PROGETTO DI INGEGNERIA DI INTERNET E DEL WEB

DESIDERI GIULIA 0245061

L’applicazione realizza una comunicazione client-server per il traferimento di file tramite UDP.
La comunicazione avviene con protocollo Selective Repeat ed è resa sicura dall’uso di timeout per il rinvio di pacchetti e ack di risposta.

SCELTE PROGETTUALI:

- I pacchetti di richiesta, risposta e ack sono struct definite in “common.h”.
  La scelta di utilizzo di struct piuttosto che stringhe di bytes è dovuta alla maggior facilità di implementazione, manutenzione e leggibilità del codice.

- Il progetto è suddiviso in eseguibile “client.c”, eseguibile “server.c” e un file eseguibile in comune “common.c” in cui sono inserite le funzioni comuni per evitare così codice duplicato, in quanto PUT e GET sono operazioni speculari che differiscono per sender e receiver. Per cui i metodi sendFile() e receiveFile() che implementano queste operazioni sono stati inseriti in “common.c” e possono essere utilizzati sia dal client che dal server.

- Ho cercato di alleggerire il più possibile il main di client e server delegando l’invio e la ricezione a funzioni esterne, mantenendo solo operazioni di setup della socket.

- Per la simulazione di perdita dei pacchetti, ho deciso di saltare l’invio di un pacchetto in modo casuale con una probabilità scelta dall’utente.
  Per la scelta casuale del pacchetto che non deve essere inviato viene chiamata la funzione lossPkt() che prende come parametro la probabilità scelta dall’utente e restituisce true o false se quel pacchetto deve essere inviato o no.

SCELTE IMPLEMENTATIVE

- Il pacchetto di richiesta e di risposta è unico e realizzato con due campi:

```
typedef struct pkt{
   PKT_HEADER header_pkt;
   char payload[MAX_PACKET_SIZE+1];
}PKT;
```

con header_pkt di tipo:

```
typedef struct header_pkt{
   int seqNum; //offset of first byte
   int size;   // size of payload
   bool isLastPacket; // 1 --> yes
   int seqWindow;
   char command[20];
   int winSize;
   int probability;
   int timeout;
}PKT_HEADER;
```

Il pacchetto di ack contiene solo due campi che indicano il numero di sequenza di riferimento al pacchetto che si è ricevuto e il campo booleano per l'operazione richiesta:

```
typedef struct ack{
   int seq_num;
   bool confirmOperation;
}ACK;
```

Le funzioni createPkt() e createAck() in common.c si occupano di creare pacchetti settando i campi necessari.

- Ho implementato un protocollo StopAndWait per inviare i pacchetti da client a server contenenti il comando da eseguire scelto dall'utente e i parametri di ingresso comuni :
    - dimesione finestra
    - probabilità di perdita dei pacchetti(intero tra 0 e 100)
    - timeout

  Dopo l'invio del comando il client rimane in attesa di arrivo dell'ack di conferma dell'operazione e solo in caso "true" procede all'invio o alla ricezione di pacchetti di dati.

- Il comando LIST viene implementato dalla funzione getFileList() in server.c che torna la lista di tutti i file presenti su server.

```
sprintf(list_command, "ls ServerFile -p | grep -v /")
pipe = popen(list_command, "r")
```

Tramite chiamata popen(..) si ottiene uno stream di dati su pipe : la popen fa eseguire su shell il comando espresso in list_command.
I dati ottenuti vengono copiati su un'area di memoria allocata dinamicamente individuata da file_list di dimensione stabilita inizialmente.
Se la dimensione dell'area di memoria non è adeguata per memorizzare tutta la lista di file, tramire realloc(..) viene raddoppiata, mantenendo stesso indrizzo di memoria e i dati già inseriti inalterati.

- I comandi GET e PUT sono implementati tramite due funzioni comuni a client e server: sendFile() e receiveFile() in common.s .
  Questo perché i comandi sono speculari: ad esempio nel caso di GET richiesta dall'utente il client chiama la receiveFile() e il server la sendFile(); viceversa nel caso di PUT.

