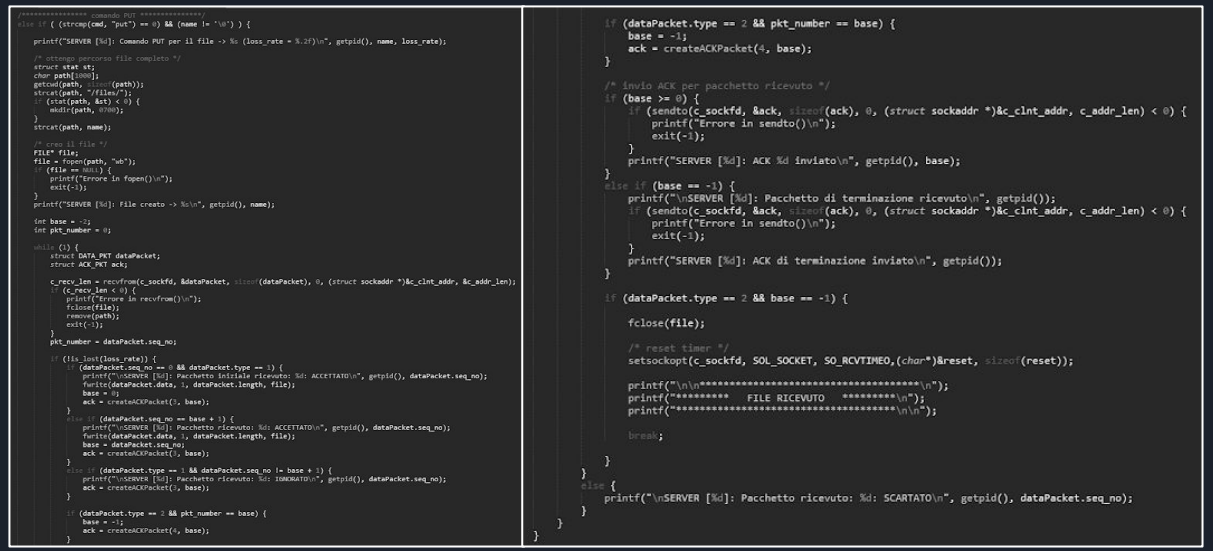
L'implementazione è simile a quella della richiesta get, tranne per il fatto che in questo caso il client agirà da mittente nel trasferimento del file, mentre il server riceverà il file fungendo da destinatario

