Carmine Francesco Matricola: 32969A

Relazione progretto: Client rete P2P GNUTELLA-LIKE

**Descrizione progetto**:

Il progetto riguarda l’implementazione di una rete **Peer-to-Peer (P2P)** ispirata al protocollo **Gnutella**.  
Ogni nodo della rete è un **servent** (un peer che funge sia da client che da server), e **non è previsto alcun nodo centrale o superpeer**.

I messaggi scambiati tra i peer sono quelli standard di Gnutella: **PING**, **PONG**, **QUERY** e **QUERYHIT**. Lo scambio dei messaggi avviene tramite **flooding**, ovvero i messaggi vengono inoltrati a tutti i peer connessi (eccetto quello da cui si è ricevuto il messaggio).

Il meccanismo di bootstrap avviene tramite un **well-known host**: l’utente inserisce manualmente un indirizzo IP e una porta a cui connettersi all’avvio.  
In alternativa, è possibile estendere il progetto per utilizzare un file di testo come **registro dei peer attivi**, aggiornato automaticamente in seguito alla ricezione di messaggi PING.

Per simulare il file system condiviso, ogni servent mantiene una lista statica di file disponibili, identificati da un **nome** e un **indice**. Nella versione reale di Gnutella, a seguito di un messaggio QUERYHIT, il download dei file avviene tramite connessione HTTP diretta; nel mio progetto, per motivi di tempo, il trasferimento vero e proprio dei file non è stato implementato.

L’obiettivo principale del progetto è stato comprendere e simulare le dinamiche di una rete decentralizzata P2P, con particolare attenzione alla gestione dei messaggi, alla tabella di routing, e alla ricerca distribuita di contenuti.

**Strutture dati utilizzate**:

Ho implementato diverse strutture dati per il corretto funzionamento del client. permettono di rappresentare lo stato della rete, i peer connessi, e i messaggi scambiati tra nodi, oltre a gestire il routing dei messaggi in modo efficiente e senza duplicazioni. E sono:

**PEER**: utilizzata per memorizzare le informazioni relative a ciascun peer connesso, sia in entrata che in uscita. Essa contiene il file descriptor (sd) del socket associato alla connessione, l’indirizzo IP e la porta del peer (addr, di tipo sockaddr\_in), e un flag active che indica se la struttura è attualmente occupata (1) o disponibile (0). Questo consente di gestire un array di peer connessi e riutilizzare gli slot liberi per nuove connessioni

**RoutingEntry**: associa un identificatore univoco di messaggio (id) al socket da cui il messaggio è stato ricevuto (sockfd). Se il valore di sockfd è -1, significa che il messaggio è stato generato localmente dal nodo stesso. Utilizzato per gestione di messaggi duplicati e ricezione della risposta.

**MessageHeader**: intestazione comune a tutti i messaggi che specifica il tipo del messaggio (type), il suo identificatore univoco (id), il valore TTL (Time To Live) che limita la propagazione del messaggio, e la lunghezza del payload (payload\_length). Il campo type permette di distinguere tra messaggi di tipo PING, QUERY, PONG e QUERYHIT, mentre id consente il tracciamento univoco dei messaggi.

A seconda del tipo di messaggio cambia la struttura del payload (ad esempio ping non possiede alcun payload)

**PongPayload**: contiene l’indirizzo IP e la porta del peer che ha risposto al ping.

**QueryPayload**: specifica la velocità minima richiesta per il peer cercato e la stringa della query da cercare all'interno del file system simulato.

**query\_hit\_payload:** viene utilizzata per inviare le risposte a una query. contiene il numero di risultati trovati (n\_hits), l’indirizzo IP e la porta del peer che risponde, la sua velocità (speed), e un array di file\_result che rappresentano i file trovati. Ogni **file\_result** include un indice numerico e il nome del file, simulando un semplice file system. La dimensione massima dell’array è fissata da MAX\_RESULTS, per evitare la trasmissione di messaggi troppo grandi.