    - SendFile() si occupa di aprire il file richiesto tramite filename, di dividerlo in chunck di dimensione MAX_CHUNK_SIZE dove ogni chunk viene suddiviso in ulteriori pacchetti di dimensione data dal MAX_DATA_SIZE, e infine tramite Selective Repeat si occupa di inviarlo al receiver.
      Per i pacchetti che non hanno ack riscontrato, o per i quali scade il timeout, la funzione si occupa del loro rinvio al receiver.
      La comunicazione da parte del sender termina quando tutti i pacchetti hanno un corrispettivo ack che segnala la loro ricezione.
      L'ascolto di ack in arrivo viene eseguito da una funzione esterna tramite thread.

    - ReceiveFile() si occupa di ricevere pacchetti di dati, di estrarre il payload e copiarne il contenuto sul file aperto in base al path che prende come parametro di ingresso, ed infine di mandare ack con numero di sequenza relativo al pacchetto arrivato.

- Il protocollo Selective Repeat è stato implementato sul lato sender utilizzando tre array di dimensione winLen, parametro passato in ingresso da terminale.

```
ackList = malloc(winLen*sizeof(bool));
pktSentList = malloc(winLen*sizeof(bool));
pktTimeList = malloc(winLen*sizeof(time_t));
for(int i=0; i<winLen; i++){
   ackList[i] = false;
   pktSentList[i] = false;
}
```

L'ackList è un array di booleani per indicare gli ack ricevuti (true).
Il pktSentList indica quali pacchetti sono stati inviati.
Il pktTimeList è un array con i timer relativi ai singoli pacchetti.
AckList e pktSentList vengono settati inizialmente a false.

Sul lato receiver la finestra windowRecv è realizzata come un array di dimensione winLen di booleani che indicano quali pacchetti sono stati ricevuti; anche questa settata inizialmente a false.

Dal lato sender quando un pacchetto *i* viene inviato, si avvia il timer relativo in poszione *i.* Se il pacchetto riceve l'ack prima dello scadere del timer, si imposta a true acklist[i]:

- Se *i* è indice del primo elemento dell'array di ack, e i successivi ack sono già impostati a true, si fa scorrere la finestra pktSentList fino al primo elemento con ack impostato a false.

- Se *i* è indice del primo elemento dell'array di ack, ma i successivi elementi sono ancora impostati a false, la finestra viene fatta scorrere di una sola unità.

Di conseguenza si fanno scorrere dello stesso shift anche i due array relativi ad ack e timer. Se invece il pacchetto *i* non riceve ack perché andato perduto o il timer supera il valore di timeout allora si procede al rinvio del pacchetto.

Dal lato receiver windowRecv la si fa scorrere quando ad arrivare è il pacchetto con numero di sequenza pari all'indice del primo elemento della finestra, con le stesse modalità del lato sender.

- La ricezione degli ack è affidata ad un thread che esegue la listenAck(), rimanendo in ascolto per intercettarli.
  Per l'implementazione ho utilizzato un pthread che fa la recvfrom(..) attendendo pacchetti di tipo ack. Se riceve tali pacchetti, modifica l'array ackList settando a true l'elemento che ha indice pari al numero di sequenza dell'ack.
  Ho utilizzato i mutex lock per lockare zone di codice dove si andava a modificare l'array stesso.

- Per lo scadere del timeout ho utilizzato la funzione time() che conta i secondi a partire dal 01/01/1970. Prendendo il time() all'invio del pacchetto e al successivo controllo, se la differenza tra i due tempi è maggiore del timeout, significa che il tempo è scaduto e occorre reinviare il pacchetto.

LIMITAZIONI

Il sistema funziona al variare di winLen e timeout con qualsiasi valore omettendo la probabilità di perdita, ma inserendo la probabilità di perdita ho riscontrato i problemi sotto elencati:

- Testando il sistema con file di varie dimensioni, nel caso della PUT e grandi file il server non riesce a concludere l'esecuzione.
  Ad esempio PUT con file "cenerentola" nel clientFile.

- Altra limitazione riscontrata nel testare sia la GET che la PUT, se la probabilità supera il 35% la receiverFile() non funziona.

