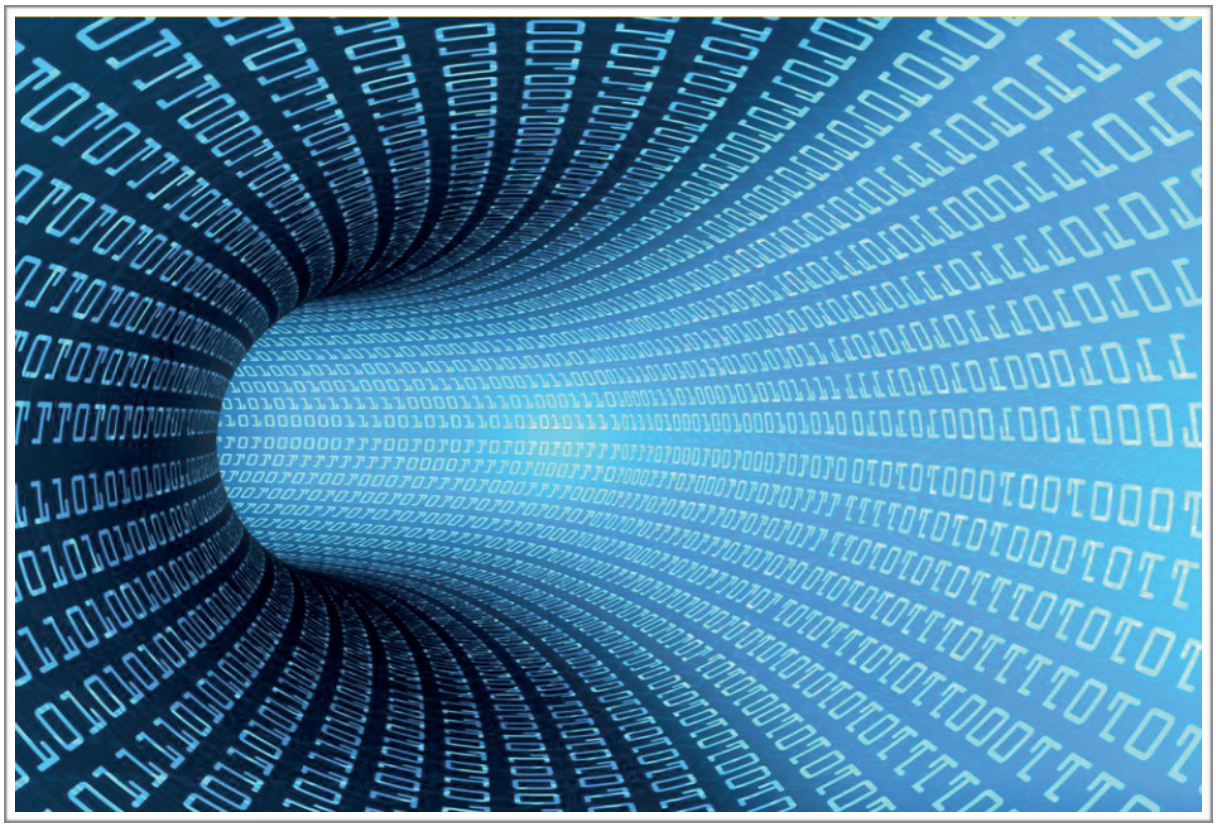
Relazione di Ingegneria di Internet e Web

Trasferimento file su UDP - TCP 4.5



Federica Villani

Alessandro Montenegro

Davide Sfameli

Settembre 2020

Università degli Studi di Roma Tor Vergata

Indice

Introduzione

Si presenta il lavoro di progetto svolto a termine del corso di Ingegneria di Internet e Web, corso di Ingegneria Informatica, Laurea Triennale.

Il seguente elaborato è frutto della collaborazione con Sfameli Davide, distaccatosi dal gruppo nel mese di Luglio 2020 in cui ha presentato il lavoro svolto fino a quel momento.

La richiesta che veniva fatta era di progettare e implementare, in linguaggio C, sfruttando l'API del socket di Berkeley, un'applicazione in architettura
client-server, che permetta il trasferimento file affidabile impiegando il servizio di trasporto UDP senza connessione, tramite implementazione del protocollo ARQ di TCP.

Si richiedeva che il software permettesse connessione client-server senza autenticazione, di mettere a disposizione del client un comando 'LIST' per visualizzare i file presenti nella memoria del server, un comando 'GET' per richiedere l'invio dal server di un file presente nella sua memoria e il comando 'PUT' che permettesse al client di caricare nella memoria del server un file dalla propria memoria.

Lo sviluppo del software, per garantire i requisiti, è stato preceduto da una fase di progettazione che ci ha spinti a creare varie librerie di funzioni che potessero garantire proprietà di modularità e leggibilità del codice, ma soprattutto, considerata la dimensione estesa delle logiche implementate, una maggior facilità nell'individuare errori e debolezze del codice.