**Codice**:

Il punto di ingresso del programma è la funzione main(), che si occupa di:

* Vengono inizializzati i dati, il semplice file system interno e creato un socket TCP. Il socket viene associato a una porta specifica tramite bind() e posto in ascolto con listen().
* **Connessione al well-known host** il programma tenta di connettersi a un "well-known host", ovvero un peer già presente nella rete. Questa fase permette di entrare nel sistema P2P già esistente e iniziare lo scambio di messaggi. Se dovesse fallire è sempre possibile in seguito collegarsi con un peer in qualsiasi momento.
* **Gestione concorrente con select()** Per gestire contemporaneamente più connessioni e l'interazione con l'utente ho utilizzato la funzione select(). Questo approccio consente di ottenere uno pseudo-parallelismo senza dover ricorrere a fork e processi multipli. Oltre ai socket, anche lo **standard input (stdin)** viene incluso nei descrittori monitorati, così da consentire l'inserimento di comandi da parte dell'utente durante l'esecuzione. FD\_SET(STDIN\_FILENO, &readFDSET);
* **Gestione dei peer connessi:** Per evitare problemi legati al numero massimo di connessioni simultanee, sono stati mantenuti due array distinti: uno per i peer **che si sono connessi a me** (connessioni in ingresso), uno per i peer **a cui mi sono connesso io** (connessioni in uscita).

Questo schema previene situazioni in cui, a causa del numero limitato di connessioni, un nuovo peer non riesca ad entrare nella rete (ad esempio: in una rete con limite 3 connessioni per peer e 4 nodi totali, se tutti si connettono reciprocamente nei primi tre, il quarto resta isolato). L’adozione di due limiti distinti per i due tipi di connessione rende il comportamento della rete più bilanciato. Un’alternativa sarebbe stata la gestione dinamica tramite malloc() per ogni nuova connessione, ma si è preferito imporre un limite massimo per favorire l’efficienza del protocollo di **flooding** (in linea con quanto fatto dal protocollo Gnutella

Dopo la fase iniziale di configurazione, il client entra in un ciclo while all'interno del quale si svolge la parte operativa del client. All'inizio di ogni iterazione, il set di descrittori monitorati viene copiato in un set temporaneo di appoggio. Questo passaggio è necessario perchè la chiamata a select() modifica direttamente il set fornito, impostando a 1 solo i descrittori pronti per la lettura. La copia del set originale permette quindi di mantenere l'elenco completo dei descrittori da monitorare tra un'iterazione e l'altra.

memcpy(&temp, &readFDSET, sizeof(temp));

n= select(maxfd+1, &temp, NULL, NULL, &time)

Dopo la chiamata alla funzione select(), il programma verifica quali descrittori risultano pronti e distingue il tipo di evento a seconda del descrittore attivato.

Se viene rilevata l’attività sul descrittore corrispondente allo standard input (stdin), significa che l'utente ha inserito un comando da terminale. In tal caso, viene letta la scelta dell’utente viene eseguita l’operazione corrispondente: invio di un messaggio di tipo ping, invio di una query per la ricerca di file, connessione a un nuovo peer, visualizzazione dei peer attualmente connessi, disconnessione e uscita dal programma oppure chiusura della connessione con un peer specificato. Per identificare i peer, il programma utilizza unicamente la porta remota, in quanto, per semplicità, tutti i peer risiedono sull’indirizzo di loopback 127.0.0.1, definito tramite macro LOCALHOST. Questa scelta semplifica lo sviluppo e i test locali, ma non limita l’estensibilità del codice: per supportare indirizzi IP arbitrari, sarebbe sufficiente richiedere all’utente anche l’IP del peer al momento della connessione o disconnessione e sostituire la costante LOCALHOST con l’indirizzo fornito. Le funzioni prevedono già il passaggio dell’indirizzo.

Dopo viene verificato se la socket sulla quale il programma è in ascolto risulta attiva, ovvero se ha ricevuto una richiesta di connessione da parte di un altro peer. In tal caso, il programma scorre l’array incoming\_peers alla ricerca di una posizione libera (ovvero non attiva), alla quale poter assegnare la nuova connessione. Se viene trovata una posizione disponibile, la connessione viene accettata tramite la accept(), il relativo socket restituito viene salvato nella struttura del peer, il flag di attività viene impostato, e infine il nuovo descrittore viene aggiunto al readFDSET per poter essere monitorato nelle successivamente tramite select(). Nel caso in cui tutte le posizioni dell’array risultino occupate, viene segnalata l’impossibilità di accettare ulteriori connessioni, notificando un errore

Se il socket attivo rilevato da select() corrisponde a uno dei socket delle connessioni in ingresso, allora viene effettuata una recv() per leggere il primo messaggio, che secondo il protocollo implementato è sempre l’header. Se la recv() restituisce il valore 0, ciò indica che il peer remoto ha chiuso la connessione: in tal caso, il peer viene rimosso dalla lista segnandolo come non attivo e il relativo descrittore viene eliminato dal set di descrittori monitorati. In caso contrario, se la ricezione è andata a buon fine, viene analizzato il campo type dell’header, che specifica la tipologia del messaggio ricevuto, e in base a questo vengono eseguite le operazioni appropriate. Le operazioni specifiche associate a ciascun tipo di messaggio sono descritte nei paragrafi successivi relativi alle funzioni di gestione. Lo stesso identico procedimento viene applicato anche ai socket delle connessioni in uscita.

