PROGETTO RETI DI CALCOLATORI

di **Giovanni Castellano** matricola: **0124001514**

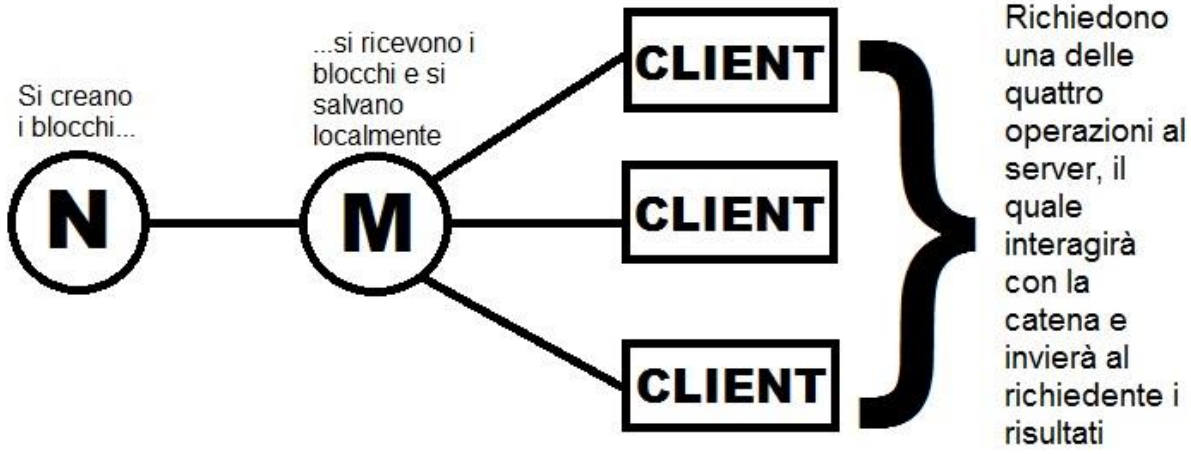
C.d.L. Informatica A.A. 2018/2019

LEGENDA

1. Descrizione del progetto

Il progetto della traccia 2 chiede di progettare ed implementare una rete client-server per la simulazione di una blockchain, con la gestione dei blocchi contenenti le transazioni in modo tale che i client possano richiederne la visualizzazione o, comunque, l'interazione in qualunque momento. I blocchi sono strutture contenenti, oltre alla transazione, anche un ID e un tempo: l'ID è sequenziale, ed ogni nuovo blocco riceve come ID l'ID del precedente incrementato di 1, mentre il tempo è un numero casuale tra 5 e 15, che stabilirà quanti secondi separano l'operazione di creazione del blocco dall'operazione di inserimento dello stesso nella catena. La catena è implementata come un array di blocchi a cui si aggiungono i blocchi creati automaticamente ogni [5;15] secondi. I nodi partecipanti al cuore della struttura, aventi compito la creazione e la custodia dei blocchi, sono il nodo N e il nodo M (da traccia chiamato blockserver, rinominato per semplicità nodo M, Middle Node, dal momento che non svolge solo la funzionalità di server, ma è il nodo che fa da ponte tra l'N e i client): il nodo N crea i blocchi attendendo il tempo definito dalla variabile tempo del blocco appena generato, lo aggiunge alla sua blockchain e lo invia al nodo M, il quale ne conserva una copia locale con cui interagisce laddove i client a esso connessi richiedano una delle quattro operazioni disponibili, tutte con la catena come protagonista. I client, quindi, possono richiedere al server quattro operazioni diverse, precisamente, le ultime X transazioni, valore inviato dal client stesso al server, che le raccoglie e le invia al client richiedente; possono inoltre richiedere i dettagli di una transazione inviando un certo ID al server, oppure possono richiedere il totale transitato fino a quel momento. Infine, l'ultima richiesta che un client può inviare al server è anche la più specifica, ovvero, dato un socket (IP_Porta), visualizzare tutte le transazioni in cui quel socket è coinvolto.