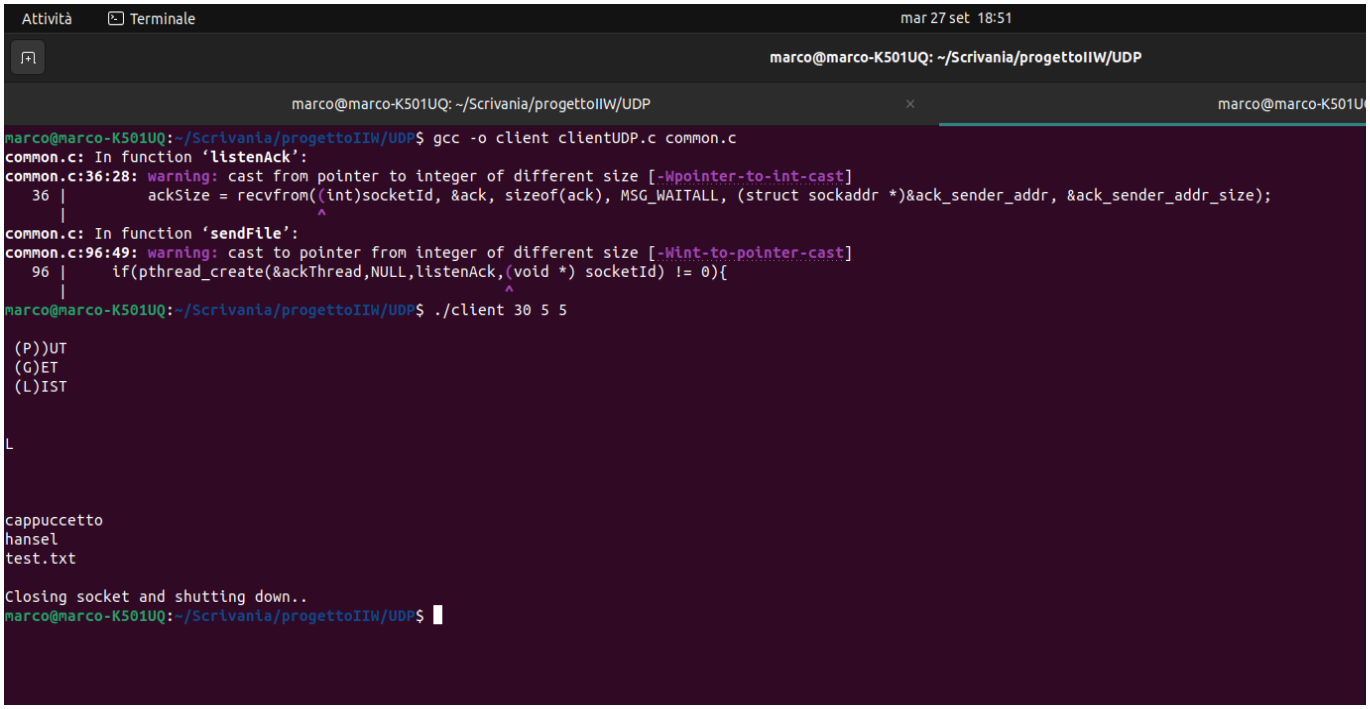
- Se uso una finestra più piccola di 5, la PUT funziona ma la GET no: senza perdita funziona.