Questo controllo viene eseguito ciclicamente all’interno del while principale, costituendo di fatto il cuore operativo del client. Grazie all’uso di select(), è possibile gestire contemporaneamente l’interazione con l’utente e le comunicazioni in ingresso e in uscita con altri peer, garantendo una gestione reattiva ed efficiente delle connessioni.

La funzione ping il cui prototipo è int ping(Peer\* outgoing\_peers, Peer\*incoming\_peers, RoutingEntry\* routingTable) viene invocata quando l’utente seleziona l’opzione di inviare un ping. La funzione costruisce un header da inviare a tutti i peer attivi, popolando i campi ttl, payload\_length, type e id. Il campo id deve essere univoco per permettere l’identificazione del messaggio lungo la rete e prevenire l’elaborazione duplicata. A tal fine, ho scelto di utilizzare la funzione rand() per generare un identificatore casuale: pur non garantendo l’unicità globale, questa scelta rappresenta un compromesso accettabile tra semplicità e affidabilità, considerata la scala del progetto. In contesti reali, si potrebbero adottare librerie specializzate per la generazione di UUID (come ad esempio libuuid) o usare una combinazione di valori univoci come indirizzo IP, porta e una componente casuale per ridurre drasticamente il rischio di collisioni. Dopo aver generato l’ID, la funzione cerca una posizione libera nella routingTable, dove verrà registrata la coppia (ID, socket). Se la tabella è piena, viene segnalato un errore. Successivamente, si itera su tutti i peer alla ricerca di connessioni attive a cui inoltrare l’header del messaggio di ping, che non prevede alcun payload. Infine, se almeno un peer è stato trovato e contattato, l’ID viene inserito nella routing table con socket = -1 per indicare che la richiesta è partita da questo nodo, e viene restituito il numero di peer a cui è stato effettivamente inviato il messaggio.

La funzione rispondiPing con prototipo è int rispondiPing (int sd, RoutingEntry\* routingTable, MessageHeader\* header, Peer\* outgoing\_peers, Peer\* incoming\_peers, struct sockaddr\_in peer\_addr) rappresenta l’handler invocato quando il nodo riceve un messaggio di tipo *PING* da un altro peer. Il suo compito principale è duplice: rispondere al mittente con un messaggio di tipo *PONG* e inoltrare il ping ricevuto a tutti gli altri peer a cui è connesso, ad eccezione del peer da cui ha ricevuto il messaggio. La funzione inizia controllando se l’id del messaggio è già presente nella routingTable. Se l’identificatore è già noto, il messaggio viene ignorato (quindi non inoltrato) e viene eventualmente segnalato a terminale per fini di debug o logging. Se, invece, è la prima volta che si riceve quel ping, viene generata una risposta *PONG*: l’header viene riempito in modo appropriato, mantenendo lo stesso id per permettere il tracciamento a ritroso della risposta fino al mittente originario. Il payload del *PONG* include l’indirizzo IP e la porta su cui il nodo è in ascolto, così che l'origine del *PONG* possa essere identificata. Dopo aver inviato la risposta, l’id viene registrato nella routingTable associandolo al socket da cui è stato ricevuto il messaggio. Successivamente, il campo ttl (Time-To-Live) del messaggio viene decrementato. Se il valore risultante è ancora positivo, il messaggio viene inoltrato a tutti i peer connessi (sia in entrata sia in uscita), ad eccezione del peer da cui il ping è stato ricevuto. In questo modo, si garantisce sia la propagazione controllata del messaggio sia il corretto instradamento delle risposte *PONG*.