2. Descrizione e schemi dell'architettura



2.1 – La struttura della rete

La rete si struttura, come sopra citato, in due nodi principali e dei client di numero indefinito. Il nodo N è colui che crea i blocchi e li inserisce nella catena; al nodo N, si collega solo ed esclusivamente un nodo M come client, per cui il nodo M avrà ambedue i lati client (per comunicare col nodo N) e server (per accogliere i client e ascoltare le loro richieste). I client, infine, sono semplici client che si collegano al nodo M e che non conoscono l'esistenza del nodo N: i client sono il software con cui interagisce l'utente e che, quindi, avrà l'interfaccia più articolata, che comunica con l'utente guidandolo in ogni suo passo per effettuare la richiesta desiderata al server.

2.2 – La progettazione dei nodi

Nodo N – Il nodo N è progettato per avviarsi, preparare il suo ambiente inizializzando le variabili necessarie ed attendere un nodo M. La preparazione consiste nella creazione di quattro socket IP_Port generati casualmente secondo la struttura "**127.0.0.1xx:15xxy:**":

- Gli x saranno valori compresi tra 0 e 9
- L'y sarà un valore compreso tra 1 e 9, dato che non si vuole creare un socket fittizio che abbia la stessa porta del nodo M, ovvero 15000

In sintesi, gli indirizzi IP potranno variare tra 127.0.0.100 a 127.0.0.199, mentre le porte potranno variare dalla porta 15001 alla 15999.

Dopo aver creato gli host fittizi, crea un blocco Genesis con ambo ID e tempo inizializzati a 0: questo è il nodo da cui deve cominciare la catena per ogni

nodo N e, altrettanto, per ogni nodo M, e che non verrà mai visualizzato dai client.

Dopo aver preparato il necessario, il nodo N si mette in attesa di un nodo M e, una volta collegato, il nodo N comincerà la sua creazione tendente all'infinito di blocchi della catena, a partire dall'ID 1. Una volta creato un blocco, occupando opportunamente i campi *id*, *time* e *str*, aspetta i secondi registrati in *time* e inserisce il neocreato blocco nella sua catena, e lo invia anche al nodo M. Il ciclo di creazione si interrompe solo se la comunicazione con il nodo M fallisce, per cui, il nodo N attende che un nuovo nodo M lo contatti per continuare la procedura: in questo caso, però, il nodo M riceve prima (ed immediatamente) tutti i blocchi già contenuti nella catena del nodo N, per poi continuare a ricevere i successivi blocchi che il nodo N andrà a creare nella sua routine standard. Se più di un nodo M dovessero essere avviati, solo il primo a connettersi interagirà col nodo N stesso dato che esso accetta un solo client e la sua coda ha spazio solo per un nodo M per volta, dopo di che, il nodo M successivo a connettersi verrà messo in coda, gli altri verranno ignorati. Laddove il primo nodo M dovesse andare offline, il secondo nodo M in coda verrà accolto come descritto sopra, inviandogli tutti i blocchi già creati, inoltre, la coda della Listen() avrà un posto libero per un altro nodo M da poter mettere in coda.

Nodo M – Il nodo M è il nodo più complesso di tutti, dato che deve svolgere la funzione di client per comunicare col nodo N e la funzione di server, per accogliere e comunicare con i client. Così come il nodo N, anche il nodo M si prepara prima di interagire con gli altri inizializzando il dovuto e poi si connette ad un nodo N, se disponibile, altrimenti rimane in attesa senza andare avanti finché un nodo N non è online. Quando la connessione col nodo N avviene con successo, il nodo M invia al nodo N il suo numero di blocchi, che essendo un nuovo nodo, è pari a zero, per cui il nodo N invia solamente il nodo Genesis non avendo altro a disposizione, oppure invia tutti i nodi a partire dal Genesis se ha già interagito con un altro nodo M precedentemente. A questo punto, dopo essersi correttamente sincronizzato col nodo N, il nodo M si lega alla sua porta, la 15000, e inizia ad accogliere client con una coda di Listen() con 10 spazi liberi, poi, avvia la sua routine ripetuta all'infinito.

La gestione dei descrittori di socket avviene tramite **I/O Multiplexer**, quindi grazie alla funzione select(), che consente di rimanere in attesa finchè uno o più descrittori non sono disponibili alla ricezione oppure finchè non scade il tempo di timeout definito come ultimo parametro della select() stessa. I descrittori vengono quindi tutti inseriti in una variabile di tipo fd_set nominata *fset* con l'aiuto delle macro messe a disposizione, e si impostano 3 secondi di

timeout, per poter rispondere alla disconnessione eventuale del nodo N, di cui verrà precisata la reazione successivamente.