Reliable UDP

Comando put

put [nome_file] - comando usato dal client per caricare file sul server

Codice server



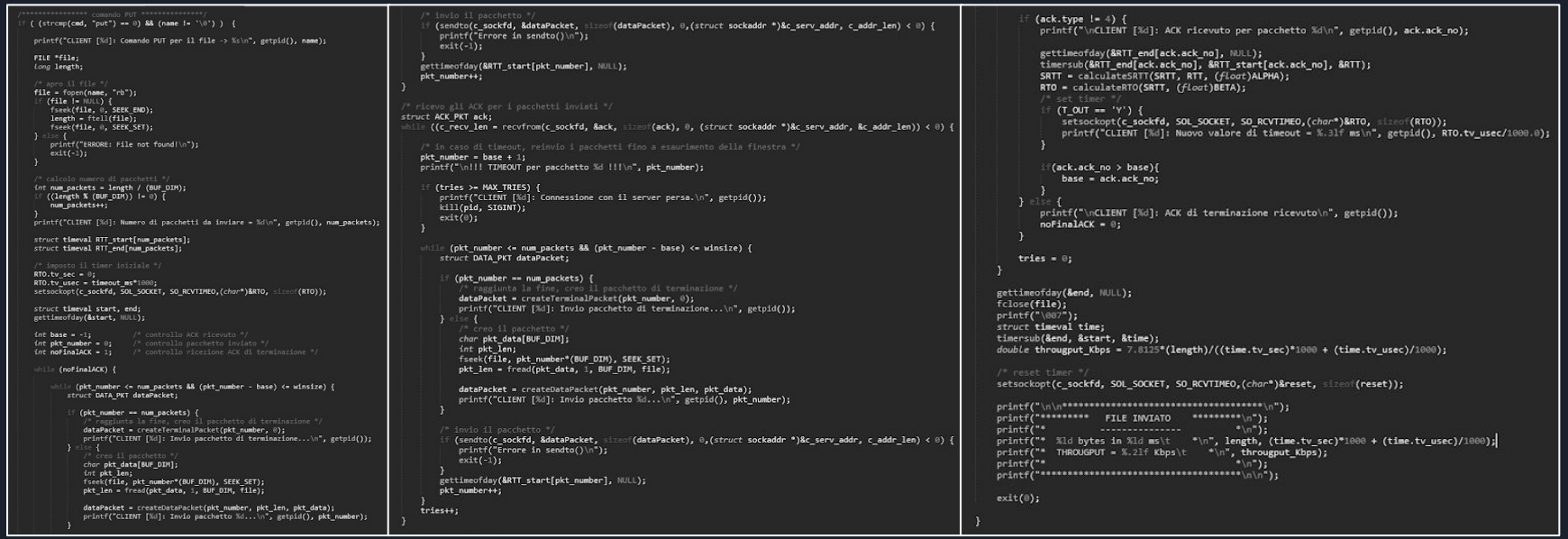
```
/************** comando PUT **************/
else if ( (strcmp(cmd, "put") == 0) && (name != '\0') ) {

    printf("SERVER [%d]: Comando PUT per il file -> %s (loss_rate = %.2f)\n", getpid(), name, loss_rate);

    /* ottengo percorso file completo */
    struct stat st;
    char path[1000];
    getcwd(path, sizeof(path));
    strcat(path, "/files/");
    if (stat(path, &st) < 0) {
        mkdir(path, 0700);
    }
    strcat(path, name);

    /* creo il file */
    FILE* file;
    file = fopen(path, "wb");
    if (file == NULL) {
        printf("Errore in fopen()\n");
        exit(-1);
    }
    printf("SERVER [%d]: File creato -> %s\n", getpid(), name);

    int base = -2;
    int pkt_number = 0;

    while (1) {
        struct DATA_PKT dataPacket;
        struct ACK_PKT ack;

        c_recv_len = recvfrom(c_sockfd, &dataPacket, sizeof(dataPacket), 0, (struct sockaddr *)&c_clnt_addr, &c_addr_len);
        if (c_recv_len < 0) {
            printf("Errore in recvfrom()\n");
            fclose(file);
            remove(path);
            exit(-1);
        }
        pkt_number = dataPacket.seq_no;

        if (!is_lost(loss_rate)) {
            if (dataPacket.seq_no == 0 && dataPacket.type == 1) {
                printf("\nSERVER [%d]: Pacchetto iniziale ricevuto: %d: ACCETTATO\n", getpid(), dataPacket.seq_no);
                fwrite(dataPacket.data, 1, dataPacket.length, file);
                base = 0;
                ack = createACKPacket(3, base);
            }
            else if (dataPacket.seq_no == base + 1) {
                printf("\nSERVER [%d]: Pacchetto ricevuto: %d: ACCETTATO\n", getpid(), dataPacket.seq_no);
                fwrite(dataPacket.data, 1, dataPacket.length, file);
                base = dataPacket.seq_no;
                ack = createACKPacket(3, base);
            }
            else if (dataPacket.type == 1 && dataPacket.seq_no != base + 1) {
                printf("\nSERVER [%d]: Pacchetto ricevuto: %d: IGNORATO\n", getpid(), dataPacket.seq_no);
                ack = createACKPacket(3, base);
            }

            if (dataPacket.type == 2 && pkt_number == base) {
                base = -1;
                ack = createACKPacket(4, base);
            }

            /* invio ACK per pacchetto ricevuto */
            if (base >= 0) {
                if (sendto(c_sockfd, &ack, sizeof(ack), 0, (struct sockaddr *)&c_clnt_addr, c_addr_len) < 0) {
                    printf("Errore in sendto()\n");
                    exit(-1);
                }
                printf("SERVER [%d]: ACK %d inviato\n", getpid(), base);
            }
            else if (base == -1) {
                printf("\nSERVER [%d]: Pacchetto di terminazione ricevuto\n", getpid());
                if (sendto(c_sockfd, &ack, sizeof(ack), 0, (struct sockaddr *)&c_clnt_addr, c_addr_len) < 0) {
                    printf("Errore in sendto()\n");
                    exit(-1);
                }
                printf("SERVER [%d]: ACK di terminazione inviato\n", getpid());
            }

            if (dataPacket.type == 2 && base == -1) {

                fclose(file);

                /* reset timer */
                setsockopt(c_sockfd, SOL_SOCKET, SO_RCVTIMEO,(char*)&reset, sizeof(reset));

                printf("\n\n*************************************\n");
                printf("*********   FILE RICEVUTO   *********\n");
                printf("*************************************\n\n");

                break;

            }
        }
        else {
            printf("\nSERVER [%d]: Pacchetto ricevuto: %d: SCARTATO\n", getpid(), dataPacket.seq_no);
        }
    }
}
```

