Sviluppo di un Sistema di Chat Decentralizzato Peer-to-Peer con Gestione Dinamica del Server

Giorgia Roselli* - Kevin Attarantato*, Claudio Antares Mezzina [1]

Sommario
Questo progetto, realizzato per l'esame di *Sistemi Distribuiti* presso *l'Università degli Studi di Urbino Carlo Bo*, si concentra sull'implementazione di un sistema di chat *Peer-to-Peer* (P2P) resiliente e decentralizzato.
L'obiettivo principale del progetto è permettere la comunicazione tra più utenti senza fare affidamento su un server centrale permanente, tramite una rete distribuita in cui ciascun nodo può agire sia come client che, in caso di necessità, come server.
Ogni nodo si connette inizialmente a un server attivo per inviare e ricevere messaggi, ma è in grado di gestire autonomamente la riconnessione e la riorganizzazione della rete nel caso in cui il server si disconnetta. In tal caso, un meccanismo di elezione deterministica seleziona automaticamente un nuovo nodo server, garantendo la continuità del servizio e l'autonomia della rete.
Il sistema è implementato in linguaggio *Python* e utilizza la libreria standard 'Socket' per la comunicazione TCP/IP tra nodi. L'architettura del progetto è modulare e comprende la gestione dei thread per il parallelismo nella ricezione e trasmissione dei messaggi, l'aggiornamento della lista dei peer connessi e la gestione dello stato interno del nodo (client/server).
Il progetto si propone di esplorare i concetti fondamentali delle architetture distribuite, come la gestione decentralizzata delle connessioni, la tolleranza ai guasti e la sincronizzazione tra peer, offrendo un ambiente di chat robusto, scalabile e autonomo.

Keywords
Socket - Chat - Peer-to-Peer - Thread

[1] *Docente di Sistemi DIstribuiti, Università degli Studi di Urbino Carlo Bo, Urbino, Italia*
***Corresponding author**: g.roselli1@campus.uniurb.it - k.attarantato@campus.uniurb.it

Introduzione

Nel panorama odierno delle comunicazioni digitali, l'affidabilità e la resilienza delle reti rappresentano requisiti fondamentali per lo sviluppo di applicazioni distribuite. I sistemi centralizzati, seppur ampiamente utilizzati, presentano diversi limiti strutturali, tra cui la dipendenza da un singolo punto di accesso e la vulnerabilità a guasti del nodo centrale. Tali caratteristiche risultano particolarmente critiche in contesti in cui la continuità del servizio e la tolleranza ai guasti sono essenziali. Per rispondere a queste esigenze, le architetture *peer-to-peer* (P2P) emergono come una valida alternativa, grazie alla loro capacità di distribuire responsabilità e risorse tra tutti i nodi partecipanti alla rete.

L'approccio P2P consente a ciascun nodo di comportarsi sia da client che da server, favorendo la decentralizzazione e migliorando la scalabilità del sistema. In questo contesto, il presente progetto si propone di sviluppare una chat distribuita in grado di funzionare autonomamente anche in assenza di un nodo server permanente. Il sistema è progettato per adattarsi dinamicamente ai cambiamenti della rete: se il nodo server si disconnette, viene automaticamente avviata una procedura di elezione che seleziona un nuovo *nodo leader*, garantendo la continuità della comunicazione.

Obiettivo del Progetto

L'obiettivo principale di questo progetto è la realizzazione di una chat P2P completamente funzionante, capace di:

- Stabilire connessioni tra nodi in modalità client-server su rete locale tramite socket TCP;
- Permettere l'invio e la ricezione simultanea di messaggi mediante l'utilizzo di thread;
- Gestire dinamicamente la lista dei peer connessi e lo stato del sistema;
- Eseguire una procedura di elezione deterministica alla disconnessione del server, eleggendo un nuovo nodo leader;
- Garantire la riconnessione automatica dei client al nuovo server eletto.

Architettura del Sistema

Il sistema è implementato in linguaggio *Python*, utilizzando la libreria standard *socket* per la gestione delle comunicazioni di rete e il modulo *threading* per il supporto al parallelismo. La struttura del progetto è suddivisa in più componenti logici:

- Un modulo centrale *ChatNode* che gestisce lo stato del nodo, sia in modalità client che server;

- Un sottosistema di gestione dei messaggi, che si occupa della ricezione, invio e broadcasting delle comunicazioni;
- Un modulo di coordinamento per l'elezione del nodo leader in caso di disconnessione del server;
- Meccanismi di aggiornamento e sincronizzazione della *peer list* tra i nodi.