Tutta la produzione del nostro codice è avvenuta sfruttando la piattaforma VisualStudioCode, che, oltre a offrire un ambiente di sviluppo moderno e funzionale, presenta notevoli funzionalità di connettività con il servizio di repository online di GitHub. E' stato infatti fondamentale, soprattutto in questo
periodo, poter fare affidamento su una piattaforma che ci permettesse di lavorare in parallelo su componenti diverse e a fine giornata di ottenere un merge dei due lati che fosse semplice, veloce e soprattutto consistente.

Tutta la fase di compilazione, esecuzione e debbugging è avvenuta mediante tool da riga di comando, in particolare, lavorando da una macchina MacOS la shell utilizzata è stata la classica zsh offerta dai sistemi OSX.

Si procederà mostrando lo scopo e le scelte progettuali fatte per ciascuna libreria di funzioni.

Struttura dei pacchetti

Libreria PacketStructure.h

La libreria PacketStructure.h è stata pensata per offrire a ogni componente del software un accesso il meno invasivo possibile, in termini di leggibilità, a tutte le funzioni di costruzione, modifica e lettura delle componenti delle strutture di 'Header' e 'Packet', costituenti la '*protocol data unit*' del nostro TCP 4.5.

Come appare evidente, la struttura del nostro segmento è caratterizzata da un'intestazione che estende le informazioni già trasportate da quella del vero protocollo di trasporto su cui risiede la nostra applicazione, e sufficienti a implementare la logica di richiesta di ritrasmissione automatica distintiva di TCP.

In tale intestazione si trasporterà quindi il numero di sequenza caratterizzante il payload del segmento e il numero di sequenza che, in veste di riceventi, si desidera riscontrare, offrendo il servizio di '*piggybacking*' degli ack tipico di TCP.

```
typedef struct Headers{
    uint32_t seqNumber;         // Sequence number of the packet
    uint32_t ackNumber;         // Ack number
    uint16_t receiverWindow;    // Receiver Window size
    int      dataDimension;     // Dimensione dei dati inviati
    unsigned int ack :1;           // Flag for ACK
    unsigned int syn :1;           // Flag to start a communication
    unsigned int fin :1;           // Flag to end a communication
} Header;


// Packet: 1472 Byte
typedef struct Packets{
    Header hdr;
    char   d[MSS];
} Packet;
```

Vengono inseriti un campo per la comunicazione all'altro estremo della connessione, dello spazio rimanente nel proprio buffer di ricezione, per garantire il controllo di flusso, altro servizio fondamentale per ogni corretta implementazione di TCP.

Si inserisce un campo per comunicare all'altro lato della connessione la quantità di dati che tale segmento sta trasportando, e, infine, si sono aggiunte tre variabili di flag per discriminare il significato di quello specifico segmento all'interno della connessione.
Tali ulteriori Flag, come si può notare in figura, permettono di capire se il pacchetto che si sta processando contiene un ACK, se richiede la fine della connessione tramite il bit di FIN o se richiede l'instaurazione della connessione, grazie al bit di SYN.
L'intera struttura del pacchetto dati è pensata per replicare al meglio il protocollo a livello di trasporto TCP, in tutti i suoi aspetti.

Lista di Pacchetti

Libreria PacketList.h

Il protocollo di comunicazione affidabile di TCP, come studiato, lascia libera la scelta sul destino dei pacchetti ricevuti fuori ordine.
La scelta di più facile implementazione è chiaramente quella di scartarli rispondendo all'evento di una loro ricezione con l'ack dell'ultimo byte correttamente ricevuto.
Per garantire una maggior efficienza del nostro software si è deciso di mantenere, fino a un certo valore limite pari proprio alla dimensione della finestra di ricezione, una lista ordinata di pacchetti ricevuti fuori ordine, in modo da evitare di 'sprecare' dei dati arrivati correttamente.

La libreria PacketList.h mantiene la definizione della struttura dati che implementa tale lista e la funzione che permette di inserire un pacchetto arrivato fuori ordine in questa, mantenendoli ordinati per numero di sequenza. Disponendo il contenuto della struttura in maniera ordinata, si garantisce che il tempo di accesso a questa, per la consegna di dati dopo la ricezione di un ACK nuovo, sia minimo.