I descrittori vengono quindi ascoltati nell'ordine che segue: socket controllato dalla Listen, con conseguente Accept se il socket è pronto alla ricezione, descrittore che comunica col nodo N, denominato *Nfd*, per la ricezione dei nuovi blocchi, e infine controlla tutti i descrittori dei client.

Nel caso in cui il **nodo N dovesse scollegarsi**, al momento in cui si disconnette, la funzione select() riceve valore 0 dal suo descrittore e capisce immediatamente che il nodo N si è disconnesso. Il nodo M quindi genera un thread che resta in attesa di un nuovo nodo N, provando a connettersi ripetutamente ogni due secondi, dopodiché riceve i blocchi di sincronizzazione del nuovo nodo e li scarta, dato che un nuovo nodo N ha solo il nodo Genesis che, se aggiunto alla lista non avente indice 0, può essere visualizzato dai client, quando non deve essere visto da nessun client dato che non contiene una transazione reale. Se il **client dovesse scollegarsi**, il descrittore del socket per comunicare a quel client verrà rimosso dalla lista dei descrittori, e il client dovrà ricollegarsi nuovamente se vuole continuare con ulteriori richieste.

I segnali sovrascritti sono il **SIGPIPE**, che viene ignorato dato che ad ogni funzione di invio o ricezione vi sono controlli che assicurano la corretta individuazione di problemi di connessione, e il **SIGINT** anche è sovrascritto per far si che, laddove il nodo N risulti ancora attivo, gli venga inviato un valore di uscita prima di chiudere il nodo M, in modo tale che possa accorgersene e smettere di generare blocchi.

Client – i client sono i software più semplici, ma anche quelli con interfaccia più complessa, dal momento che devono interagire con l'utente. I client hanno molto meno codice rispetto ai due nodi sopra descritti, dato che non devono gestire la catena di proprio conto, ma hanno come compito di interrogare il nodo M per conoscere lo stato della catena, senza averne una copia locale. Anche i client, all'avvio, inizializzano le poche variabili utili alla ricezione dei blocchi dal nodo M, sovrascrivono i segnali **SIGPIPE** (ignorato), **SIGINT** e **SIGQUIT** (sovrascritti con la funzione Exit(), in clientheader.h) e si connettono al nodo M, rimanendo in attesa se nessun nodo M è online, altrimenti, se la connessione è avvenuta con successo, mostrano il loro menù di scelta, dettagliato e di semplice interpretazione, che chiede di inserire un valore da 1 a 4 per le varie operazioni che il client può richiedere, o qualsiasi altro valore numerico per chiudere il client inviando un valore *DISCONNECT* al nodo M, in modo tale che possa cancellare il client dalla lista. Se provando a contattare il nodo M il client in questione dovesse fallire, reputerà il nodo M

disconnesso e tenterà di riconnettersi, ogni due secondi, ad un nuovo ipotetico nodo M.

Il client, per ogni operazione, richiede l'interazione con l'utente senza contattare il nodo M, dopo aver ricevuto tutti i dati in input dall'utente via scanf(), invia il necessario al nodo M per far si che possa rispondere alla domanda e inviare dati che il client attenderà per poi mostrare in output all'utente.

3. Descrizione e schemi del protocollo applicazione

La comunicazione tra nodo N ed M avviene in pochi passaggi utili alla sincronizzazione dei due nodi, e poi la comunicazione diventa unilaterale dal momento che il nodo M aspetta i blocchi dal nodo N senza dovervi interagire prima. La connessione inizia non appena il nodo M effettua, per l'appunto, la richiesta di connessione come client al nodo N, che invece ha ruolo di server. Il nodo M quindi invia il numero di blocchi che ha in locale, ed il nodo N capisce quindi quanti ne deve inviare per effettuare la sincronizzazione:

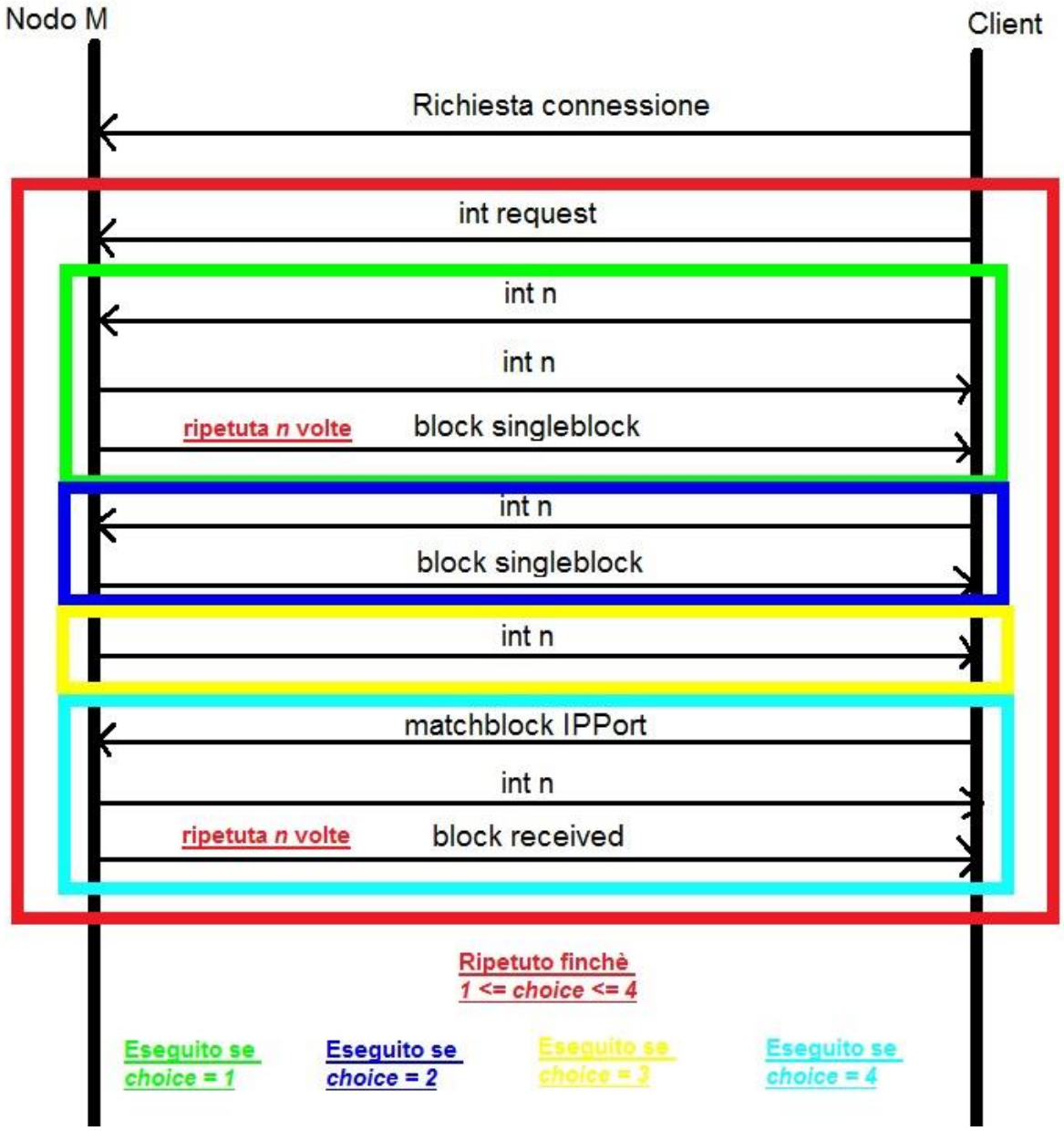
- Se il nodo M ha zero nodi, il nodo N invia a partire dal blocco Genesis
- Se il nodo M ha più nodi del nodo N, il nodo N invia a partire dal blocco con ID = 1