L'efficacia di questa architettura si fonda sull'impiego sinergico di due componenti tecniche fondamentali: i *socket*, utilizzati per la comunicazione tra nodi su rete TCP/IP, e i *thread*, indispensabili per gestire la concorrenza delle operazioni in modo asincrono.

Concetti Fondamentali: Thread e Socket

Nel progetto, due elementi tecnici ricoprono un ruolo cruciale per il corretto funzionamento della comunicazione tra nodi: i *thread* e i *socket*.

Thread Un *thread* è il più piccolo flusso di esecuzione di un programma. In questo sistema, i thread permettono al nodo di eseguire più operazioni contemporaneamente, come ricevere messaggi in ingresso e, allo stesso tempo, accettare nuove connessioni o inviare messaggi in uscita.

Grazie all'utilizzo del modulo *threading* di Python, è possibile ottenere un comportamento concorrente, migliorando la reattività del sistema e la gestione parallela delle connessioni.

Socket Un *socket* è un'interfaccia software che consente a due processi, anche su macchine diverse, di comunicare tra loro attraverso una rete.

In particolare, un socket rappresenta un endpoint di una connessione bidirezionale basata su protocollo TCP/IP.

Ogni socket è associato a un indirizzo IP e a una porta, che identificano univocamente il punto di comunicazione all'interno della rete.

Nel progetto, i socket vengono utilizzati sia per accettare connessioni (lato server), sia per instaurare una connessione a un nodo remoto (lato client).

La libreria *socket* di Python fornisce un'interfaccia di basso livello che permette di:

- Creare socket TCP attraverso la specifica *AF_INET* e *SOCK_STREAM*;
- Collegare il socket a un indirizzo e porta con *bind()*;
- Accettare nuove connessioni in ingresso tramite *listen()* e *accept()*;
- Stabilire connessioni verso altri nodi usando *connect()*;
- Inviare e ricevere dati attraverso i metodi *send()* e *recv()*.

Rete TCP/IP Il protocollo TCP/IP (Transmission Control Protocol / Internet Protocol) è il fondamento della comunicazione su reti moderne, incluse Internet e le reti locali. Esso

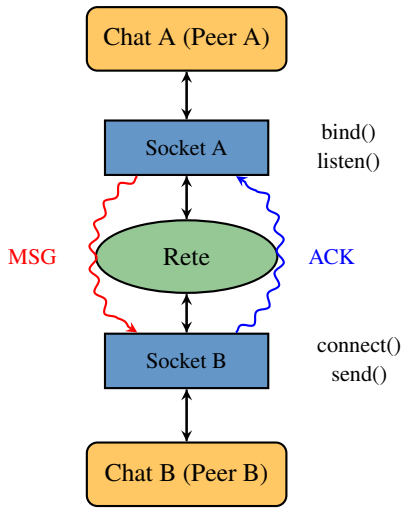

Figura 1. Socket in chat P2P

definisce un insieme di protocolli gerarchici che regolano come i dati vengono suddivisi, indirizzati, trasmessi, instradati e ricevuti tra dispositivi connessi.

Nel contesto del progetto, il protocollo *TCP* viene utilizzato per garantire una comunicazione affidabile tra nodi: esso assicura che i pacchetti arrivino nell'ordine corretto, senza perdite o duplicazioni.

Il protocollo *IP*, invece, si occupa dell'instradamento dei pacchetti attraverso la rete, utilizzando indirizzi IP per identificare in modo univoco i dispositivi.

L'unione di questi due protocolli consente ai socket di stabilire connessioni robuste e ordinate, rendendo possibile la comunicazione tra peer anche in ambienti distribuiti complessi.

Abbiamo scelto di implementare il protocollo TCP/IP, e in particolare TCP, per le sue caratteristiche di affidabilità e controllo dell'ordine dei messaggi, fondamentali per un sistema di chat distribuito. Altri protocolli, come UDP, offrono una trasmissione più veloce ma non garantiscono la consegna dei pacchetti né l'ordine corretto, risultando quindi inadatti a un contesto in cui ogni messaggio deve essere ricevuto interamente e nella sequenza prevista. TCP, invece, assicura un canale di comunicazione persistente e controllato, che si adatta perfettamente agli obiettivi di stabilità e correttezza del nostro progetto.