AVL delle Connessioni

Libreria AvlConnection.h

La principale mancanza che la socket UDP mostra alle entità di livello applicativo che ne sfruttano i 'servizi', è la tecnica di demultiplazione adottata.
Il socket UDP non offre cognizione di connessione alle applicazioni, trasferendo tutto il traffico ricevuto da molteplici entità sparse e distinte nella rete all'unica applicazione locale da cui viene utilizzata.
Sebbene ciò generi, nel lato client dell'applicazione, una banale necessità di controllare la sorgente del segmento, costituisce, invero, un elemento di notevole complessità nel lato server, il quale dovrà valutare la sorgente del segnale e operare una funzione di ricerca, tra le varie 'connessioni' attive, di quella a cui effettivamente quei dati erano destinati.
La nostra scelta progettuale è stata guidata da studi precedenti nell'ambito degli algoritmi e delle strutture dati ottimali per tale scopo.
Il server mantiene, infatti, una struttura AVL, ovvero un albero binario di ricerca che mantiene i nodi ordinati tramite un ordinamento variabile e impostato in fase di progettazione.
In particolare, dovendo demultiplare traffico sulla base di indirizzo IP sorgente e contemporaneamente del numero di porta sorgente, la scelta è stata di distinguere prima l'indirizzo IP secondo l'ordinamento numerico e solo in caso di uguaglianza di passare mediante lo stesso ordinamento a valutare i numeri di porta.
La libreria AvlConnection.h mantiene tutte le funzioni e le strutture necessarie a un corretto funzionamento e un facile utilizzo di questa

struttura che, come introdotto, viene generata e utilizzata solo dal lato server del nostro codice.
Si fa notare che l'accesso alla struttura AVL viene fatto solo in caso di ricezione di un segmento sul socket UDP o nel caso di rimozione o inserimento di un nuovo nodo.
Grazie alla scelta di questa specifica struttura dati si garantisce un tempo di accesso logaritmico nel numero di connessioni, numero che può essere impostato dal server tramite la funzione '*listenTCP*' che impone il massimo numero di connessioni attive pendenti all'interno dell'albero.

Parametri globali ai Thread

Libreria ThreadStructure.h

La necessità di garantire la comunicazione full-duplex, come offerta dal reale protocollo TCP, ci ha spinti a pensare alla singola connessione come, per lo meno localmente, a un collettivo di tre entità distinte, ciascuna con uno specifico ruolo.
Con lo scopo di non costruire un software eccessivamente gravante sulla C.P.U. del calcolatore su cui lo si esegue, e per la semplicità delle API Posix offerte per la loro gestione, si è scelto di optare per una versione multithreaded piuttosto che per una multi-processo.
I tre thread che caratterizzano, quindi, una connessione nella nostra versione di TCP avranno compiti distinti, ma pur sempre strettamente connessi tra loro. Abbiamo quindi dovuto progettare la forma delle strutture contenenti i parametri che, in fase di generazione di questi, andavano passati ai thread e, in fase di esecuzione condivisi da essi.
La libreria ThreadStructure.h mantiene, quindi, le definizioni di queste strutture, rinominate chiaramente '*tParam*', '*tParam_receive*' e '*tParam_send*'.
Come si può notare le ultime due contengono a loro volta la prima, che costituisce, infatti, il nodo centrale delle informazioni necessarie a offrire tutti i servizi richiesti da una connessione TCP, e che sarà sempre condivisa tra tutte le entità della specifica connessione. Le restanti componenti delle ultime due saranno invece informazioni che, a livello

4.5, saranno di uso esclusivo, rispettivamente, delle entità *tReceive* e *tSend*.
Oltre a queste, si definiscono nella presente libreria altre due strutture dati, una necessaria per il passaggio dei dati richiesti al thread *tHandshaking*, che viene generato in seguito alla ricezione di un segmento di SYN inedito, e che viene rilasciato immediatamente dopo la conclusione dell'handshaking a tre vie come visto su TCP.
Mentre l'altra sarà destinata a trasferire i dati sulla connessione al thread che permette di gestire, in modo concorrente, le varie istanze di protocollo applicativo per i vari client che ne facciano richiesta.
Come si può notare in quest'ultima saranno presenti i puntatori alle locazioni di memoria su cui è possibile scrivere o leggere dati, offrendo al livello applicativo l'astrazione di una comunicazione affidabile, intesa come stream di byte.
Tutti i buffer che vengono inseriti per il mantenimento dei dati vengono implementati e gestiti come buffer circolari, giustificando la presenza di due puntatori a locazioni interne a questi, l'uno per la scrittura e l'altro per la lettura.
Il collegamento tra il livello 5 e il nostro 4.5 saranno quindi i char buffer '*rxWndData*' e '*txWndData*', istanziati rispettivamente nel *tParam_receive* e nel *tParam_send,* e richiamati, mediante puntatori, proprio nella struttura sopra nominata *applicationParams.*
Operando con entità multiple sia in lettura che in scrittura su aree di memoria evidentemente condivise, si è fatta necessaria la sincronizzazione degli accessi a queste, al fine di garantire un utilizzazione concorrente, mantenendone l'integrità e la consistenza.
Ancora una volta ci siamo potuti appoggiare alle API Posix che offrono dei meccanismi di gestione di mutex tra thread in modo rapido, affidabile e semplice.