Il nodo M ha zero nodi nel caso solitamente previsto, ovvero quando è stato appena mandato in esecuzione e vuole connettersi, per la prima volta, al nodo N, il caso opposto invece capita esclusivamente quando si disconnette il nodo N, ma il nodo M rimane online: in questo caso, il nodo N perde tutti i suoi nodi e ricomincia da ID = 1.
Nonostante, in quest'ultimo caso, la comunicazione riprenda a funzionare normalmente, è **fortemente sconsigliato** collegare un secondo nodo N allo stesso nodo M, dato che il nodo M conserva i vecchi blocchi e il nodo N,

invece, ricomincia a mandare da ID = 1 in poi, causando la **duplicazione** dei blocchi nella catena del nodo M. Ciò comporta che l'utilizzo del secondo comando da client, la richiesta di uno specifico ID, richieda l'effettiva locazione della transazione e non più il suo ID, cosa che il client può intuire, ma non sa effettivamente e potrebbe risultare complicato portare a termine; tutte le altre richieste funzionano normalmente, nonostante nei loro risultati compaiano degli ID potenzialmente avanzati troncati dalle transazioni successive, che ricominciano da ID = 1. L'implementazione della riconnessione è un "di più", ma dal momento che il nodo N si disconnette, **è consigliato non riconnetterne altri**, interrompere prima anche il nodo M e poi ricominciare con la creazione delle catene, mentre se si preferisce, si possono ancora utilizzare i client con il nodo M disconnesso dal nodo N senza problemi, semplicemente, la catena non riceverà nuovi blocchi.

Ritornando al protocollo applicazione, dopo aver ricevuto il numero dal nodo M, il nodo N calcola quanti blocchi deve inviare, e invia prima il numero di blocchi che vuole inviare, poi i blocchi stessi, in modo tale che il nodo M sappia quante volte ascoltare sul socket. Dopo aver effettuato la sincronizzazione, la comunicazione diventa **quasi unilaterale**, dato che è richiesto che il nodo N invii i neocreati blocchi al nodo M, senza interventi del nodo M, salvo il caso in cui arrivi un blocco **con ID maggiore dell'ultimo ID conservato dal nodo M**: in questo caso, il nodo M chiede al nodo N i blocchi a partire dal suo ultimo ID, in modo da sincronizzarsi nuovamente. Questo caso **non** può accadere in **locale**, avviene solamente in caso di lontananza tra nodo M e nodo N che può causare la perdita di qualche pacchetto.

Ovviamente, la connessione segue il protocollo **TCP** con la creazione di un socket e la comunicazione attraverso esso, è impensabile eseguire un'operazione quale l'invio di un intero per definire quanti pacchetti verranno inviati successivamente se, con un protocollo UDP, non è garantito l'ordine d'arrivo dei pacchetti.



La comunicazione tra nodo M e client avviene invece in modo differente in base all'operazione che l'utente, utilizzatore del client, vuole eseguire. Anche in questo caso, si inizia con la richiesta di connessione da parte del client al nodo M, che in questo caso svolge ruolo di server. Il client, innanzitutto, invia **un intero** che indica la richiesta, poi, invia i dati necessari affinché quella richiesta possa essere soddisfatta dal server, eccetto per la richiesta 3, che non richiede ulteriori dati dal client.

- Se la richiesta è la **prima**, in cui si richiedono le ultime N transazioni, il client deve inviare n, per l'appunto, in modo tale che il server sappia quante transazioni sono richieste. Il valore viene **rinviato al client** da parte del nodo M dato che potrebbero essere state chieste più transazioni di quante ve ne sono disponibili, per cui, il client deve sapere quante ne arriveranno poiché, con mancata comunicazione del server, sarebbe rimasto in attesa di ulteriori pacchetti che non sarebbero mai arrivati. Successivamente, verranno inviati tanti pacchetti

quanto è il valore di n in ritorno, in modo che il client possa mostrarli in output all'utente.