Questa infrastruttura consente la trasmissione di messaggi tra peer in modo affidabile, garantendo il mantenimento dell'ordine e l'integrità dei dati, requisiti fondamentali per la comunicazione in un sistema distribuito.

L'integrazione di thread e socket costituisce quindi la base tecnica per la gestione efficace delle connessioni e del flusso di messaggi tra peer. Tuttavia, in un contesto distribuito e privo di un'infrastruttura centralizzata permanente, risulta fondamentale implementare anche strategie di tolleranza ai guasti, in grado di garantire la continuità del servizio in caso di disconnessione del nodo server.

Comunicazione e Tolleranza ai Guasti

Un aspetto cruciale del progetto è la gestione della tolleranza ai guasti: nel momento in cui il nodo server viene disconnesso, viene innescato un meccanismo di elezione deterministica basato su parametri condivisi (es. timestamp di connessione) per identificare il nuovo nodo leader. Una volta eletto, quest'ultimo assume il ruolo di server e attende le riconnessioni da parte degli altri peer.

1. Architettura e Funzionamento del Sistema

Lo sviluppo della chat peer-to-peer è stato condotto adottando un approccio incrementale e modulare, con l'obiettivo di progettare un sistema distribuito capace di garantire comunicazione affidabile tra nodi anche in scenari di disconnessione del server. La pipeline di sviluppo è stata suddivisa in fasi ben distinte, ognuna delle quali ha affrontato specifici aspetti legati alla gestione della rete, alla trasmissione dei messaggi, alla tolleranza ai guasti e al coordinamento tra peer.

Le fasi principali del progetto sono state:

- Progettazione dell'architettura logica della rete peer-to-peer;
- Gestione delle connessioni tramite socket TCP;
- Implementazione della concorrenza attraverso thread;
- Meccanismo di elezione per la tolleranza ai guasti;
- Scambio dei messaggi e aggiornamento della lista dei peer.

Questo schema di lavoro ha permesso di sviluppare un sistema flessibile, in cui ogni nodo può assumere dinamicamente il ruolo di client o di server, mantenendo costantemente aggiornate le informazioni sulla rete.

1.1 Architettura Peer-to-Peer

La chat è strutturata secondo un'architettura decentralizzata in cui ogni nodo può fungere sia da server che da client, in base al contesto operativo.

All'avvio, un nodo si collega a un nodo già attivo, inizialmente designato come server, che mantiene una lista aggiornata dei peer connessi. Ogni nodo client invia al server le proprie informazioni identificative (nome utente, timestamp di connessione, indirizzo IP e porta) per essere integrato nella rete.

Il server si occupa di accettare nuove connessioni in ingresso e di inoltrare a tutti i peer aggiornamenti in tempo reale, relativi alla struttura della rete. Quando il server si disconnette, i peer rimanenti attivano un meccanismo di elezione per selezionare un nuovo nodo leader.

1.2 Gestione delle Connessioni

La comunicazione tra i nodi avviene tramite socket TCP, utilizzando la libreria standard *socket* di Python.

Ogni nodo, in modalità client, stabilisce una connessione verso un nodo server tramite un socket dedicato. Quando un nodo assume il ruolo di server, utilizza un socket in ascolto per accettare nuove connessioni da altri peer.

Per ogni nuova connessione accettata dal server, si ha un nuovo socket dedicato alla comunicazione con quel client (restituto dal metodo *accept()*) e verrà generato un thread dedicato alla gestione delle comunicazioni con il relativo client.

Il server si occupa di accettare nuove connessioni in ingresso e di inoltrare a tutti i peer aggiornamenti in tempo reale, relativi alla struttura della rete. Ogni messaggio ricevuto viene deserializzato (formato JSON) e processato in base alla sua tipologia (messaggio di chat, aggiornamento della rete, elezione, disconnessione, ecc.).

1.3 Concorrenza tramite Thread

Per consentire l'esecuzione simultanea di più operazioni, ogni nodo utilizza thread dedicati per:

- Accettare nuove connessioni (lato server);
- Ascoltare messaggi in arrivo (lato client/server);
- Eseguire il flusso di input/output dell'utente.

L'uso dei thread è risultato fondamentale per evitare blocchi nel flusso di esecuzione, permettendo a ciascun nodo di rimanere responsivo anche durante operazioni di I/O di rete o attese bloccanti.

Nel caso del server, i messaggi ricevuti da un client vengono gestiti e inoltrati agli altri peer direttamente dal thread associato a quel client, senza bisogno di thread di invio separati.