La funzione handlePong di prototipo int handlePong(int sd, MessageHeader\* header, RoutingEntry\* routingTable) si occupa di inoltrare il messaggio PONG al peer che aveva originato la richiesta PING corrispondente. Inizialmente, la funzione verifica se nella routingTable è presente una entry relativa all'ID del messaggio PONG ricevuto. Se l'ID non è stato trovato, significa che il messaggio non è stato richiesto o è già stato gestito, quindi si segnala un errore e non si inoltra nulla. Se l'ID è presente, si ottiene l'indice corrispondente nella routing table. A questo punto si riceve anche il payload del messaggio, che contiene l'indirizzo IP e la porta del peer che ha risposto. Se il socketDescriptor associato all’entry è uguale a -1, significa che siamo stati noi a generare la richiesta PING originale: in tal caso, il contenuto del payload viene stampato a schermo per permettere all’utente di visualizzare l’identità del peer che ha risposto e, se desiderato, stabilire una connessione. Se invece il socketDescriptor è diverso da -1, si tratta di un PONG che va inoltrato a un altro peer (il mittente del PING originario): pertanto si inviano sia l’header che il payload al socket specificato nella routing table, completando così il meccanismo di risposta a ritroso del protocollo

La funzione Query con prototipo int query(Peer\* outgoing\_peers, Peer\* incoming\_peers, RoutingEntry\* routingTable) viene invocata quando l’utente desidera effettuare una ricerca nella rete. La funzione inizia cercando uno spazio libero nella routingTable: se non è disponibile alcuno slot, restituisce un errore. Successivamente, richiede all’utente di inserire la velocità minima di trasmissione desiderata e il nome del file da cercare. Tali informazioni vengono utilizzate per costruire l’header e il payload del messaggio QUERY: l’header viene popolato nello stesso modo della funzione ping, con la sola differenza che il campo type viene impostato a TYPE\_QUERY, mentre il payload contiene il nome del file e la velocità richiesta. Il messaggio viene quindi inoltrato a tutti i peer attualmente connessi, sia in uscita che in ingresso. Se la query viene effettivamente inviata ad almeno un peer, viene aggiunta una nuova entry nella routingTable, con il socketDescriptor impostato a -1 per indicare che si tratta di una richiesta originata localmente. Infine, la funzione restituisce il numero di peer a cui la query è stata inoltrata

La funzione handleQuery di prototipo int handleQuery(int sd, MessageHeader\* header, Peer\* outgoing\_peers, Peer\* incoming\_peers, RoutingEntry\* routingTable, int mySpeed, struct sockaddr\_in bind\_ip\_port, int listenPort, char fs[NMAXFILE][MAXLEN]) viene invocata alla ricezione di un messaggio di tipo QUERY. Essa ha una duplice responsabilità: da un lato verifica se il file richiesto è presente nel proprio file system simulato; dall’altro si occupa dell’eventuale inoltro della query agli altri peer connessi.

La funzione inizia controllando se la query è già presente nella routingTable, per evitare di processare messaggi duplicati. Successivamente riceve il payload del messaggio QUERY, e se la propria velocità (mySpeed) soddisfa il requisito minimo indicato nella richiesta, procede a cercare i file corrispondenti nel proprio file system simulato (fs).

Per ogni file presente, si utilizza la funzione strstr() per verificare se il nome contiene come sottostringa la query ricevuta. In caso positivo, si aggiunge un risultato nel campo results del payload di tipo QUERYHIT, popolando i campi name (strncpy(queryHitPayload.results[queryHitPayload.n\_hits].name, fs[i], MAX\_FILENAME)) con il nome del file, index con l’indice corrente del ciclo (queryHitPayload.results[queryHitPayload.n\_hits].index = htons(i)), e incrementando n\_hits. Se il numero di risultati trovati raggiunge MAX\_RESULTS, la ricerca termina anticipatamente.

Se sono stati trovati uno o più risultati (n\_hits > 0), si costruisce un messaggio di tipo QUERYHIT, compilando opportunamente l’header (con lo stesso msg\_id ricevuto) e inviandolo direttamente al peer che ha originato la richiesta (identificato da sd), includendo la propria velocità, IP e porta.

Immagine che contiene testo, schermata, software

Il contenuto generato dall'IA potrebbe non essere corretto.

Infine, la funzione valuta se inoltrare la query ricevuta ad altri peer: se c’è spazio nella routingTable, il TTL del messaggio viene decrementato, e se resta maggiore di 0, la query viene inoltrata a tutti i peer connessi (in ingresso e in uscita). Se viene effettivamente inoltrata ad almeno un peer, viene aggiunta una nuova entry nella routingTable.