- Se la richiesta è la **seconda**, ovvero la richiesta di una transazione specifica tramite ID, ovviamente è necessario che il client invii prima di tutto l'ID del pacchetto di cui vuole i dettagli, successivamente, rimane in attesa di un singolo blocco, dato che il server invierà il blocco corrispondente a **quell'ID se esiste** oppure invierà il blocco con **ID maggiore** se l'ID richiesto è maggiore del numero di blocchi esistenti.
- Se la richiesta è la **terza**, quindi, se il client richiede il totale transitato, il nodo M sarà subito capace di restituirgli il risultato, dato che possiede una variabile globale aggiornata ad ogni aggiunta di blocco che estrapola il valore della transazione dal blocco e lo somma al totale transitato fin a quel momento, per cui, in caso di tale richiesta, sarà necessario un banale invio di un **intero**.
- Se la richiesta è la **quarta**, il client deve inviare una stringa che individua un socket, con cui il nodo M andrà a confrontare i mittenti e destinatari delle transazioni per selezionare solo le transazioni in cui quel socket è coinvolto, sia esso mittente o destinatario. L'invio dei dati avviene quindi dopo l'invio della richiesta secondo l'ordine per il quale il client invia un ***matchblock*** contenente una stringa di **25 caratteri** all'interno utilizzata dal nodo M per individuare i suoi mittenti e destinatari, poi, dopo aver selezionato tutte le transazioni, invia al client **un intero n** che definisce quante transazioni verranno inviate subito dopo, sempre in modo che il client sappia per quanto rimanere in attesa, e successivamente, vengono inviati **gli effettivi blocchi** delle transazioni positive al socket scelto dall'utente inizialmente.

Anche in questo caso, la connessione avviene via protocollo **TCP**, per lo stesso motivo riportato descrivendo la linea temporale tra nodo M e nodo N.

4-5. Dettagli implementativi del client/server

Partendo dall'header in comune con tutte e tre le parti, il "Generalheader.h", si possono capire le **struct** da cosa sono composte e le **define** piuttosto comuni nei vari programmi:

- **struct**: le struct sono due, in questo header, e sono la *block*, una struttura formata da due interi e una stringa di 100 byte, che vuole proprio raffigurare i blocchi della blockchain, con i due interi raffiguranti **id** e **time**, e la stringa **str** raffigurante la transazione effettuata. È la struct più utilizzata dato che definisce la dimensione di un blocco della blockchain e viene usata sia dal nodo N che dal nodo M per definire la catena (array di block), sia dal client per ricevere i blocchi richiesti.

- **define**: le define sono state create per dare maggiore leggibilità al codice, difatti esse sono TRUE=1 e FALSE=0, perché spesso è stato necessario definire valori booleani, successivamente troviamo TO_INT=&207 e TO_CHAR=|48 che effettuano le due conversioni da char ad int e da int a char, sfruttando gli operatori bitwise & e |. Seguono DOWNCASE=|32 e UPPERCASE=&223 che servono, dato un carattere contenente una lettera dell'alfabeto, a ridurla a minuscolo o a ingrandirla a maiuscolo. Infine, vi è la define della funzione max, che dati in input x e y ritorna il massimo, e le define dei vari colori, che non sono altro che printf() alle quali viene passato il codice del colore per impostarlo sulla shell, per cui, gli output successivi saranno di quel colore. Queste define sono poco utili al completamento della richiesta del progetto, ma conferiscono un'immediatezza nella comprensione degli output (i colori verranno spiegati nel manuale utente).
- **enum**: infine, vi è una enum particolarmente importante, dal momento che definisce un valore per ogni richiesta che un client può effettuare al nodo M, con in aggiunta DISCONNECT per poter comunicare col client che si vuole uscire dal programma e che il nodo M deve rimuovere quel client dalla sua lista.

Le liste utilizzate, quindi ListInt e ListStr, contenute nei rispettivi file header, sono liste a doppio nodo sentinella che utilizzano determinate funzioni preparate appositamente e richiedono prima che **il main** richiami la funzione Initialize(), a cui va passato il puntatore al puntatore della lista (**), dopodichè, la lista sarà pronta all'utilizzo tramite le altre funzioni messe a disposizione:

- Push_Back(): il push back aggiunge un nuovo nodo alla fine della lista
- Next(): la funzione next modifica la variabile **pointed** all'elemento successivo della lista
- Previous(): la funzione previous modifica la variabile **pointed** all'elemento precedente della lista
- GetPointed(): nel caso della ListInt, ritorna il valore dell'elemento puntato, nel caso della ListStr, inserisce nella variabile passata come secondo parametro la stringa conservata nell'elemento puntato
- RemovePointed() ed ErasePointed(): le funzioni rimuovono l'elemento dalla lista, a differenza che ErasePointed() lo elimina anche tramite free()
- RollToStart(): l'elemento puntato è il primo della lista
- RollToEnd(): l'elemento puntato è l'ultimo della lista

Tutte le funzioni avranno un''I' (es: NextI) come suffisso se facenti parte della lista ListInt, oppure una 'S' (es: NextS) come suffisso se facenti parte della lista ListStr.

