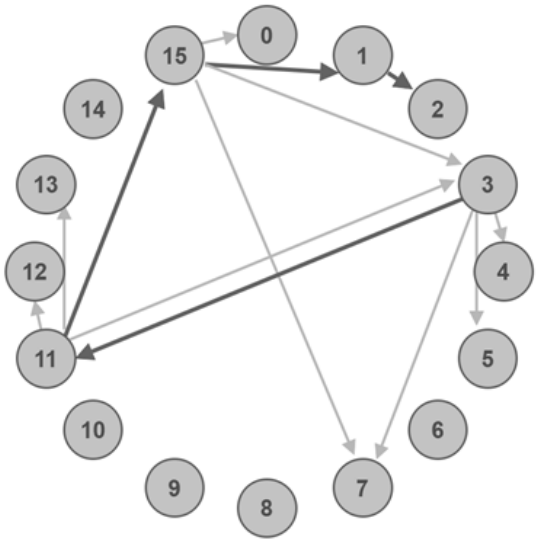
PROGETTO RETI DI CALCOLATORI

IMPLEMENTATION OF P2P CHORD PROTOCOL USING THE SOCKET LIBRARY IN C

Chord è un protocollo distribuito per la ricerca efficiente di nodi in una rete peer-to-peer basato su Distributed Hash Table (DHT). Utilizza una struttura ad anello per localizzare i dati con complessità logaritmica.



Calderon Gianluca – 31383A

luglio 2025

Introduzione

Chord è un protocollo di tipo **Distributed Hash Table (DHT)** progettato per la gestione efficiente della distribuzione e ricerca di chiavi in reti peer-to-peer.
Chord è basato su una **struttura ad anello** in cui ogni nodo e ogni chiave sono identificati da un valore hash in uno spazio di 2^m, dove *m* è la lunghezza dell'hash (160 bit, in SHA-1). Ogni nodo è **identificato da un ID** che corrisponde a una chiave nello spazio **hash**, e le chiavi vengono distribuite ai nodi in base ai loro ID.
Ogni nodo mantiene una ***finger table***, una struttura che memorizza riferimenti a nodi a distanze esponenziali, permettendo di effettuare operazioni di lookup (ricerca di una chiave) con complessità ***O(log N)***, dove N è il numero totale di nodi nella rete.
Chord supporta dinamicamente l'ingresso e l'uscita dei nodi dalla rete, garantendo la scalabilità e la tolleranza ai guasti attraverso algoritmi di stabilizzazione e aggiornamento periodico delle finger table.

Ogni nodo è responsabile di un certo intervallo di chiavi. Ogni finger table ha diverse finger entry, in particolare 160 entry, pari al numero di bit di cui è composto l'hash. Ogni entry contiene l'intervallo di chiavi cercate, e il responsabile delle chiavi, chiamato successor. Di seguito una rappresentazione di una entry, e del modo in cui si calcola l'intervallo:

start interval: $n+2^{k-1} \bmod m$
end interval: $n+2^{k} \bmod m$
successor: ID del nodo responsabile

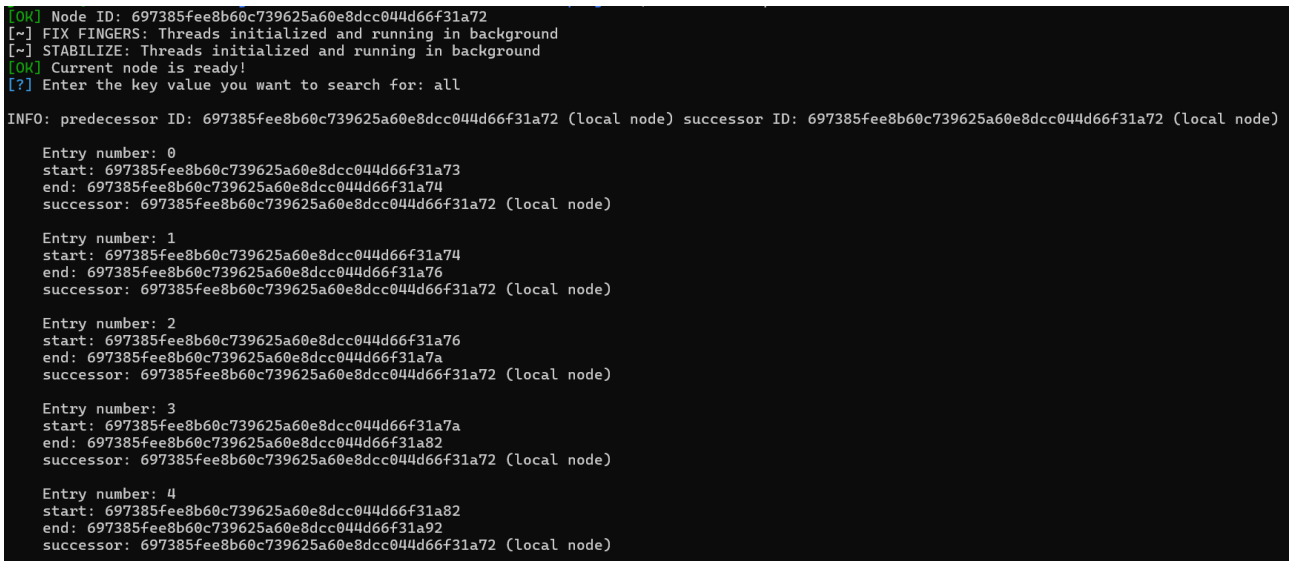
dove con *k*, si identifica la posizione *k-esima* della entry, e con *n* l'ID del nodo (nodo locale corrente).