Probabilità di perdita

Libreria Probability.h

Per poter svolgere realisticamente i test sull'efficacia delle nostre scelte implementative era necessario simulare eventi di perdita di pacchetti, cosa alquanto improbabile in una rete locale o in un test sull'interfaccia di loopback.
La libreria *Probability.h* viene pensata proprio per questo scopo. In essa è possibile modificare la probabilità di perdita di un pacchetto e viene definita e costruita la funzione per il 'lancio della moneta'.
Tale logica viene implementata costruendo un array di 100 entrate di cui esattamente il valore della probabilità di perdita P vengono impostate a 0.
Ovviamente la scelta di P sarà limitata ai valori interi.
La perdita o meno di un pacchetto sarà quindi in dipendenza della scelta randomica dell'entrata dell'array che in una specifica chiamata si desidera considerare.

Timer e RTT

Libreria Timer.h

I protocolli che implementano la comunicazione affidabile su un canale inaffidabile soggetto a perdite mediante riscontri, sono inevitabilmente legati alla gestione delle tempistiche della comunicazione, proprio in quanto, così come i pacchetti, anche i riscontri possono andare perduti nel canale (a maggior ragione che facendo piggybacking trasporteranno anch'essi payload).
In TCP, grazie anche al meccanismo degli ACK cumulativi è possibile gestire le ritrasmissioni con un solo timer per connessione.
Abbiamo inserito questa componente mediante chiamate alla funzione *clock*() presente nella libreria *time.h.*, la quale restituisce il ciclo di clock in cui ci si trova quando la si chiama, l'era della funzione *clock*() inizia, con un valore 0, quando il programma inizia l'esecuzione.

La libreria *Timer.h* include tale funzione e in essa vengono definite le funzioni per la gestione e l'update dei timer adattativo, mediante le tecniche spiegate nella RFC di TCP.
In particolare, si utilizzano due medie mobili esponenziali ponderate (EWMA), anche considerabili come filtri passa-basso, che garantiscono di mantenere un valore del timer adattivo alle condizioni della rete e quanto più possibile simile al valore di RTT, che sia poco suscettibile a forti variazioni di quest'ultimo.

I valori di *alpha* e di *beta* usati per pesare effettivamente le medie, sono gli stessi presentati a lezione, in modo tale da rendere minimo il tempo in cui il processore effettua le moltiplicazioni e divisioni.

Gestione dei File

Libreria FileManaging.h

La natura della richiesta stessa alla base del progetto impone un ampio utilizzo dei file, sia in lettura che in scrittura.
La libreria *FileManaging.h* include e offre tutte le funzioni necessarie al loro utilizzo.
In particolare, contiene delle funzioni per la gestione dei file di testo mantenenti le liste di file presenti nelle memorie di ciascun client o server.
La scelta fatta è stata di mantenere i "file lista" ordinati, in modo che per la ricerca di un nome al loro interno si potesse procedere con una ricerca binaria piuttosto che con una sequenziale, riducendo la complessità dell'operazione esponenzialmente nel numero di file contenuti in memoria.
Inoltre, non è necessario per un utilizzatore dell'applicazione andare a modificare direttamente i "file lista". Infatti, essi verranno creati dinamicamente allo start-up del programma, sulla base dei file effettivamente in possesso del client o del server, e varranno mantenuti aggiornati sulla base delle GET e delle PUT effettuate in sede di esecuzione.

Tale libreria contiene anche le due funzioni, fondamentali per il livello applicativo, pensate per la lettura dal buffer di trasmissione e la riscrittura su file, e per la lettura da file e la riscrittura sul buffer di trasmissione: la *WriteOnBuff* e la *ReadFromBuff*.
In realtà, discriminando tramite i parametri in chiamata, tali due funzioni offrono anche la possibilità di differenziare le componenti lette

tra la riga di intestazione del messaggio applicativo e il contenuto effettivo.
A entrambe queste funzioni vengono passate le strutture del livello applicativo definite in *ThreadStructure.h* per poter correttamente modificare i parametri sulle finestre di trasmissione e di ricezione, gestire tali buffer come, appunto, circolari e per notificare alle entità di gestione della connessione TCP 4.5 la quantità di byte che sono stati scritti in una, piuttosto che letti, e quindi sovrascrivibili, nell'altra.
Tali due funzioni sono bloccanti e richiedono, a priori, la quantità di byte da leggere o scrivere.

Comunicazione Affidabile

Libreria ReliableCommunication.h

La libreria *ReliableCommunication.h* costituisce la vera anima del progetto, in essa sono contenute tutte le funzioni e le logiche dei thread di gestione del protocollo ARQ di TCP, per la comunicazione affidabile, controllo di flusso e controllo di congestione.