1.4 Gestione della Peer List

Ogni nodo mantiene una lista locale dei peer connessi, che contiene informazioni essenziali per il corretto funzionamento della rete: nome utente, indirizzo IP, porta e timestamp di connessione. La lista viene inizialmente ottenuta dal server e successivamente aggiornata in modo dinamico, a seguito di nuovi ingressi o disconnessioni.

Il server ha il compito di tenere sincronizzate le peer list di tutti i client inviando aggiornamenti ogni volta che cambia la composizione della rete. I messaggi di aggiornamento vengono propagati tramite broadcast a tutti i peer.

1.5 Meccanismo di Elezione

Quando un nodo rileva la disconnessione del server (es. per chiusura del socket o timeout), attiva un meccanismo di elezione per selezionare un nuovo nodo leader. La procedura è deterministica e si basa su un criterio di priorità, tipicamente il timestamp di connessione: il nodo con il valore più basso, ovvero quello che si è connesso per primo al server, viene eletto come nuovo server.

La procedura è progettata per evitare condizioni di race conditions, ovvero quando due o più nodi eseguono operazioni concorrenti senza un adeguato coordinamento producendo

così risultati indesiderati o incoerenti, e promozioni multiple. Per prevenire questo ogni nodo attende un tempo proporzionale alla propria posizione nella lista ordinata dei candidati prima di tentare la promozione. In pratica, il nodo che si è connesso per primo attenderà meno, mentre gli altri attenderanno progressivamente di più.

Durante questo intervallo, ciascun nodo verifica se un altro peer si è già promosso a server e, in tal caso, interrompe immediatamente la propria procedura di promozione, evitando conflitti. Questo approccio riduce significativamente la possibilità che due o più peer diventino server contemporaneamente.

1.6 Formato dei Messaggi

La comunicazione tra nodi avviene attraverso messaggi serializzati in formato JSON, che includono un campo *type* per distinguere le diverse tipologie (messaggio utente, aggiornamento, promozione, ecc.). Ogni messaggio contiene anche eventuali payload specifici, come il contenuto del messaggio di chat o le informazioni sui peer.

Esempio di messaggio:

```
{
  "type": "chat_message",
  "username": "Alice",
  "message": "Ciao a tutti!",
  "timestamp": "2025-06-28 10:45:12"
}
```

Questa struttura permette una facile estendibilità e una gestione uniforme lato client e server.

1.7 Gestione degli Errori e Disconnessioni

Il sistema è progettato per tollerare disconnessioni temporanee o improvvise.

In caso di perdita della connessione al server, ogni nodo tenta una riconnessione automatica al nuovo nodo leader. Il server, rileva automaticamente la chiusura dei socket client e aggiorna la lista dei peer, propagando le modifiche ai nodi rimanenti.

Le eccezioni dovute a errori di rete, invii falliti o dati corrotti sono gestite con try-except, evitando crash improvvisi e mantenendo il sistema stabile anche in condizioni di rete non ideali.

1.8 Organizzazione del Codice

Il codice del progetto è stato suddiviso in moduli funzionali per migliorare la leggibilità e facilitare la manutenzione:

- *chat_node.py*: classe principale del nodo, con logica client/server.
- *server_mode.py* e *client_mode.py*: flussi separati per le due modalità operative.
- *helpers.py*: funzioni di utilità come l'ottenimento del timestamp.
- *constants.py*: costanti condivise come buffer size, porta e host predefiniti.

Questa separazione ha permesso una migliore organizzazione dello sviluppo e una gestione modulare delle funzionalità.

2. Flusso di Avvio e Utilizzo della Chat

Di seguito viene illustrata l'esecuzione tipica del sistema di chat, sia in modalità server che client, con esempi pratici del comportamento visualizzato nel terminale.