L'intervallo di responsabilità delle chiavi di un nodo, p un intervallo chiuso a destra e aperto a sinistra. Ovvero, nodo successor della finger table, è responsabile di un intervallo di chiavi così formato:

[finger[k].start, finger[k+1].start)

Nel caso di intervallo degenere, vengono considerati tutti i valori hash che vanno dall'ultimo nodo al primo nodo, in modo da formare un anello. Questa osservazione è importante per rendere la struttura chord, una struttura stabile ad anello.

Nel caso si singolo nodo all'interno della rete, il nodo sarà responsabile di tutte le chiavi, avendo così una finger table con i successori che "puntano" al singolo nodo. Inoltre, ogni nodo tiene traccia del suo successore e predecessore all'interno del nodo. Sempre in caso di singolo nodo nella rete, questi



campi dovranno “puntare” all’unico nodo della rete (il nodo corrente). Di seguito un esempio, estratto dall’implementazione in C, di questo scenario:

L’operazione fondamentale per questo protocollo è la funzione di “*find_successor(k)”*, in grado di trovare il nodo più vicino, data una chiave K. Grazie alla struttura della finger table, basata sulle funzioni sopra descritte, la ricerca sarà esponenziale.

Infine, quando un nodo desidera entrare in una rete già formata, è necessario inserire nel nodo che sta per entrare, il riferimento al *bootstrap node*, un nodo già presente nell’anello, a cui verranno eseguite diverse query *“find_successor(n)”* in modo da determinare il punto dell’anello in cui il nuovo nodo deve inserirsi.

Implementazione iniziale

Il programma riceve in input dal terminale diversi parametri:

- **-p**: porta, unica opzione obbligatoria per far partire il programma. La porta identifica la porta su cui il server deve stare in ascolto per comunicare con gli altri nodi della rete.
- **-i**: indirizzo ip, è possibile definire un indirizzo ip esplicitamente. Qualora non venisse indicato, verrà preso l’indirizzo ip dell’host automaticamente (tramite delle funzioni che sfruttano il file “etc/host”)
- **-d**: domain name, è possibile indicare un nome di dominio. Tramite una ricerca DNS verrà estratto l’indirizzo IP.
- **-v**: grado di verbosità. Il programma prevede 3 livelli: il primo, che mostra solo i log inerenti alla fase di inizializzazione del nodo (tranne il print della finger table); il secondo livello che mostra i nodi che comunicano con un determinato nodo *n* osservando le comunicazioni tra nodi (quali funzioni, determinati nodi, fanno eseguire ad altri nodi, come ad esempio la *find_successor()*); il 3 livello mostra, oltre ai log descritti fin’ora, i log dei thread di stabilizzazione e fix_finger table e la finger table nel momento dell’inizializzazione del nodo. I log di stabilizzazione e fix sono dei log di tipo “*update*”, ovvero dei log che descrivono i cambiamenti che avvengono nella rete, in particolare i cambiamenti dei successor nelle finger table.
- **-t**: time in millisecondi che determina lo sleep() di un thread tra l’aggiornamento di una entry e un’altra della finger table, in modo da non “floodare” i nodi con i diversi update. Se non indicato, il refresh di default configurato è di 500ms.
- **-b**: identifica l’indirizzo ip del bootstrap a cui connettersi all’inizio, per inizializzare un nuovo nodo nella rete.
- **-q**: identifica la porta (Server) del boostrap node
- **-h**: viene stampato l’help con tutte le option possibili per comporre il comando.