La funzione handleQueryHit di prototipo int handleQueryHit(int sd, MessageHeader\* header, RoutingEntry\* routingTable). si occupa di gestire i messaggi di tipo QUERYHIT, ovvero le risposte alle richieste di ricerca inoltrate nella rete. Quando viene ricevuto un messaggio di risposta, la funzione estrae il payload e cerca nella routingTable l’entry associata all’ID del messaggio ricevuto, per determinare il socket verso cui inoltrare la risposta. Se non viene trovata alcuna entry, la funzione restituisce un errore. Se invece la ricerca ha successo, si considerano due casi distinti. Nel caso in cui il valore associato nella routing table sia -1, significa che la query originaria è stata inviata da questo nodo: in tal caso, la funzione stampa i risultati trovati contenuti nel payload, mostrando per ciascun risultato il nome del file e l’indice corrispondente, e termina con successo. Se invece il socket indicato non è -1, la funzione inoltra il messaggio (sia header che payload) al peer indicato, proseguendo così il meccanismo di routing delle risposte nella rete P2P. In questo modo, la funzione consente di gestire correttamente sia i messaggi diretti a questo nodo, sia quelli che devono essere inoltrati ad altri peer.

Funzionamento:

All’avvio del programma, viene inizialmente richiesto all’utente di specificare la porta su cui il nodo dovrà mettersi in ascolto. Una volta ricevuto questo valore, il programma procede con la fase di bind, associando il socket alla porta specificata, e successivamente esegue la listen per accettare eventuali connessioni in ingresso. Dopo essersi configurato come nodo in ascolto, il programma richiede all’utente di popolare il file system simulato, inserendo i nomi dei file che saranno disponibili localmente per eventuali ricerche effettuate dagli altri peer nella rete. In questo modo, ogni nodo è inizializzato con una propria lista di file condivisibili, che sarà poi consultata durante la gestione delle query.

Immagine che contiene testo, software, computer, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.

Successivamente, viene richiesto all’utente di inserire la porta del well-known peer a cui connettersi, permettendo così di entrare a far parte della rete P2P. Se la connessione ha esito positivo, viene stampato un messaggio di conferma che segnala l’avvenuto collegamento con il nodo remoto. A questo punto, il programma presenta all’utente un menu con le varie operazioni disponibili, tra cui l’invio di ping, l’effettuazione di query, la visualizzazione dei peer connessi e l’uscita dalla rete

Immagine che contiene testo, elettronica, schermata, computer

Il contenuto generato dall'IA potrebbe non essere corretto.

Se, ad esempio, l’utente seleziona l’opzione **1** dal menu, viene eseguito l’invio di un messaggio di **ping** ai peer connessi. Dopo l’invio, il menu viene ristampato e il programma entra in attesa di nuovi eventi attraverso l’uso della select, che permette di gestire simultaneamente input da tastiera e messaggi di rete. In questo scenario, il client riceve risposte da due nodi presenti nella rete: entrambe le risposte attraversano il solo peer a cui il client è direttamente connesso, dimostrando così il meccanismo di forwarding tipico delle reti P2P.

Immagine che contiene testo, schermata, computer, software

Il contenuto generato dall'IA potrebbe non essere corretto.

Se decido di fare una **query**, il programma richiede all’utente di inserire due informazioni: il **nome del file** da cercare e la **velocità minima di trasmissione** desiderata. Immagine che contiene testo, schermata, software, schermo

Il contenuto generato dall'IA potrebbe non essere corretto.

Ottengo quindi dei risultati in risposta alla query: come si può vedere, viene prima stampato **il peer che ha inoltrato la risposta** (cioè il nodo che ha fatto da tramite), seguito da **chi ha effettivamente generato la risposta** (ovvero il peer che possedeva il file nel proprio file system simulato). Infine, vengono **stampati i risultati trovati**, ciascuno con il **nome del file** e il relativo **indice** nel file system di quel peer

Immagine che contiene testo, elettronica, computer, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

Posso anche connettermi a un altro peer già attivo nella rete, ad esempio quello in ascolto sulla porta **4444**. Una volta stabilita la connessione, il peer conferma l’avvenuto collegamento e aggiorna le sue strutture dati

Immagine che contiene testo, schermata, software, computer

Il contenuto generato dall'IA potrebbe non essere corretto.

Digitando **4** dal menu, posso stampare a schermo la lista dei peer attualmente connessi. Immagine che contiene testo, schermata, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.

Ora posso decidere di chiudere la connessione con uno dei peer, ad esempio quello connesso alla porta **2222**.

Immagine che contiene testo, schermata, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.

E infine digitando 5 posso terminare il programma.

Immagine che contiene testo, schermata, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.