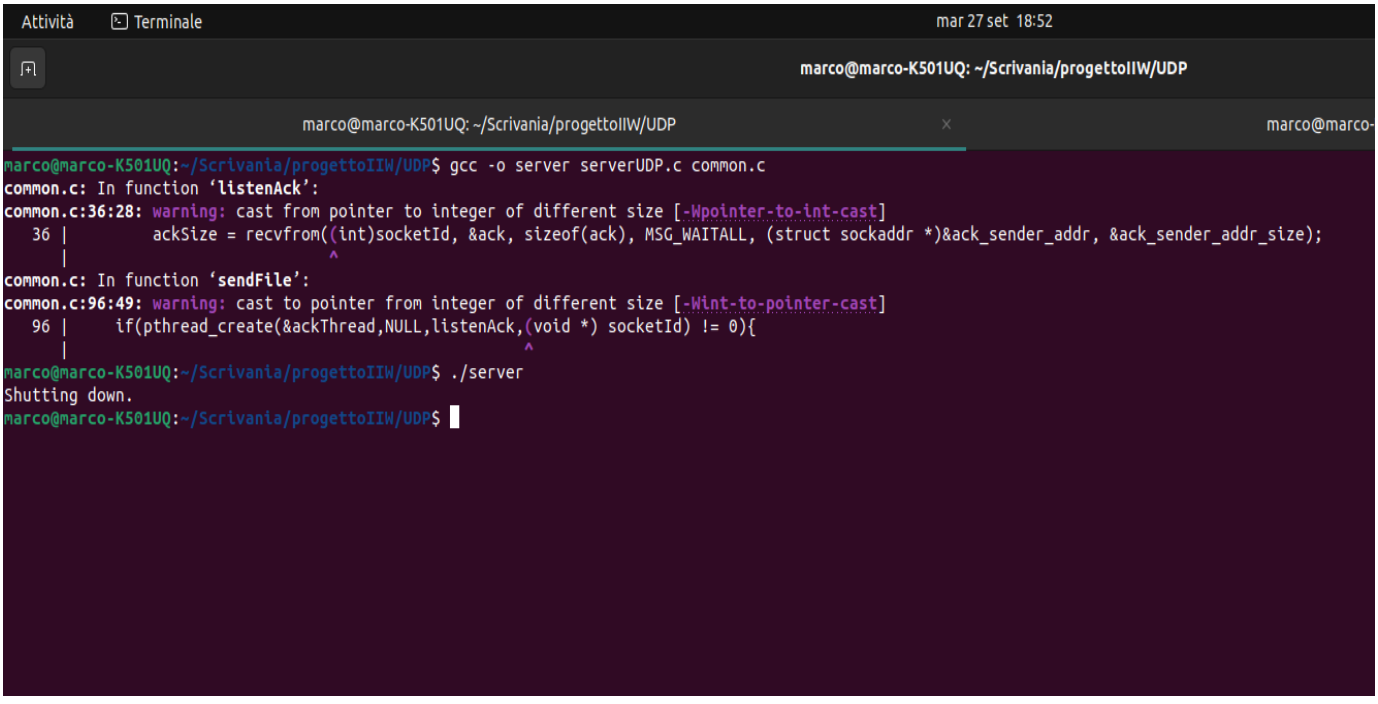
PIATTAFORMA UTILIZZATA

Il progetto è stato implementato tramite Clion 2022.2, compilato ed edeguito tramite gcc da terminale.

ESEMPIO DI FUNZIONAMENTO

- LIST





- GET

```
Closing socket and shutting down..
marco@marco-K501UQ:~/Scrivania/progettoIIW/UDP$ ./client 30 5 5

 (P))UT
 (G)ET
 (L)IST


G



Please insert file name: cappuccetto
Ack 0 sent.
Ack 1 sent.
Ack 2 sent.
Ack 4 sent.
Ack 5 sent.
Ack 6 sent.
Ack 7 sent.
Ack 3 sent.
Closing socket and shutting down..
marco@marco-K501UQ:~/Scrivania/progettoIIW/UDP$
```

```
Shutting down.
marco@marco-K501UQ:~/Scrivania/progettoIIW/UDP$ ./server
Resending pkt: 0
packet 0 sent
Resending pkt: 1
packet 1 sent
Thread: received ack number 0
Resending pkt: 2
packet 2 sent
Resending pkt: 3
packet 3 sent
Resending pkt: 4
packet 4 sent
Resending pkt: 4
Thread: received ack number 1
packet 5 sent
Resending pkt: 4
Thread: received ack number 2
packet 6 sent
Resending pkt: 4
Packet 7 is last packet
Thread: received ack number 4
packet 7 sent
Thread: received ack number 5
Thread: received ack number 6
Thread: received ack number 7
Resending pkt: 0
packet 3 sent
Thread: received ack number 3
Shutting down.
marco@marco-K501UQ:~/Scrivania/progettoIIW/UDP$
```