L’ID di un nodo viene calcola tramite la funzione hash standard SHA-1, una funzione che restituisce una stringa a 160 bit. Come input, per calcolare l’ID, viene passato l’indirizzo IP e il numero di porta. All’interno del programma, per poter eseguire operazioni a 160 bit, e per poter gestire questo tipo di dati, ho utilizzato la libreria “gmp.h”, una libreria che permette di definire tipi di grandi dimensioni.

L’entrata di un nodo all’interno della rete Chord si divide in due scenari: non esiste ancora alcun nodo, esiste almeno un nodo nella rete. Tuttavia, in entrambi i casi, la prima funzione ad essere eseguita, prevista per l’implementazione di chord, è la funzione *join()*.

Non esiste alcun nodo nella rete:
In questo scenario, il primo nodo che entra, non deve lanciare il comando con un bootstrap node, in quanto è il primo nodo ad entrare. Di seguito il comando:

```
./bin/main -p 12345
```

A questo punto (dopo aver eseguito la funzione per memorizzare i parametri, e altre operazioni preliminari), viene eseguita l'operazione di *join()* per entrare nella rete. La funzione si divide in due rami: una in cui è presente il boostrap node, l'altra in cui non è presente. In questo scenario verranno eseguite le operazioni del ramo in cui non è presente il boostrap node.

```
//If the user doesn't specify a bootstrap node, assume that the current node is the first node in the chord ring
init_join_first( info: current_node_info,*current_node, verbose_flag: verbose_level);
node_copy( dest_node: &current_node_info->predecessor, src_node: *current_node);
```

Viene eseguita la *init_join_first()*, una funzione che imposta tutti i riferimenti agli altri nodi, al nodo corrente locale. Tutte le entry della finger table, il successor/predecessor del nodo punteranno al nodo corrente appena entrato.

Esiste il bootstrap node:

In questo caso, il nodo che desidera entrare contatta il bootstrap node, e gli chiede di eseguire localmente la funzione *find_successor(k)*, usando come *k*, l'ID del nodo che desidera entrare. Una volta ottenuto il nodo successore, il nuovo nodo può posizionarsi correttamente all'interno dell'anello. Inoltre, viene riempita la finger table, eseguendo, per ogni *finger[k].start* (160 in totale) della tabella, la *find_successor(),* sempre localmente al bootstrap node. Una volta finita questa parte di inizializzazione, e dopo essersi posizionato nella posizione corretta, il nuovo nodo esegue la funzione update_others(), chiamando il predecessore di ogni successor nella finger table locale. Ad ogni nodo invia il proprio ID, dicendogli per quali intervalli di chiavi il nodo entrato ha la responsabilità.

```
void update_others(const node current_node,const node_info current_node_info,pthread_mutex_t* finger_loc

    node p; //Is the node that may need to update its i-th finger to point to the new node n.
    mpz_t temp_pow;
    mpz_t id_to_find;
    int temp_exp;

    mpz_init(temp_pow);
    mpz_init(id_to_find);

    for(short i=0;i<160;i++){
        temp_exp = (i-1<0)?0:i-1;                       //Avoid to obtain -1 in the first cycle
        mpz_ui_pow_ui(temp_pow, 2, temp_exp);           //2^(i-1)
        mpz_sub(id_to_find,current_node.id , temp_pow);     //n-2^(i-1)
        //find_predecessor(n-2^(i-1)). n= current node (id)
        p = find_predecessor(id_to_find,current_node,current_node_info,finger_lock,NULL,verbose_level);
        client_update_remote_table( remote_node: p, new_successor: current_node, pos_table: i,verbose_level);
    }
}
```

Al termine dell'inizializzazione, partono i seguenti theads: ***stabilize(), fix_finger(), die(), server(), exit_signal().***

- ***Stabilize()****:* Serve a mantenere la correttezza dell'anello aggiornando il successore di un nodo. Periodicamente, ogni nodo chiede al suo successore chi è il suo predecessore: se è un nodo più vicino, aggiorna il proprio successore. Infine, notifica il successore della sua esistenza per

mantenere i collegamenti corretti. Questo controllo avviene su un intervallo aperto: si prende un nodo x (di cui si è appreso come predecessore del proprio successore) si trova nell'intervallo aperto tra il nodo corrente e il suo attuale successore. Questo aiuta a decidere se il nodo x è in realtà un successore più "vicino" e quindi più corretto.