La prima funzione che viene definita è la *deliverData* pensata per essere invocata dal thread *tReceive* che si occupa di gestire i pacchetti ricevuti. Tale funzione ha lo scopo di consegnare il payload dei segmenti TCP 4.5 ricevuti e che, dopo la demultiplazione, sono passati a *tReceive*, all'interno del buffer di ricezione, dove saranno letti dal protocollo applicativo attraverso l'invocazione della già precedentemente citata *ReadFromBuff*. Vengono quindi lasciati a questa i compiti di notificare al livello applicativo l'arrivo di nuovi dati da leggere, aggiornando la variabile *byteToRead* e di gestire la scrittura sul buffer circolare di ricezione.

La seconda funzione che viene definita è la *congestion_control*, ideata per implementare l'automa che, come studiato, implementa il servizio di controllo di congestione, fondamentale per la rete tanto quanto per la garanzia di fairness che offre ai processi applicativi. Il protocollo viene implementato nella sua versione integrale, comprensiva anche dello stato di *Fast Recovery*.

Come era stato possibile apprezzare attraverso la teoria, abbiamo potuto ottenere una logica che produce questi risultati basandosi su un livello di rete senza notifica esplicita di congestione, solo attraverso l'overhead che i segmenti TCP hanno già per garantire il corretto funzionamento della comunicazione affidabile.

La funzione *congestion_control* viene invocata anch'essa dal thread *tReceive,* il quale gestendo i pacchetti in ingresso, è il primo deputato a impostare e passare a questa, i flag sulle osservazioni indirette dello stato di congestione della rete.

Ciò viene ovviamente fatto per ogni segmento ricevuto.

L'unica modifica che presenta la nostra funzione per il controllo di congestione, rispetto all'automa presentato a lezione, consiste nel modo di calcolare il valore della *congestion window* in fase di "Congestion Avoidance". Infatti, invece di aumentare di un *MSS* tale finestra ogni RTT, si aumenta di *MSS*(MSS/cwnd)* ogni volta che si è in presenza di un nuovo ACK. Tale approssimazione è stata ripresa da una rappresentazione dell'automa del controllo di congestione presente a pagina 257 di "*Reti di Calcolatori e Internet - Kurose e Ross*".

Così facendo, si riescono a sfruttare meglio i dati già presenti nei thread di gestione della comunicazione affidabile, senza il bisogno di rendere la funzione *congestion_control* un altro thread atto anche a misurare intervalli di tempo.

La terza e ultima funzione che viene definita è la *sendSimple,* questa è l'ultimo nodo di livello 4.5 che viene attraversato da un pacchetto in uscita. Invocata dal thread *tManage,* le vengono passati i pacchetti comprensivi della componente di payload e di quella di ACK, o di una sola delle due. La *sendSimple* procederà quindi alla gestione del timer per le ritrasmissioni in attivazione e all'implementazione della probabilità di

perdita attraverso la chiamata *isLoss*(), già precedentemente discussa in *Probability.h*.
Una volta superato lo scoglio della probabilità di perdita, viene sfruttata l'API *sendto* per l'effettiva scrittura dei dati in uscita sul socket UDP.

Di seguito, sono riportate le descrizioni accurate dei thread cheti occupano di gestire, collaborando, la comunicazione affidabile di TCP.

tReceiveTCP
Il thread *tReceive* viene investito del compito di elaborare i pacchetti in ingresso a una specifica connessione. Ne viene generato uno per ogni connessione attraverso le chiamate *acceptTCP* da un lato e *connectTCP* dall'altro.

Questo è diviso in due fasi principali, la prima di gestione dell'handshake, permette di discriminare, sulla base del tipo di 'primo pacchetto' ricevuto il comportamento da mantenere. Il server riceverà un segmento di SYN e risponderà con un SYNACK, riscontrando il SYN ricevuto e generando un numero di sequenza random per la nuova connessione accettata, mentre il client riceverà come 'primo pacchetto' un SYNACK, del quale dovrà controllare il valore di riscontro e al quale dovrà rispondere con un ultimo ACK per riscontrarne la corretta ricezione.

La seconda fase è un ciclo while infinito, dal quale il thread esce solo in caso di chiusura della connessione o rottura di qualche componente non recuperabile.
In questo il thread compierà tutte le operazioni tipiche che il lato ricevente di una connessione TCP ha il dovere di svolgere.

Innanzitutto acquisirà i pacchetti che la componente di demultiplazione ha scaricato nel nostro socket TCP 4.5, ovvero nella struttura *ConnData* dei nodi dell'AVL delle connessioni pendenti.
A ogni iterazione tutti i pacchetti ricevuti verranno raccolti dalla 'buca delle lettere', copiati in un buffer d'appoggio e processati in un ciclo for annidato.

Per ciascuno di questi si valuterà sia la componente payload, se presente, che quella ACK, anch'essa se presente.