Riguardo le connessioni client e server, la funzione che consente ad un client di connettersi ad un server è conservata nell'header "**Connection.h**" con nome **TCPConnection()**, che richiede un **intero** per la porta e un **char*** per l'indirizzo IP come parametri, ed esegue tutte le funzioni necessarie alla connessione, quindi inizializza la struct sockaddr_in coi vari dati passati come parametro ed imposta **AF_INET** come famiglia, dato che la connessione deve essere TCP, poi chiama le funzioni Socket e Connect, per definire il socket e connettersi attraverso quello, e ritorna un valore **che equivale al socket** se la connessione è andata a buon fine, altrimenti ritorna -1. D'altra parte, invece, se un server vuole legarsi ad una porta, utilizza la funzione contenuta in "**Binder.h**" di nome **TCPBindToPort()**, a cui si passa **la porta** come parametro e la funzione esegue le operazioni necessarie all'appropriarsi della porta richiesta; nella struct sockaddr_in inserisce AF_INET come famiglia per il TCP, INADDR_ANY come indirizzo a cui devono arrivare le richieste, dato che qualsiasi indirizzo va bene per quella rete, e infine la porta passata come parametro come sin_port. A questo punto, chiama la funzione Socket per definire il socket in ascolto, setta le opzioni del socket per far si che la porta possa essere riutilizzata, e successivamente chiama la funzione Bind che se va a buon fine, non esegue l'if e ritorna **il descrittore di socket**, altrimenti, entra nell'if, scrive un messaggio sull'output in cui rende noto il problema e prova, ogni due secondi, ad **appropriarsi della porta**, finchè essa non diventa libera. Le funzioni utilizzate in questi due header sono a loro volta contenute nell'header "**wrapper.h**" e come suggerisce il nome dell'header, sono funzioni wrapper.

Volendo andare nello specifico per ogni programma, è possibile analizzare, per i nodi M ed N, che la blockchain è un array dinamico di *block*, il nodo N crea un blocco dopo aver aspettato, tramite sleep(), il tempo del blocco stesso, e poi lo aggiunge alla catena allo stesso modo in cui il nodo M, dopo averlo ricevuto via socket, lo aggiunge alla sua catena.
Il nodo N utilizza una select() non bloccante per verificare, ad ogni blocco creato, se il nodo M vuole comunicare qualcosa, o semplicemente, se il nodo M si è disconnesso: nel caso vi sia comunicazione, entra nell'if successivo alla select, se invece non vi sono comunicazioni da parte del nodo M, passa alla creazione del blocco successivo. Il client invece riceve i blocchi uno ad uno in base alla richiesta e li mostra in output, cancellandoli subito dopo, non ne conserva una copia locale dal momento che non ha bisogno di farlo, non gestisce la catena in alcun modo, ma ha il solo compito di interfacciare un utente alla catena contenuta nel nodo M.

6. Manuale utente

La compilazione dei file C è stata fatta nella stessa cartella per tutti e tre i file, per cui, è necessario rinominarli aggiungendo il parametro -o, inoltre, alcuni file necessitano di librerie da passare come parametro alla compilazione. I comandi consigliati sono quindi:

- `gcc N-node.c -o N-node.out -lm`
- `gcc M-node.c -o M-node.out -lm -pthread`
- `gcc Client.c -o client.out -lm`

Per lanciare in esecuzione il nodo M, è richiesto come parametro l'indirizzo IP dell'host su cui si trova il nodo N, invece per i client è richiesto l'indirizzo IP del nodo M; il nodo N è l'unico programma che non ha bisogno di parametri. Gli eseguibili possono essere lanciati in esecuzione **in qualsiasi ordine**, il client attenderà un nodo M, che a sua volta attenderà un nodo N prima di accettare i client, per cui, avviando un client e un nodo M, sarà necessario un nodo N per far partire il tutto. Il nodo N e il nodo M non hanno bisogno di alcuna interazione con l'utente, il client è l'interfaccia con cui l'utente può visualizzare il contenuto della catena di blocchi.