- ***Fix_finger()****:* aggiorna periodicamente le voci della **finger table** di un nodo per mantenerla corretta ed efficiente nel tempo, soprattutto in reti dinamiche. Esegue il find_successor(finger[k].start) per mantenere le tabelle aggiornate.

Si noti che il tempo di sleep, espresso in millisecondo, viene usato all'interno del loop infinito, di questi 2 thread.

- ***Die()****:* è un thread che "dorme" su una variabile condizionale. Non appena si verifica un errore nel server, quest'ultimo sveglia il thread tramite la variabile. Una volta svegliatosi, termina tutti i thread in esecuzione e chiude il processo in maniera sicura.
- ***Server()****:* È il thread fondamentale per gestire tutte le chiamate che i moduli client dei nodi eseguono ad altri nodi. Il server è gestito tramite la funzione ***select() bloccante***. Tutti i dati vengono "parsati" come stringhe, in modo da passare più dati, di seguito un esempio di un log di un modulo client di un nodo che riceve il find_successor da un bootstrap node.

```
[~] CLIENT: connect to 127.0.1.1:12345. Send: 'FS 41677ad91e3b607688db8e734ded75199f9688b8'
[~] CLIENT: 'FS' to find node '41677ad91e3b607688db8e734ded75199f9688b8' sent to 127.0.1.1:12345
[~] CLIENT: received node '697385fee8b60c739625a60e8dcc044d66f31a72' from 127.0.1.1:12345
```

- ***Exit_signal()****:* è un thread in grado di catturare i segnali "ctrl+c" e "uccisione gentile" dal sistema operativo, e prima di eseguire il kill del processo, aggiorna i successori e predecessori, in modo da mantenere l'anello consistente. In particolare, al successore del nodo da killare gli viene detto di impostare come predecessore, il predecessore del nodo che sta per uscire, e al predecessore del nodo da killare, gli viene comunicato che il nuovo successore è il successore del nodo che sta per lasciare la rete.

```
//set the predecessor.successor = current_node.successor
//and set the successor.predecessor = current_node.predecessor
```

Quest'ultima operazione non è prevista dal paper ufficiale di Chord, ma l'ho implementata per garantire la robustezza dell'anello.

Operazioni fondamentali

Oltre alla funzione di *join()*, la prima funzione fondamentale per costruire un anello consistente e robusto, sono presenti altre funzioni fondamentali e obbligatorie per poter implementare Chord correttamente.

- ***Find_successor()****:* La funzione *find_successor()* implementa la logica per trovare il successore di un dato nodo in un sistema Chord. Essa si basa sulla funzione *find_predecessor()* per localizzare il predecessore del nodo di destinazione e ottenere direttamente il successore senza necessitare di chiamate di rete aggiuntive. Questa funzione riceve come argomenti l'ID del nodo da cercare, il nodo corrente e le informazioni associate al nodo corrente (oltre al mutex per la sincronizzazione multi-thread). La funzione invoca find_predecessor() per determinare il predecessore e quindi restituisce il successore del nodo trovato. Questo processo riduce la necessità di comunicazioni di rete, migliorando l'efficienza della ricerca in un sistema distribuito. Una volta trovato il

successore, la funzione lo restituisce al chiamante, che può poi utilizzarlo per proseguire altre operazioni nel contesto del protocollo Chord.