1. Avvio del programma

All'avvio, l'utente sceglie come prima cosa il nome che verrà poi mostrato nella chat:

```
====================================
        CHAT PEER-TO-PEER
====================================

Il tuo nome utente:
```

Il nome dovrà essere univoco in tutta la chat. In caso di nome doppio verrà data l'opportunità all'utente di poter cambiare nome o di uscire tramite il seguente messaggio:

```
Nome utente gia' in uso!
Vuoi riprovare con un altro nome utente? (s/N):
```

2. Scelta della Modalità: Server o Client

Successivamente, il programma richiede all'utente di scegliere se avviare il nodo in modalità **server** o **client**:

```
Come vuoi partecipare?
1. Avvia una nuova chat (diventerai il server)
2. Unisciti a una chat esistente (diventerai client)
Scelta (1/2):
```

Se l'utente seleziona "*1*", il nodo viene avviato come server, mettendosi in ascolto su una porta TCP predefinita:

```
Server 'Bob' in ascolto su localhost:12345
Massimo 5 client consentiti
```

3. Connessione di un Nodo Client

Un altro utente può avviare il programma in modalità client. Gli verrà chiesto di specificare l'indirizzo IP e la porta del server al quale connettersi:

```
Indirizzo server (default localhost):
Porta server (default 12345):
Tentativo di connessione a localhost:12345...
CONNESSO AL SERVER
```

Dopo la connessione, il client può iniziare a partecipare attivamente alla chat.
Se il nodo è avviato in modalità server, è possibile visualizzare in qualsiasi momento la lista dei peer attualmente connessi digitando il comando *list*.
Questo comando **non** è disponibile in modalità client.

```
Comandi disponibili:
  list    - Mostra client connessi
  quit    - Chiudi server
  <testo> - Invia messaggio a tutti
list
Client connessi:
  • Bob (127.0.0.1:51010)
  • Alice (127.0.0.1:51027)
  • Carl (127.0.0.1:51045)
```

Il numero massimo di client accettati è limitato a 5. Eventuali richieste di connessione oltre questo limite verranno rifiutate e i relativi peer non appariranno nella lista, in quanto non entrati mai a far parte della rete.

4. Invio e Ricezione dei Messaggi

Ogni peer può ora inviare messaggi che vengono automaticamente propagati a tutti i peer connessi, tramite il server:

```
Hai scritto: Ciao a tutti!
[22:07:23] Bob ha scritto: Ciao Alice!
```

5. Disconnessione del Server e Elezione Automatica

Se il server si disconnette (per chiusura o crash), i client rilevano l'evento e avviano automaticamente la procedura di elezione:

```
Server disconnesso!
Server disconnesso, avvio procedura di elezione...
Avvio elezione - Il mio ID: (0, 1751314213.2440398)
Aspetto 1.00 secondi per l'elezione...
Sono stato eletto come nuovo server!
Avvio promozione a server...
SERVER AVVIATO
Server 'Alice' in ascolto su localhost:12345
```

6. Riconnessione dei Peer al Nuovo Server

Gli altri client che non sono stati eletti server si riconnettono automaticamente al nuovo nodo leader:

```
Server disconnesso!
Server disconnesso, avvio procedura di elezione...
Avvio elezione - Il mio ID: (1, 1751314228.675849)
Aspetto 2.00 secondi per l'elezione...
Alice e' stato eletto come nuovo server
Aspetto che il nuovo server si avvii...
Tentativo riconnessione 1/8
CONNESSO AL SERVER
Connesso al server 'Alice' su localhost:12345
```

7. Chiusura del Nodo

Alla chiusura del programma da parte dell'utente, il nodo libera le risorse in modo ordinato e salva l'intera chat in un file .log:

```
Chiusura in corso...
Chat salvata in: <PercorsoLog>.log
Disconnesso.
```

3. Analisi delle Funzionalità Principali

Le funzionalità principali del progetto sono state implementate come metodi della classe *ChatNode*.

Di seguito vengono analizzate le più significative, con una descrizione del loro comportamento e della loro importanza nel contesto distribuito.

`connect_as_client(self, host=DEFAULT_HOST, port=DEFAULT_PORT)`

Tenta di stabilire una connessione con un nodo server specificato tramite host e porta. Esegue un handshake iniziale con invio dei metadati (username e timestamp) e attende la risposta di accettazione o rifiuto. Se la connessione va a buon fine, avvia un thread per la ricezione continua dei messaggi. Gestisce diversi casi d'errore, tra cui server non disponibile, nome utente duplicato o problemi di rete.

`start_as_server(self, host=DEFAULT_HOST, port=DEFAULT_PORT)`

Avvia il nodo in modalità server. Inizializza il socket TCP in ascolto su un host e porta specificati, abilita la ricezione di connessioni multiple e avvia un thread dedicato per accettare nuovi client. Imposta gli stati interni del nodo e fornisce un'interfaccia di avvio user-friendly. È una funzione centrale per l'inizializzazione della rete distribuita.