I programmi utilizzeranno dei colori per l'output in base a che tipo di messaggio si vuole visualizzare:

- **Blu**: output importante di cui si vuole rendere noto il lettore al meglio
- **Verde**: operazione che poteva trovare ostacoli effettuata con successo
- **Giallo**: il programma ha incontrato un errore e cerca di risolverlo automaticamente
- **Rosso**: messaggio di errore che solitamente segnala un cambiamento radicale nel funzionamento o l'impossibilità di svolgere qualche operazione

Il client si apre con un messaggio di attesa di un nodo M se esso non è online o se sta aspettando, a sua volta, un nodo N, e l'utente non potrà interagire finché il client è in questo stato. A connessione avvenuta, il client mostrerà un menù con quattro scelte più una generica, che chiuderà il programma. Le quattro scelte richiedono un **valore numerico** come input per richiedere poi, se necessario, altre informazioni all'utente; **un inserimento di valori NON numerici causa comportamenti aleatori nel programma**, per cui si prega di attenersi alle richieste, altrimenti il client potrebbe finire in un ciclo infinito o, comunque, si potrebbe compromettere la corretta interagibilità dovendo poi chiudere e riaprire il programma. In base al valore scelto, son solitamente necessari altri parametri inseriti dall'utente:

- Nel primo caso, sarà necessario inserire un ulteriore valore **intero**, dato che il client vuole sapere quante ultime transazioni richiedere. Se le transazioni son arrivate ad un ID pari a 60, ad esempio, e si richiedono 3 transazioni, verranno mostrate le transazioni **con ID 60, 59 e 58**, se

invece si chiedono ad esempio 70 transazioni, verranno mostrate **tutte e 60 le transazioni disponibili**, dato che il nodo M effettuerà dei controlli di out of bound. Se non vi sono transazioni da mostrare, sia perché l'input è zero, sia perché il server non possiede transazioni, verrà mostrato un messaggio d'errore.

- Nel secondo caso, sarà necessario un **intero** per definire l'ID da richiedere al nodo M; è necessario inserire un **ID maggiore di zero**, se esso esiste nella catena, verrà restituita quella transazione, altrimenti se è maggiore dell'ultimo ID inserito, sarà restituita **proprio l'ultima transazione** inserita. Se si inserisce un valore minore di uno, verrà mostrato un messaggio di errore e non verranno effettuate richieste al server.
- Nel terzo caso, non sono necessari ulteriori input e verrà visualizzato **immediatamente** il totale transitato.
- Nel quarto caso, è necessario inserire una stringa come input, e l'operazione è piuttosto delicata:
  è necessario inserire una combinazione IP:Porta del tipo **127.0.0.100:15001:** in cui è di fondamentale importanza non mancare i due **':'**, dal momento che la funzione del server che identifica la combinazione in una transazione necessita del secondo carattere **':'** per accertare la presenza della combinazione nella transazione, quindi, se in input si dovesse ad esempio inserire **127.0.0.100:15001** come combinazione senza il **':'** finale, **l'esito sarà negativo** anche se la combinazione è corretta. Dopo aver inserito la combinazione, inoltre, verrà chiesto di inserire un ulteriore carattere che sarà **'s'** se si vuole confermare l'invio al server di quanto inserito (che verrà mostrato nuovamente in output), altrimenti qualsiasi altro carattere negherà l'invio, scrivendo in output il messaggio d'errore di operazione annullata. Non è necessario badare ai caratteri **'\n'** inseriti, la funzione che effettua i confronti **non necessita di altro dopo il carattere ':'** ed ignorerà qualsiasi cosa dovesse esserci.

Qualsiasi altro **valore numerico** inserito quando si deve effettuare la scelta chiuderà il programma. I client possono essere chiusi anche via segnale SIGINT o SIGQUIT, entrambi sovrascritti, che invieranno un pacchetto DISCONNECT al nodo M che saprà di dover rimuovere il client in questione dalla sua lista dei client attivi.