- ***Find_predecessor()****:* La funzione *find_predecessor()* è una parte fondamentale del protocollo Chord, in quanto si occupa di determinare il predecessore di un nodo dato un ID specificato. Inizialmente, la funzione copia le informazioni sul nodo corrente e sul suo successore nella variabile *find_node* e *find_node_successor*. La funzione si comporta in modo iterativo, controllando se l'ID cercato si trova nell'intervallo (chiuso a destra) tra il nodo corrente e il suo successore. Se l'ID non è presente nell'intervallo, il nodo corrente contatta un nodo successivo remoto (presumibilmente più vicino all'ID cercato, questo viene garantito dalla funzione *closest_preceding_finger()*), invocando la funzione client *client_get_remote_node ()*, e prosegue la ricerca. In caso contrario, se il nodo corrente è il predecessore, la funzione lo restituisce.
- ***Closest_preceding_finger()****:* La funzione *closest_preceding_finger()* ha il compito di determinare il nodo più vicino che precede l'ID desiderato, partendo dal nodo corrente e utilizzando la sua finger table. La funzione scorre la tabella del nodo corrente, partendo dall'ultimo elemento, e verifica se ogni nodo della tabella è all'interno dell'intervallo aperto tra il nodo corrente e l'ID cercato. Non appena trova un nodo che soddisfa questa condizione, lo restituisce come il "nodo precedente più vicino" (*closest preceding finger*). Se non trova alcun nodo adatto, la funzione restituisce il nodo corrente. Anche in questo caso vengono usati opportunamente i lock per garantire la correttezza della finger table tra i diversi thread in esecuzione.

Di seguito i prototipi delle tre funzioni.

```
node find_successor( mpz_t id, const node current_node,const node_info current_node_info,pthread_mutex_t* finger_lock, short verbose_level);
node find_predecessor(mpz_t id, const node current_node,const node_info current_node_info,pthread_mutex_t* finger_lock,node* output_successor, short verbose_level);
node closest_preceding_finger(const mpz_t id, const node current_node,const node_info current_node_info,pthread_mutex_t* finger_lock);
```

I controlli di appartenenza agli intervalli avvengono tramite specifiche funzioni che ho implementato:

- ***In_interval_right()****:* La funzione *in_interval_right()* verifica se un dato ID (*id_to_check*) appartiene all'intervallo aperto a sinistra e chiuso a destra, ossia (a, b], all'interno della struttura del Chord ring. In Chord, gli ID sono organizzati in un cerchio, quindi l'intervallo può "avvolgere" da un valore massimo a zero. La funzione gestisce tre casi distinti:
  Intervallo normale: quando a è minore di b, verifica se *id_to_check* è strettamente maggiore di a e minore o uguale a b.
  Intervallo “avvolto”: quando a è maggiore di b, cioè l'intervallo attraversa il punto zero, verifica se *id_to_check* è maggiore di a o minore o uguale a b.
  Intervallo degenerato: quando a è uguale a b, considerato un caso speciale, la funzione restituisce sempre 1 (vero), indicante che qualsiasi ID rientra nell'intervallo. Questo caso è utile per gestire situazioni particolari in cui un solo nodo potrebbe occupare l'intero intervallo.

- ***In_interval_left()****:* La funzione *in_interval_left()* verifica se un dato ID (*id_to_check*) appartiene all'intervallo chiuso a sinistra e aperto a destra, ossia [a, b), all'interno del Chord ring. Come la funzione precedente, essa gestisce anche qui i casi di intervallo "normale" e "avvolto" con l'aggiunta di un controllo specifico per la condizione di "intervallo vuoto" in cui a è uguale a b:
  Intervallo normale: se a è minore di b, verifica se *id_to_check* è maggiore o uguale a e minore di b.
  Intervallo avvolto: se a è maggiore di b, cioè l'intervallo attraversa il punto zero, verifica se *id_to_check* è maggiore o uguale a a o minore di b.
  Intervallo vuoto: quando a è uguale a b, la funzione restituisce 0, indicando che non esiste un intervallo valido e quindi nessun ID può appartenervi.

- ***in_interval_open()****:* La funzione *in_interval_open()* verifica se un dato ID (*id_to_check*) appartiene all'intervallo aperto (a, b), ossia che sia strettamente maggiore di a e minore di b nell'anello Chord. Come nelle precedenti funzioni, la funzione si adatta a tre possibili casi:
  Intervallo normale: se a è minore di b, verifica che *id_to_check* sia strettamente maggiore di a e minore di b.
  Intervallo avvolto: se a è maggiore di b, ossia l'intervallo attraversa il punto zero, verifica che *id_to_check* sia maggiore di a o minore di b.
  Intervallo vuoto: quando a è uguale a b, la funzione restituisce false (0) se *id_to_check* è uguale a, altrimenti restituisce true, indicando che l'ID non può appartenere all'intervallo nel caso in cui a e b coincidano.