Reliable UDP

Comando put

put [nome_file] - comando usato dal client per caricare file sul server

Codice client



```
/*************** comando PUT **************/
if ( (strcmp(cmd, "put") == 0) && (name != '\0') ) {

    printf("CLIENT [%d]: Comando PUT per il file -> %s\n", getpid(), name);

    FILE *file;
    long length;

    /* apro il file */
    file = fopen(name, "rb");
    if (file != NULL) {
        fseek(file, 0, SEEK_END);
        length = ftell(file);
        fseek(file, 0, SEEK_SET);
    } else {
        printf("ERRORE: File not found!\n");
        exit(-1);
    }

    /* calcolo numero di pacchetti */
    int num_packets = length / (BUF_DIM);
    if ((length % (BUF_DIM)) != 0) {
        num_packets++;
    }
    printf("CLIENT [%d]: Numero di pacchetti da inviare = %d\n", getpid(), num_packets);

    struct timeval RTT_start[num_packets];
    struct timeval RTT_end[num_packets];

    /* imposto il timer iniziale */
    RTO.tv_sec = 0;
    RTO.tv_usec = timeout_ms*1000;
    setsockopt(c_sockfd, SOL_SOCKET, SO_RCVTIMEO,(char*)&RTO, sizeof(RTO));

    struct timeval start, end;
    gettimeofday(&start, NULL);

    int base = -1;          /* controllo ACK ricevuto */
    int pkt_number = 0;     /* controllo pacchetto inviato */
    int noFinalACK = 1;     /* controllo ricezione ACK di terminazione */

    while (noFinalACK) {

        while (pkt_number <= num_packets && (pkt_number - base) <= winsize) {
            struct DATA_PKT dataPacket;

            if (pkt_number == num_packets) {
                /* raggiunta la fine, creo il pacchetto di terminazione */
                dataPacket = createTerminalPacket(pkt_number, 0);
                printf("CLIENT [%d]: Invio pacchetto di terminazione...\n", getpid());
            } else {
                /* creo il pacchetto */
                char pkt_data[BUF_DIM];
                int pkt_len;
                fseek(file, pkt_number*(BUF_DIM), SEEK_SET);
                pkt_len = fread(pkt_data, 1, BUF_DIM, file);

                dataPacket = createDataPacket(pkt_number, pkt_len, pkt_data);
                printf("CLIENT [%d]: Invio pacchetto %d...\n", getpid(), pkt_number);
            }
```

```
            /* invio il pacchetto */
            if (sendto(c_sockfd, &dataPacket, sizeof(dataPacket), 0,(struct sockaddr *)&c_serv_addr, c_addr_len) < 0) {
                printf("Errore in sendto()\n");
                exit(-1);
            }
            gettimeofday(&RTT_start[pkt_number], NULL);
            pkt_number++;
        }

        /* ricevo gli ACK per i pacchetti inviati */
        struct ACK_PKT ack;
        while ((c_recv_len = recvfrom(c_sockfd, &ack, sizeof(ack), 0, (struct sockaddr *)&c_serv_addr, &c_addr_len)) < 0) {

            /* in caso di timeout, reinvio i pacchetti fino a esaurimento della finestra */
            pkt_number = base + 1;
            printf("\n!!! TIMEOUT per pacchetto %d !!!\n", pkt_number);

            if (tries >= MAX_TRIES) {
                printf("CLIENT [%d]: Connessione con il server persa.\n", getpid());
                kill(pid, SIGINT);
                exit(0);
            }

            while (pkt_number <= num_packets && (pkt_number - base) <= winsize) {
                struct DATA_PKT dataPacket;

                if (pkt_number == num_packets) {
                    /* raggiunta la fine, creo il pacchetto di terminazione */
                    dataPacket = createTerminalPacket(pkt_number, 0);
                    printf("CLIENT [%d]: Invio pacchetto di terminazione...\n", getpid());
                } else {
                    /* creo il pacchetto */
                    char pkt_data[BUF_DIM];
                    int pkt_len;
                    fseek(file, pkt_number*(BUF_DIM), SEEK_SET);
                    pkt_len = fread(pkt_data, 1, BUF_DIM, file);

                    dataPacket = createDataPacket(pkt_number, pkt_len, pkt_data);
                    printf("CLIENT [%d]: Invio pacchetto %d...\n", getpid(), pkt_number);
                }

                /* invio il pacchetto */
                if (sendto(c_sockfd, &dataPacket, sizeof(dataPacket), 0,(struct sockaddr *)&c_serv_addr, c_addr_len) < 0) {
                    printf("Errore in sendto()\n");
                    exit(-1);
                }
                gettimeofday(&RTT_start[pkt_number], NULL);
                pkt_number++;
            }
            tries++;
        }
```

```
        if (ack.type != 4) {
            printf("\nCLIENT [%d]: ACK ricevuto per pacchetto %d\n", getpid(), ack.ack_no);

            gettimeofday(&RTT_end[ack.ack_no], NULL);
            timersub(&RTT_end[ack.ack_no], &RTT_start[ack.ack_no], &RTT);
            SRTT = calculateSRTT(SRTT, RTT, (float)ALPHA);
            RTO = calculateRTO(SRTT, (float)BETA);
            /* set timer */
            if (T_OUT == 'Y') {
                setsockopt(c_sockfd, SOL_SOCKET, SO_RCVTIMEO,(char*)&RTO, sizeof(RTO));
                printf("CLIENT [%d]: Nuovo valore di timeout = %.3lf ms\n", getpid(), RTO.tv_usec/1000.0);
            }

            if(ack.ack_no > base){
                base = ack.ack_no;
            }
        } else {
            printf("\nCLIENT [%d]: ACK di terminazione ricevuto\n", getpid());
            noFinalACK = 0;
        }

        tries = 0;
    }

    gettimeofday(&end, NULL);
    fclose(file);
    printf("\007");
    struct timeval time;
    timersub(&end, &start, &time);
    double throughput_Kbps = 7.8125*(length)/((time.tv_sec)*1000 + (time.tv_usec)/1000);

    /* reset timer */
    setsockopt(c_sockfd, SOL_SOCKET, SO_RCVTIMEO,(char*)&reset, sizeof(reset));

    printf("\n\n************************************\n");
    printf("*********   FILE INVIATO    *********\n");
    printf("**          ---------------         **\n");
    printf("**  %ld bytes in %ld ms\t    **\n", length, (time.tv_sec)*1000 + (time.tv_usec)/1000);
    printf("**  THROUGPUT = %.2lf Kbps\t    **\n", throughput_Kbps);
    printf("**                                **\n");
    printf("************************************\n\n");

    exit(0);

}
```