`broadcast_to_clients(self, message_dict)`

Questa funzione consente al server di inviare un messaggio (sotto forma di dizionario JSON) a tutti i client attualmente connessi. È fondamentale per garantire la coerenza tra i peer, ad esempio per la propagazione di messaggi di chat o aggiornamenti della peer list. In caso di errore o disconnessione di un client, il relativo socket viene rimosso per evitare malfunzionamenti futuri.

`calculate_election_delay(self)`

Determina il ritardo (in secondi) che ogni nodo deve attendere prima di procedere alla propria candidatura come nuovo server. Il ritardo è proporzionale alla posizione del nodo nella lista dei candidati ordinata per `connection_time`, così da garantire che il nodo più "anziano" abbia priorità. Questa strategia riduce drasticamente il rischio di promozioni simultanee e conflitti di leadership.

`promote_to_server(self)`

Promuove il nodo corrente a nuovo server della rete. Questo metodo inizializza un socket TCP in ascolto, avvia i thread per la gestione delle connessioni in ingresso e aggiorna lo stato locale. Inoltre, notifica il resto della rete, consentendo agli altri peer di riconnettersi correttamente. È una componente cruciale per la resilienza del sistema.

`process_server_message(self, message_data)`

Gestisce i messaggi ricevuti dal server. Analizza il campo *type* per distinguere tra messaggi di chat, annunci del server, notifiche di connessione/disconnessione o terminazione della sessione. Ogni tipo di messaggio attiva un'azione specifica, come la stampa a schermo o l'aggiornamento della peer list. Questa funzione agisce come dispatcher centrale lato client.

`handle_server_disconnect(self)`

Viene invocata quando il nodo client rileva la disconnessione del server. Imposta i flag interni per segnare la perdita della connessione, chiude il socket del server e avvia la procedura di elezione per promuovere un nuovo nodo a server. Questa funzione è cruciale per garantire la tolleranza ai guasti e la continuità operativa della rete.

`shutdown(self)`

Gestisce la chiusura controllata del nodo. Chiude i socket aperti, interrompe i thread attivi e invia eventuali notifiche di disconnessione agli altri peer. Questa funzione è progettata per rilasciare correttamente le risorse e mantenere la stabilità del sistema anche in fase di arresto, prevenendo errori nella riconnessione successiva.

4. Protocollo di Comunicazione tra Peer

Il protocollo di comunicazione alla base del sistema si fonda su uno scambio strutturato e continuo di messaggi tra i nodi (peer) connessi. Ogni messaggio viene serializzato in formato JSON e contiene un campo *type* che identifica la tipologia dell'informazione inviata (es. messaggio utente, aggiornamento, elezione, disconnessione, ecc.).

4.1 Flusso dei Messaggi

Il flusso dei messaggi si articola come segue:

- Un nodo client si connette al server e invia un messaggio di identificazione contenente *username*, *IP* (disponibile dal socket stesso), *port* (disponibile dal socket stesso) e *timestamp*;
- Il server aggiorna la propria *peer list* e invia un messaggio di aggiornamento a tutti i client connessi;
- I messaggi di chat vengono inviati da un client al server, che li inoltra a tutti gli altri peer;
- In caso di eventi straordinari (es. disconnessione di un nodo), il server invia un broadcast aggiornato;
- Se il server si disconnette, i client attivano un protocollo di elezione che genera messaggi specifici per il coordinamento.

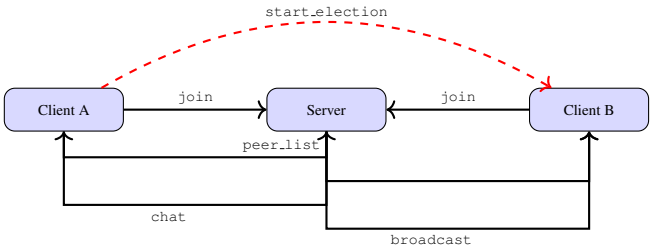

Figura 2. Schema di comunicazione tra peer e server

Il sistema garantisce coerenza tra i peer grazie alla gestione centralizzata della *peer list* da parte del server e alla propagazione degli aggiornamenti tramite messaggi *update*. La sincronizzazione è gestita in tempo reale, assicurando che ogni nodo abbia sempre una visione coerente della rete.

5. Tolleranza ai Guasti e Elezione del Nuovo Server

Un punto centrale dell'architettura è la gestione automatica della disconnessione del server. Il sistema implementa un algoritmo di elezione distribuita per promuovere un nuovo nodo a server senza intervento esterno.