Qualora nel pacchetto dell'iterazione sia presente un payload, viene valutato il numero di sequenza: se in ordine, i dati vengono consegnati al buffer di ricezione mediante la *deliverData,* altrimenti, come anticipato, vengono sfruttate le logiche definite in *PacketList.h* per bufferizzare il pacchetto arrivato fuori ordine nella lista di pacchetti. Questi saranno controllati alla ricezione di un nuovo segmento in ordine, per valutare se possano essere consegnati anch'essi o meno. Il mantenimento della lista ordinata permette di ridurre drasticamente i tempi di accesso a questa in fase di recupero dei pacchetti.
Al termine di questa fase, per ogni pacchetto dati ricevuto si costruisce un pacchetto di riscontro dell'ultimo numero di sequenza correttamente ricevuto in ordine.
Esso verrà inserito all'interno del buffer circolare *receiving_buff,* dal quale sarà prelevato e gestito dal thread *tManage*.

Viene poi controllato se nel pacchetto dell'iterazione era stato impostato il bit flag di ACK. Si opererà quindi un primo controllo veicolato dalla variabile *readAckHandshake* che discrimina l'ultimo ack dell'handshake per una gestione differenziata, subito dopo si analizzano gli 'ACK numbers' per capire se si tratti di un ACK duplicato o di uno nuovo. Si

fa, infine, scorrere la finestra di ricezione di conseguenza. Si mantengono le variabili con il numero di ACK duplicati per indurre, eventualmente, la ritrasmissione rapida, alla ricezione del terzo ACK duplicato.

tSend

Il thread *tSend,* costituisce l'altra faccia della medaglia.
Anch'esso avrà una prima fase in cui gestirà l'handshake esclusivamente lato client, inviando il segmento di SYN e mettendosi in attesa, in modo molto analogo a quanto è stato discusso per la componente di SYNACK del server.
Nella sua seconda fase, anche il *tSend* entrerà in un ciclo while infinito durante il quale gestirà le letture dal buffer di trasmissione, popolato dalla funzione *WriteOnBuff,* il mantenimento del pacchetto a cui viene associato il timer di ritrasmissione e lo scorrimento di questo, la costruzione dei segmenti TCP 4.5 contenenti solo la componente lato trasmittente, ovvero il payload e il sequence number e, infine, la gestione della scadenza del timer, al termine del quale si notifica la presenza di un pacchetto con priorità da ritrasmettere e si inserisce il possessore del timer scaduto nella suddetta locazione.

Una volta completata la costruzione dei pacchetti, questi vengono inseriti nel buffer di comunicazione con il thread *tManage*, nominato *transmission_buff*, anch'esso gestito come circolare.

tManage

Ultimo dei tre thread di gestione della connessione del nostro TCP 4.5, il *tManage* entra in gioco quando i primi due hanno inserito qualcosa nei buffer circolari di comunicazione con il presente. Il thread *tManage* si occuperà quindi di leggere inizialmente dal buffer *receive_buff*, per

valutare la presenza di eventuali segmenti di ACK, costruiti da *tReceive*, da inviare. Se presenti, proverà a vedere se esistono segmenti dati su cui caricarli in piggybacking, altrimenti discriminerà cosa fare sulla base della loro quantità, implementando la logica di ACK ritardati e gestendo, quindi, un ulteriore timer impostato come precisamente descritto per TCP a 500ms. Qualora invece dovesse gestire un secondo ACK da inviare, resetterebbe questo timer inviando subito il secondo ricevuto.

Nel caso in cui non siano presenti ACK da trasmettere, il *tManage* procederà a elaborare i pacchetti di soli dati costruiti e inseriti dal *tSend* all'interno del *transmission_buff*, o ancora prima nella locazione *pkt_with_priority*, dove come accennato vengono depositati i segmenti prioritari per la ritrasmissione.

Ulteriori
Si fa notare che per limitare l'uso della C.P.U. a tali thread, che operano in cicli infiniti, quando non utili, si usano dei semafori che vengono sbloccati per il *tReceive* dal demultiplatore, per il tSend dalla *WriteOnBuff* e per il *tManage*
dai thread sottostanti *tReceive* e *tSend*, verranno bloccati dagli stessi alla terminazione dello svolgimento del proprio compito.

Si definisce un quarto thread comune, sia al lato client che a quello server, per la gestione dell'handshaking e ideato primariamente con lo scopo di mantenere un ulteriore timer per garantire un tempo limite per l'handshake e prevenire attacchi di syn flooding.

Infine, come già precedentemente presentato, viene qui definito il thread esclusivo del client che svolgerà il compito di 'demultiplare' il traffico in

ingresso al socket UDP del client, limitandosi a controllare che effettivamente provenga dal server.