Reliable UDP

Comando list

list - comando usato dal client per ottenere la lista di tutti i file presenti sul server

Implementazione server/client

Il server esegue la scansione di tutti i file presenti nella specifica directory dove essi sono contenuti (“…/files/”) attraverso una chiamata di sistema, e invia l'intero elenco al client, il quale lo stampa a schermo.

```
/*************** comando LIST *************/
else if (strcmp(cmd, "list") == 0) {
    printf("SERVER [%d]: Comando LIST\n", getpid());
    FILE* proc = popen("ls files","r");
    int c = 0;
    int i = 0;
    memset(buffer, 0, BUF_DIM);
    while((c = fgetc(proc)) != EOF && i < (BUF_DIM-1)) {
        buffer[i++] = c;
    }
    buffer[i] = '\0';
    pclose(proc);
    if (sendto(c_sockfd, buffer, BUF_DIM, 0, (struct sockaddr*)&c_clnt_addr, c_addr_len) < 0) {
        perror("ERRORE: sendto()");
        exit(1);
    }
}
```

codice del server

```
/*************** comando LIST *************/
else if (strcmp(cmd, "list") == 0) {
    printf("\n-----------------------\n");
    printf("     LISTA DEI FILE     \n\n");
    char fileList[100];
    if (recvfrom(c_sockfd, fileList, sizeof(fileList), 0, (struct sockaddr *)&c_serv_addr, &c_addr_len) < 0) {
        perror("ERRORE: recvfrom()");
        exit(1);
    }
    printf("%s\n", fileList);
    printf("-----------------------\n");
    exit(0);
}
```