5.1 Algoritmo di Elezione (Pseudocodice)

Algorithm 1: Procedura di Elezione del Nuovo nodo Server

Data: Lista locale dei peer (`self.peer_list`), ordinata per tempo di connessione
Result: Promozione di un solo nodo a server

```
Estrai i peer non server da self.peer_list;
Ordina i peer per connection_time crescente;
Determina la propria posizione nella lista usando lo username;
delay ← base_delay + (my_position × step_delay);
sleep(delay);
if nessun altro nodo è stato promosso nel frattempo then
    Avvia la procedura di promozione a server;
end
```

Ogni nodo calcola un ritardo proporzionale alla propria "anzianità" di connessione. Se durante l'attesa rileva che un altro nodo è già diventato server (es. ricevendo messaggi dal nuovo server), annulla la propria promozione.

5.2 Scenario di Disconnessione

1. Il server viene spento o disconnesso;
2. I client rilevano la chiusura del socket;
3. Ogni peer attiva il proprio timer di elezione.
4. Il primo nodo nella lista (con timestamp minore) scade per primo e si promuove a server;

5. Notifica la promozione agli altri peer;
6. Gli altri annullano la promozione e si riconnettono.

5.3 Limiti della Soluzione

- La promozione avviene solo se la *peer list* è aggiornata;
- Reti con latenze elevate possono causare promozioni multiple temporanee;
- Nessuna autenticazione tra peer: possibilità di elezioni non autorizzate.

6. Salvataggio dei Log di Chat

Un aspetto aggiuntivo del sistema è la possibilità per ogni nodo (sia server che peer) di salvare la cronologia completa della sessione di chat in un file *.log*.

Durante l'intera esecuzione, ogni messaggio trasmesso (sia di tipo utente che di sistema) viene aggiunto a una struttura interna di log. Al termine della sessione, quando il nodo viene chiuso, viene generato automaticamente un file contenente lo storico dei messaggi scambiati.

Il salvataggio dei log è stato implementato tramite tre funzioni principali:

`ensure_log_directory(self)`

Verifica l'esistenza della directory destinata al salvataggio dei file di log. Se la directory non è presente nel sistema, la crea automaticamente utilizzando le funzioni del modulo `os`. Questa funzione garantisce che il percorso di destinazione sia sempre disponibile prima di procedere con il salvataggio.

`add_to_log(self, message_type, username, message, timestamp)`

Aggiunge un nuovo messaggio alla struttura interna di log. La funzione riceve come parametri il tipo di messaggio (`message_type`), l'username dell'autore, il contenuto del messaggio e un timestamp opzionale. Se il timestamp non viene fornito, viene generato automaticamente. Ogni entry viene memorizzata come dizionario contenente tutti i metadati necessari per la ricostruzione cronologica della conversazione.

`save_chat_log(self)`

Genera e salva il file di log finale su disco. La funzione costruisce il nome del file utilizzando l'username del nodo e l'intervallo temporale della sessione. Include la gestione degli errori durante la scrittura e fornisce feedback all'utente sul successo dell'operazione.

Il file di log viene nominato in maniera univoca in base all'username del nodo e all'intervallo temporale della sessione (es. *chat_log_Alice_20250706_100213_to_20250706_112430.log*).

Al suo interno, vengono riportate informazioni dettagliate tra cui:

- **Intestazione**: contiene l'identificativo del nodo (username), l'intervallo temporale completo della sessione, la modalità operativa (SERVER/CLIENT);
- **Informazioni di connessione**: per i nodi client, viene riportato il riferimento al server di destinazione (username, host e porta);
- **Corpo del log**: sequenza cronologica di tutti i messaggi scambiati, con timestamp, autore e contenuto;
- **Differenziazione dei messaggi**: i messaggi vengono formattati diversamente in base al tipo:
  - *chat_message*: messaggi standard degli utenti;
  - *server_message*: messaggi del server con identificazione specifica;
  - *system*: messaggi di sistema per eventi di connessione/disconnessione.

Questa funzionalità permette di mantenere una traccia persistente delle comunicazioni, utile per l'archiviazione locale delle sessioni di chat. Il salvataggio viene eseguito automaticamente alla chiusura del nodo e non richiede alcun intervento manuale da parte dell'utente.