Divisione logica tra livello 5 e 4.5

Libreria ReliableDataTransfer.h

La libreria *ReliableDataTransfer.h* definisce il vero punto di contatto tra il livello applicativo e il nostro livello 4.5. In essa sono, infatti, definite le due funzioni distintive dei socket TCP tradizionali: la *accept* e la *connect*.
Chiaramente queste funzioni baseranno il loro funzionamento sulle logiche dei livelli inferiori discusse nei paragrafi precedenti.

La funzione *acceptTCP* viene invocata quando il server riceve un segmento di SYN da un client, verrà controllato se tale segmento è già presente all'interno dell'AVL delle connessioni e in caso contrario lo si inserisce allocando quanto necessario.
Si proseguirà generando tutte le entità che, come visto, sono necessarie all'implementazione del nostro protocollo di trasporto affidabile, e allocando le strutture condivise tra queste, inizializzandole in maniera opportuna.

La funzione *connectTCP* verrà, invece, invocata dal client, nel momento il cui questo desidera iniziare una connessione con il lato server.
Analogamente alla *acceptTCP* anche la connect avrà il compito di generare le varie entità della connessione e di allocare tutte le strutture dati necessarie al corretto funzionamento.
Tale funzione si metterà, infine, in attesa della conclusione del thread *tHandshaking*, per poter riportare il program counter nel main e

consentire a questo di spostarsi sull'esecuzione delle logiche del protocollo di livello cinque.

Per fare un confronto con il vero TCP, potremmo dire che, per quanto riguarda il lato server dell'applicazione, il 'listening socket' sarà il socket UDP stesso, mentre i 'connection socket' generati a seguito di ogni accept per ogni specifica connessione possono essere riconosciuti nei nodi dell'AVL.
D'altra parte per il lato client, come per il classico TCP, non esiste una tale distinzione, il socket TCP 4.5 completamente dedicato alla singola connessione client-server può essere identificato nell'unica struttura nodo che viene allocata,
proprio come se l'albero AVL, ovvero, nella nostra analogia, il listening socket, fosse costituito da un unico elemento.

Infine, è definita anche la funzione *closeTCP* che, come si può evincere dal nome, si occupa di chiudere la connessione TCP 4.5.
Essa viene invocata quando un client esprime direttamente la volontà di chiudere la sessione, inviando un pacchetto contenente il flag di FIN asserito.
La "Close" permette, dal lato server, di attendere che i thread deputati alla gestione della comunicazione con uno specifico client terminino, deallocando, in seguito, il nodo relativo a tale utente dall'AVL delle connessioni. Di fatto, tale processo richiede un meccanismo di chiusura della connessione del tutto simile a quello del reale TCP, composto di: FIN→ACK e FIN→ACK.
Per rendere il tutto più robusto, si è impostato un timeout specifico oltre il quale, sia dal lato client che dal lato server, si chiude comunque la connessione, nonostante non ci sia alcuna risposta dall'altro lato. Inoltre, si è disposto che chi chiude la connessione non debba aspettare

necessariamente due volte il tempo di vita medio di un segmento, pari circa a 4 minuti, in quanto, nonostante tutte le divisioni logiche tra livello 5 e 4.5, si è comunque a livello 5 e quindi questo si tradurrebbe, nel nostro caso, in un terminale che deve rimanere inattivo per tale intervallo di tempo.

Server

La modularità nella gestione del codice ci ha permesso di produrre un codice *server.c* particolarmente compatto e destinato a due soli compiti fondamentali.
Il main inizia allocando quanto necessario e inizializzando tutte le variabili necessarie, si apre il socket UDP e si genera l'albero AVL per il mantenimento delle connessioni pendenti, potremmo infatti chiamare ciascun nodo dell'albero come il nostro socket TCP 4.5.
Viene poi effettuato il collegamento del socket UDP a un endpoint interno mediante la funzione *bind*, il nostro server sarà in esecuzione sulla porta 5000.
Infine, si invocano le funzioni per l'impostazione della massima dimensione del backlog per prevenire attacchi di syn flooding, l'analogo della chiamata *listen*() offerta per i socket TCP classici.

A questo punto, il server può entrare nel ciclo di ascolto dal socketUDP, mediante la funzione *recvfrom*, discriminando i pacchetti di syn dal traffico normale.
Il server ha il delicato compito di effettuare la demultiplazione: ricevuto un pacchetto cercherà, quindi, nell'AVL, la connessione desiderata, mediante l'indirizzo del client mittente, qualora la connessione non fosse presente potrebbe semplicemente scartare il pacchetto ricevuto.
Alternativamente, il pacchetto verrà inserito all'interno del buffer di strutture *Packet* presente nei nodi dell'AVL dove i dati verranno raccolti dai thread di gestione della connessione.