codice del client

Reliable UDP

Comando exit

exit - comando usato dal client per terminare

Implementazione server/client

Viene eseguita da parte del server e da parte del client una semplice chiamata alla funzione exit, in modo da chiudere la relativa connessione client/server

```
/*************** comando EXIT *************/
else if (strcmp(cmd, "exit") == 0) {
    printf("SERVER [%d]: Comando EXIT\n", getpid());
    exit(0);
}
```

codice del server

```
/*************** comando EXIT *************/
else if (strcmp(cmd, "exit") == 0) {
    printf("CLIENT [%d]: Comando EXIT\n", getpid());
    exit(0);
}
```

codice del client

CONFIGURAZIONE DEI PARAMETRI

Reliable UDP

Configurazione dei parametri

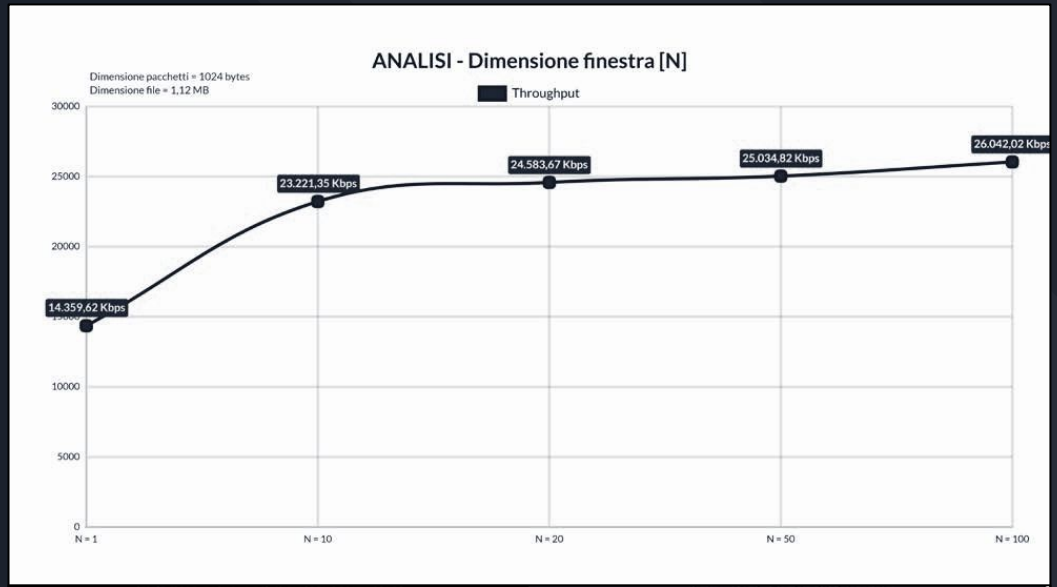
La *dimensione della finestra di spedizione*, la *probabilità di perdita dei messaggi* ***p***, la *durata del timeout fisso* e l'*utilizzo di un timer adattativo* sono parametri che possono essere settati in fase di richiesta di operazioni *get/put* da parte del client.
Ulteriori parametri che possono essere modificati all'interno del file "*util.h*" in fase di precompilazione sono:

- **SERV_PORT** - imposta il numero di porta da utilizzare per la prima socket UDP
- **BUF_DIM** - imposta la dimensione della parte dati di un pacchetto
- **WINSIZE** - imposta la dimensione di default della finestra di spedizione
- **ALPHA** - imposta il valore della costante α utilizzata nel calcolo dell'SRTT
- **BETA** - imposta il valore della costante β utilizzata nel calcolo dell'RTO.
- **MAX_TRIES** - imposta il numero massimo di tentativi di ritrasmissione di un pacchetto prima di chiudere la connessione