7. Possibili Estensioni e Sviluppi Futuri

Il sistema descritto rappresenta una solida base per l'implementazione di architetture distribuite resilienti, ma apre anche a numerose possibilità di miglioramento ed espansione.

Di seguito vengono analizzate alcune delle estensioni più significative che potrebbero essere sviluppate in futuro per potenziare le funzionalità, la sicurezza e l'usabilità del sistema.

- **Interfaccia Grafica (GUI)**: Attualmente l'interazione avviene esclusivamente tramite terminale. Lo sviluppo di una GUI migliorerebbe significativamente l'esperienza utente, rendendo il sistema più accessibile anche a persone meno esperte di ambienti CLI.

  Una possibilità interessante, anche in ottica professionale, sarebbe l'uso di **React**, sfruttando le competenze già acquisite in ambito lavorativo: la chat attuale potrebbe fungere da back-end, esposto tramite WebSocket o API REST, mentre l'interfaccia utente potrebbe essere sviluppata come una SPA (Single Page Application) moderna, esteticamente gradevole e responsive.

- **Autenticazione e Sicurezza**: L'inserimento di un meccanismo di autenticazione tra peer (basato su token o chiavi pubbliche/private) permetterebbe di evitare accessi non autorizzati e migliorare la sicurezza della rete.

- **Supporto a Chatroom Multiple**: Attualmente tutti i peer partecipano a una sola conversazione globale. Una possibile estensione prevede la possibilità di creare e

gestire più stanze di chat, permettendo conversazioni separate tra gruppi di utenti, anche con gestione di diritti (moderatori, accessi privati, ecc.).

- **Monitoraggio e Debug Distribuito**: L'introduzione di una dashboard di monitoraggio con visualizzazione in tempo reale dello stato dei nodi (online/offline), della topologia della rete e delle metriche (latenza, messaggi al secondo, disconnessioni, ecc.) aiuterebbe nella manutenzione e nel debugging di reti distribuite complesse.
- **Meccanismi di Backup e Recovery**: In caso di crash imprevisti o interruzioni prolungate, il sistema potrebbe beneficiare di un meccanismo di salvataggio periodico dello stato del nodo e di ripristino automatico alla successiva esecuzione, per garantire la continuità e la ripresa dello stato precedente.

Queste estensioni permetterebbero di trasformare il prototipo attuale in una soluzione più completa e utilizzabile in contesti reali, affrontando al contempo alcune delle sfide più interessanti dei sistemi distribuiti moderni.

Conclusioni

Il progetto ha permesso di esplorare in modo concreto e approfondito i principi fondamentali dei sistemi distribuiti, fornendo un caso applicativo reale e funzionale: una chat peer-to-peer resiliente, autonoma e completamente decentralizzata. L'intero sistema è stato costruito con tecnologie di base come `socket` e `thread` in Python, dimostrando come sia possibile implementare un'infrastruttura di comunicazione affidabile senza ricorrere a strumenti esterni complessi o a servizi cloud centralizzati.

Uno degli aspetti più significativi del progetto è la capacità del sistema di tollerare guasti critici, come la disconnessione del nodo server, grazie a un protocollo di elezione deterministico che assicura la continuità del servizio. Questo meccanismo, pur semplice, è stato efficace nel gestire scenari imprevisti, minimizzando i tempi di inattività e garantendo la stabilità della rete.

Dal punto di vista architetturale, la progettazione modulare ha permesso una buona organizzazione del codice e un'elevata chiarezza nella gestione delle funzionalità. La separazione tra client e server, l'utilizzo di thread dedicati e l'uso di strutture dati condivise (come la peer list) hanno reso il sistema facilmente estendibile e manutenibile. Ogni nodo è in grado di adattarsi dinamicamente ai cambiamenti della rete, assumendo il ruolo più adatto al contesto senza interventi esterni.

In prospettiva, questa esperienza ha fornito una base solida per lo sviluppo di sistemi distribuiti più complessi. Le estensioni proposte — come l'introduzione di una GUI in React, la gestione di più chatroom, o il supporto a scenari Internet con NAT traversal — mostrano quanto ampio sia il margine di miglioramento e quanto il progetto possa evolversi in un sistema di comunicazione distribuito completo e professionale.

In definitiva, il lavoro svolto ci ha offerto uno spunto pratico per applicare e consolidare concetti teorici dei sistemi distribuiti, aprendo nuove possibilità di studio, sperimentazione e sviluppo software in ambienti decentralizzati.