Qualora si dovesse creare una nuova connessione si genererebbe anche un nuovo thread per la gestione del protocollo applicativo per il client che ne ha fatto richiesta. La logica eseguita da questo thread è ancora definita nel codice del server in *tMenu.*

Tale thread implementa le richieste per il protocollo applicativo mediante l'utilizzo delle funzioni di *WriteOnBuff* e *ReadFromBuff,* contenute in *FileManaging.h* e già precedentemente discusse.

Client

Analogamente a quanto detto per il server, il lato client grazie alla gestione modulare del codice risulta particolarmente breve, costituendosi di una sola funzione oltre al main.

Il main risulta duale rispetto a quello visto nel server, l'importante differenza sta nella funzione *connectTCP,* anch'essa definita in *ReliableDataTransfer.h* e precedentemente discussa, che permette al client di avviare l'handshake con il server.

Per il client la scelta riguardo a quale entità affidare il trasferimento dei pacchetti in arrivo sul socket UDP e a quale lasciare il compito di gestire il livello applicativo, è stata opposta. Giustificandola con l'osservazione già fatta prima sulla semplicità delle operazioni che al thread *tClientListen* sono richieste.

Sarà quindi il proprio il main process a chiamare la funzione *menu* per la gestione del protocollo applicativo.

Ancora, come nel lato server, le varie operazioni, che sono richieste per il client, vengono implementate mediante le logiche viste delle funzioni *WriteOnBuff* e *ReadFromBuff,* contenute in *FileManaging.h.*

Risultati Simulativi

L'applicazione, nella fase di testing, ha rispecchiato i trend attesi, infatti, riducendo la finestra di trasmissione sia attraverso una riduzione del MSS, che della dimensione del buffer di ricezione, piuttosto che di quello di trasmissione, le performance diminuiscono in parallelo con la diminuzione del throughput del livello 4.5.

Da un'analisi empirica dei risultati, sicuramente non caratterizzati da un eccessivo rigore scientifico in termini di misurazioni, si evince che il tempo di trasferimento cresce linearmente anche rispetto alla probabilità di perdita.

Rispetto alla precedente versione, consegnata nel mese di Luglio 2020, le performances sono migliorate significativamente, data l'aggiunta del controllo di congestione.

Il testing è stato svolto su un MacBook Pro 2019
S.O.: MacOS Catalina 10.15.6
Processore: 2,6 GHz Intel Core i7 6-core
Memoria: 16 GB 2667 MHz DDR4

Si presentano i risultati ottenuti nella configurazione in cui si presenta:

	Tempo(s)	Carico sulla CPU a regime	Dimensione (Byte)
PUT provaclient.txt	1.57	-	833
PUT icona.png	2.07	-	854
GET bandiera.jpeg	31.56	30%	4392
GET mare.jpeg	2' 06''	26%	54.000

Miglioramenti e Bug conosciuti

Di seguito si riportano alcuni miglioramenti, che potrebbero dare notevoli valori aggiunti alla nostra applicazione, ma anche dei Bug conosciuti.

Miglioramenti possibili:

- Interfaccia grafica più appetibile e moderna.
- Miglioramento in termini di robustezza del server rispetto a crash o attacchi da parte di un utente di un'applicazione client.
- Servizio di Autenticazione, già progettato ma da implementare.
- Eliminazione dei File direttamente dall'applicazione, tramite un ulteriore comando: DELETE.
- Timer dal lato server per chiudere una connessione, e quindi eliminare un nodo dell'AVL, quando un particolare Client non invia dati per un certo lasso di tempo.
- Offerta di un vero e proprio servizio di Cloud, in modo tale che il Client abbia una cartella virtuale e decida se scaricare in loco o meno dei File.

Bug conosciuti:

- La funzione *clock()* della libreria *time.h* non sembra funzionare sempre nello stesso modo. Essa viene impiegata nel nostro codice per l'implementazione dei timer, utilizzando la macro *CLOCKS_PER_SEC* che, in relazione con la differenza tra un istante finale ed uno iniziale, dovrebbe dare il lasso di tempo desiderato in secondi. Su Ubuntu

questo accade sempre, ma su macOS Catalina 10.15.6, dopo l'ultimo aggiornamento, può capitare saltuariamente che invece di ottenere dei secondi, nel modo appena descritto, si ottengano risultati del tutto diversi che vedono numeri molto piccoli, nell'ordine di 10-3 o 10-4, non rispettando neanche i secondi effettivamente trascorsi. Tutto ciò concorre nel far fallire, quando si verifica questo spiacevole inconveniente, la funzione di EXIT, in quanto essa si rapporta ad un timeout massimo di 10 secondi, molto difficili da raggiungere.

- Quando si aumentano significativamente i valori di alcune variabili, come *MSS* o *txWnd*, sempre e solo su macOS Catalina 10.15.6, si può incorrere in un errore particolare, chiamato *bus error,* che si verifica non appena si lancia il programma. Su Ubuntu tale problema non si è mai verificato.