VALUTAZIONE DELLE PRESTAZIONI

Reliable UDP

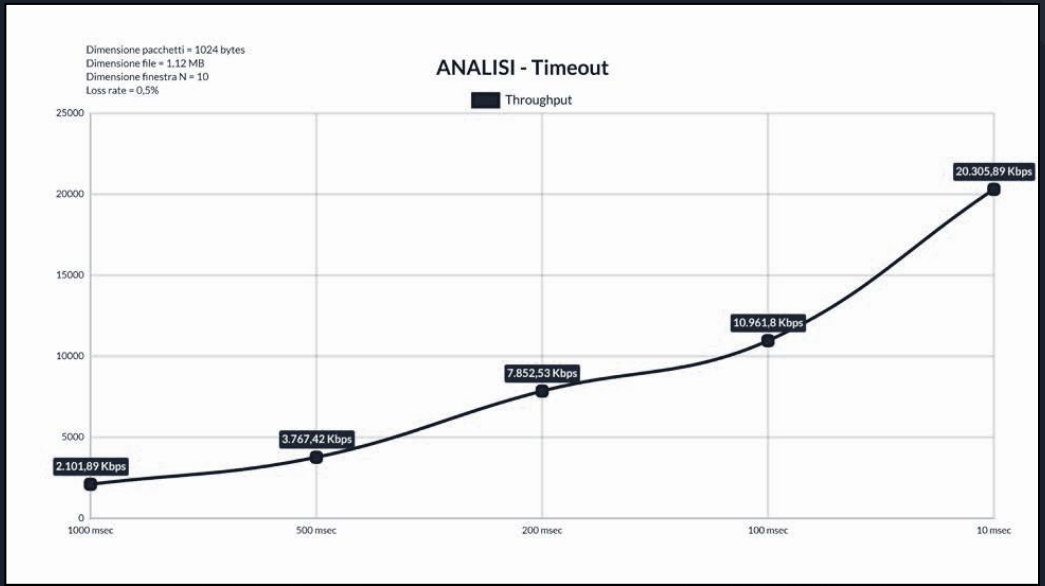
Prestazioni al variare della dimensione della finestra di spedizione



Dimensione finestra N	THROUGHPUT
1	14.359,62 Kbps
10	23.221,35 Kbps
20	24.538,67 Kbps
50	25.034,82 Kbps
100	26.042,02 Kbps

Reliable UDP

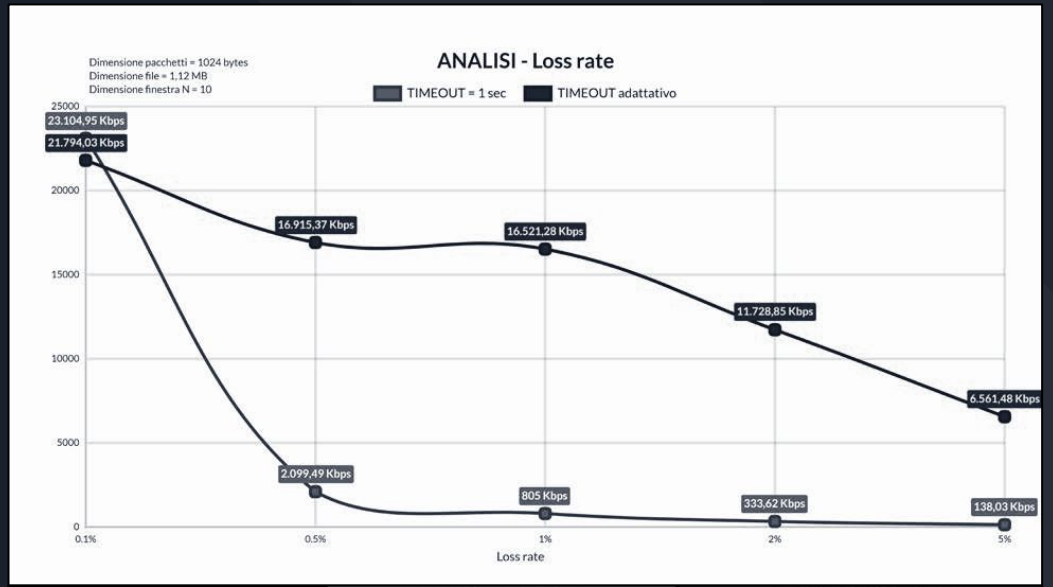
Prestazioni al variare della dimensione del valore del timeout fisso



Valore del timeout	THROUGHPUT
1000 msec	2.101,89 Kbps
500 msec	3.767,42 Kbps
200 msec	7.852,53 Kbps
100 msec	10.967,80 Kbps
10 msec	20.305,89 Kbps

Reliable UDP

Prestazioni al variare della probabilità di perdita di pacchetti in caso di timeout FISSO e in caso di timeout ADATTATIVO



Loss rate	THROUGHPUT timeout FISSO = 1 sec	THROUGHPUT timeout ADATTATIVO
0.1%	23.104,95 Kbps	21.794,03 Kbps
0.5%	2.099,49 Kbps	16.915,37 Kbps
1%	805 Kbps	16.521,28 Kbps
2%	333,62 Kbps	11.728,85 Kbps
5%	138,03 Kbps	6.561,48 Kbps

FINE