- ***Stabilize()****:* La funzione (thread) *stabilize* è responsabile del mantenimento continuo della stabilità di un nodo all'interno del sistema Chord. Si occupa di aggiornare periodicamente il successore e predecessore di un nodo per garantire che la struttura dell'anello di Chord rimanga corretta e che la topologia della rete sia sempre aggiornata. Inizialmente, la funzione attende che il server sia pronto a partire, utilizzando un semaforo per la sincronizzazione con altri thread. Una volta che il nodo è pronto, il thread segnala che può proseguire, e la funzione inizia il suo ciclo infinito di aggiornamento.
  Il primo passo consiste nel recuperare le informazioni sul successore del nodo corrente "*n*", in particolare il predecessore del successore. Se questo predecessore rientra nell'intervallo aperto tra il nodo corrente e il suo successore, il nodo corrente aggiorna il suo successore. Una volta aggiornato il successore "*n' "* del nodo corrente, invoca la fuzione *notify()* nel nodo successore per avvisarlo che il nodo corrente n potrebbe essere il suo predecessore. Questa comunicazione avviene invocando la funzione *client_notify_successor_node()*.

Stabilize e fix_finger attento la fine dell'inizializzazione del Server thread (tramite semafori), per essere in grado di comunicare in rete correttamente. Nel caso partisse uno dei due thread, prima che il server sia pronto a ricevere le connessioni, una parte di comunicazione iniziale andrebbe persa, a causa dello scheduling dei diversi thread da parte del sistema operativo. I thread sono in loop while infinito per garantire la periodicità.

- ***Notify()****:* La funzione *notify()* è utilizzata per aggiornare il predecessore di un nodo nel Chord ring. Quando un nodo si trova a contatto con un altro nodo, può essere necessario informare il nodo corrente che un nuovo nodo si sta aggiungendo alla sua vicinanza, sostituendo eventualmente il predecessore. La funzione esegue un controllo per determinare se il predecessore del nodo corrente è già definito o se è nullo (cioè l'ID è uguale a zero). Se il predecessore è nullo, il nuovo nodo diventa il predecessore del nodo corrente.
- ***Fix_finger()****:* La funzione *fix_finger()* è una parte del protocollo di mantenimento di Chord, e viene utilizzata per aggiornare periodicamente la finger table di un nodo. la funzione inizia a scorrere la tabella, aggiornando periodicamente ogni voce. Per ogni voce della tabella viene invocata la funzione *find_successor()* per trovare il successore dell'ID (*finger[k].start*) specificato dalla voce della tabella. Se il successore trovato è diverso da quello già registrato, la voce della tabella viene aggiornata. La frequenza di aggiornamento è determinata dallo *sleep_time*, che definisce il periodo tra ogni aggiornamento. Questo approccio permette a ogni nodo di adattarsi ai cambiamenti nella rete, garantendo che le ricerche degli ID siano sempre rapide ed efficienti.

Server e client

La funzione (thread) ***server*** è responsabile per la gestione delle connessioni dei diversi nodi che compongo la rete distribuita Chord. Il server che ascolta le richieste in ingresso elabora i comandi ricevuti dai client e invia risposte appropriate. Il server gestisce una varietà di operazioni relative alla gestione della topologia di Chord, tra cui il recupero di informazioni sui predecessori e successori, l'aggiornamento della tabella e il "setting" dei predecessori/successori. Il server è stato implementato utilizzando la funzione *select()*, senza definire un timeout. Questo thread deve essere il primo thread a partire, affinché il nodo sia inizializzato correttamente. Senza l'inizializzazione del Server, i thread per la stabilizzazione rischierebbero di non comunicare correttamente con alcuni nodi presenti nella rete. La maggior parte dei messaggi tra client e server sono delle stringhe contenenti diversi parametri. Le stringhe sono così formate:

"{comando che il server deve eseguire} [param. 1] [param. 2] [param. n]"

Di seguito un esempio estratto da un log durante la connessione di un nuovo nodo, che chiede al bootstap node di eseguire la *find_successor()* localmente.

```
[~] CLIENT: connect to 127.0.1.1:12420. Send: 'FS 74a70eed5fbcfa469b7c1bbb72fca2d6b4f527c7'
[~] CLIENT: 'FS' to find node '74a70eed5fbcfa469b7c1bbb72fca2d6b4f527c7' sent to 127.0.1.1:12420
[~] CLIENT: received node '12f60fbe55ce0ff96d2c597c9625d96b7a96616a' from 127.0.1.1:12420
```

I *return_code*, per avvisare il client di successo o fallimento (quando ad esempio bisogna impostare un successore/predecessore di un nodo remoto), vengono gestiti inviando un semplice codice (fallimento: -1, successo: 1). I codici vengono inviati formattando correttamente il numero con *htons()*.