- PUT

```
Closing socket and shutting down..
marco@marco-K501UQ:~/Scrivania/progettoIIW/UDP$ ./client 30 5 5

 (P))UT
 (G)ET
 (L)IST


P



Please insert file name: sirenetta
Resending pkt: 0
packet 0 sent
Resending pkt: 1
packet 1 sent
Thread: received ack number 0
Resending pkt: 2
packet 2 sent
Resending pkt: 3
packet 3 sent
Resending pkt: 4
packet 4 sent
Resending pkt: 4
Thread: received ack number 1
packet 5 sent
Resending pkt: 4
Packet 6 is last packet
Thread: received ack number 2
packet 6 sent
Thread: received ack number 4
Thread: received ack number 5
Thread: received ack number 6
Resending pkt: 0
packet 3 sent
Thread: received ack number 3
Closing socket and shutting down..
marco@marco-K501UQ:~/Scrivania/progettoIIW/UDP$
```

```
marco@marco-K501UQ:~/Scrivania/progettoIIW/UDP$ ./server

Ack 0 sent.
Ack 1 sent.
Ack 2 sent.
Ack 4 sent.
Ack 5 sent.
Ack 6 sent.
Ack 3 sent.
Shutting down.
marco@marco-K501UQ:~/Scrivania/progettoIIW/UDP$
marco@marco-K501UQ:~/Scrivania/progettoIIW/UDP$
```

MANUALE

Per la compilazione :

gcc -o client clientUDP.c common.c

gcc -o server serverUDP.c common.c

Per eseguire:

./client prob winLen timeout

./server

dove (prob, winLen, timeout) sono tre interi che rappresentano la probabilità di perdita dei pacchetti, la dimensione della finestra e il timeout.

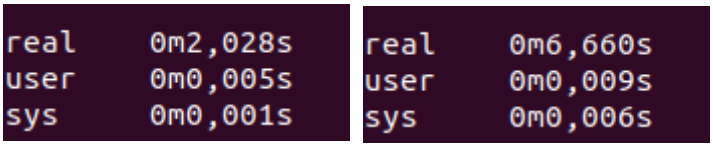
Deve essere eseguito prima il server e poi il client.

PERFORMANCE

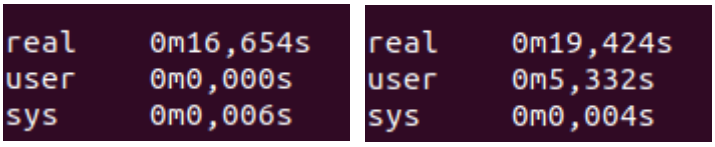
Sono test eseguiti sempre sugli stessi file, cambiando di volta in volta un solo parametro

- PROB = 30%
- WINDOW_LEN = 5
- TIMEOUT = 5 SECONDI

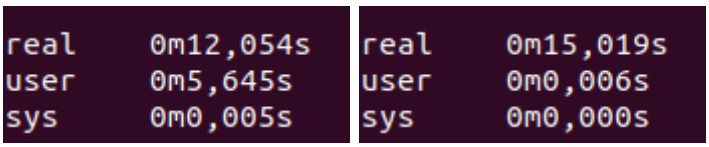
1. LIST CLIENT – LIST SERVER



2. GET CLIENT – GET SERVER

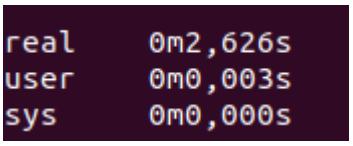


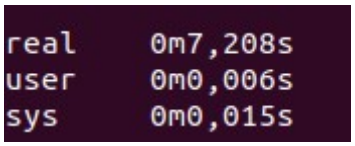
3. PUT CLIENT – PUT SERVER



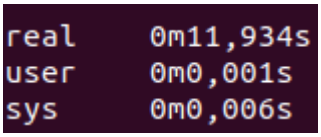
- PROB = 30%
- WINDOW_LEN = 6
- TIMEOUT = 5 SECONDI

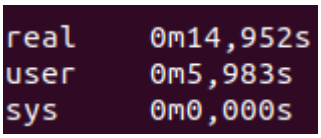
1. LIST CLIENT – LIST SERVER



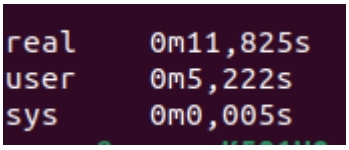


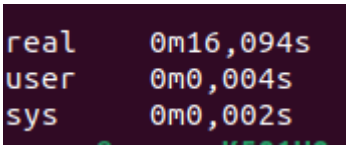
2. GET CLIENT – GET SERVER





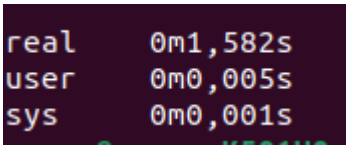
3. PUT CLIENT – PUT SERVER

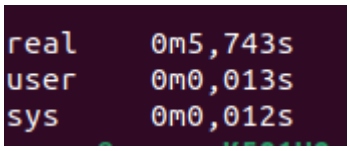




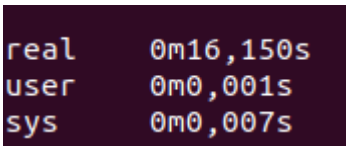
- PROB = 35%
- WINDOW_LEN = 5
- TIMEOUT = 5 SECONDI

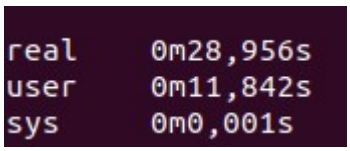
1. LIST CLIENT – LIST SERVER



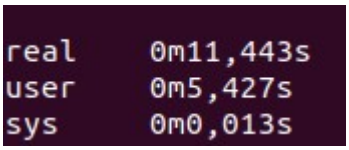


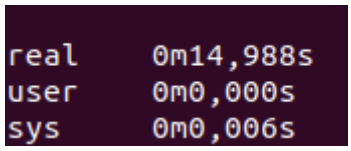
2. GET CLIENT – GET SERVER





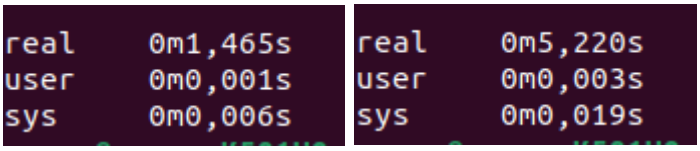
3. PUT CLIENT – PUT SERVER



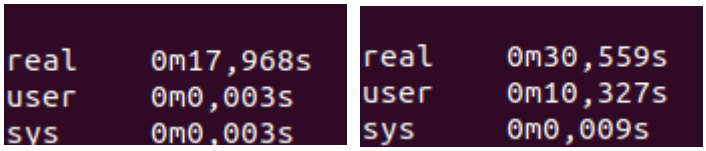


- PROB = 30%
- WINDOW_LEN = 5
- TIMEOUT = 10 SECONDI

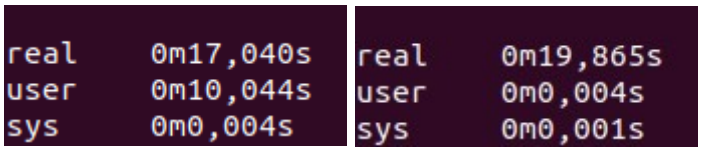
1. LIST CLIENT – LIST SERVER



2. GET CLIENT – GET SERVER



3. PUT CLIENT – PUT SERVER



Si può vedere come all’aumentare del timeout o della probabilità di perdita i tempi di GET e PUT aumentano, dovuti al fatto che ci sono pacchetti da dover inviare nuovamente perché non hanno ricevuto ack e il timeout è scaduto. Per cui il tempo di esecuzione dipende principalmente dal tempo di timeout scelto, essendo il tempo che bisogna attendere prima di poter procedere al rinvio dei pacchetti, oltre al numero di pacchetti he devono essere rinviati.

Mentre aumentando la finestra di invio, anche se di poco, il tempo impiegato nell’esecuzione di GET diminuisce.