I client sono stati gestiti in maniera differente. Non ho previsto l'utilizzo di un thread client, bensì di piccole funzioni specializzate, per inviare determinati parametri, in base alle esigenze dei nodi. Di seguito i prototipi delle diverse funzioni client.

```
int client_get_remote_node(node remote_node,mpz_t find_id, node* output_node,char* command, short verbose_level);
int client_get_remote_node_info(node remote_node, node_info* output_node, short verbose_level);
int client_set_remote_node(node remote_node, node send_node,char* command, short verbose_level);
int client_update_remote_table(node remote_node,node new_successor, short pos_table, short verbose_level);
int client_notify_successor_node(node remote_node, node current_node, short verbose_level);
```

Di seguito la lista dei comandi che un client può inviare al server remoto:

- **CPF (Closest Preceding Finger)**: Calcola il nodo (ID) più vicino precedentemente rispetto a un dato ID e restituisce il nodo corrispondente. Fa eseguire al server la *closeset_preceding_finger()* localmente.
- **FS (Find Successor)**: Trova il successore di un nodo specificato dall'ID e restituisce le informazioni su quel successore. Fa eseguire al server la *find_successor()* localmente.
- **GI (Get Info)**: Recupera da un nodo le informazioni sul predecessore e sul successore del nodo corrente.
- **SP (Set Predecessor)**: Imposta un nuovo predecessore per il nodo corrente. Se l'operazione ha successo, invia una risposta positiva, altrimenti invia un errore (-1 o 1).
- **SS (Set Successor)**: Imposta un nuovo successore per il nodo corrente, con una logica simile a quella di SP.
- **UT (Update Table)**: Aggiorna la finger table con un nuovo nodo, alla posizione indicata.
- **NS (Notify Successor)**: Notifica un altro nodo che potrebbe essere il predecessore del nodo corrente, e se necessario, aggiorna il predecessore.

Osservazioni finali

Si noti che sono state create diverse strutture per implementare chord seguendo una logica il più possibile coerente. Le informazioni di un nodo sono state divise in due parti:

- **Node** che contiene le informazioni di un nodo che non posso mutare nel tempo, ovvero l'ip (string), la porta (int), l'ID (mpz_t) e l'host name (nel caso ci fosse) (string).
- **Node_info** che contiene le informazioni che possono mutare nel tempo, come il successor (node), il predecessor (node) e la finger table (finger_entry*). La finger table è stata rappresentata tramite un'array di finger_entry, un'altra struttura che ho creato
- **Finger_entry** contiene l'ID start (mpz_t), l'ID end (mpz_t) e il successor (node). Start ed End identificano l'intervallo di responsabilità.

Per eseguire correttamente il programma bisogna importare la libreria "gmp.h", per utilizzare il tipo mpz_t, in grado di supportare i dati a 160 bit, e le diverse funzioni di comparazione. È inoltre importante importare la libreria openssl/evp.h, per poter utilizzare le funzioni per creare correttamente un valore hash calcolato con SHA-1.

Si noti infine che tutte le informazioni mutabili di un nodo, sono state protette (o per lo meno, ho cercato di farlo) da possibili problemi legati alla programmazione concorrenziale, sfruttando un mutex condiviso tra i diversi thread, chiamato "*finger_lock*", per accedere correttamente in tutte le zone critiche legate all'aggiornamento, inserimento, lettura e cancellazioni di informazioni della struttura *node_info*.

Ho creato un Makefile per facilitare la compilazione del progetto. Il progetto è stato diviso in diverse cartelle per mantenere separati i codici sorgenti, gli header file e il codice binario.

Attenzione ai primi minuti di creazione dell'anello chord, a causa dei tempi di aggiornamento (determinati dal time_sleep dei thread di stabilizzazione), affinchè la ricerca di una data chiave sia coerente con i risultati che ci si aspetta di ottenere, può essere necessario aspettare qualche minuto (in media 1 o 2 minuti, per dare il tempo al thread fix_finger di comunicare con gli altri nodi). Tuttavia, il tempo dipende comunque dal tempo definito con l'option -t TIME.

Ho studiato il protocollo tramite il paper ufficiale del MIT "*Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*", seguendo lo pseudocodice fornito nella documentazione.