Laboratorio di Sistemi Operativi A.A. 2024-25

Referente gruppo: lisa.furini@studio.unibo.it

Componenti del gruppo :

Sofia Viarani 0001115916
-
Lisa Furini 0001116844
-
Marica Paternicola 0001099782
-
Cristina Cei 0001113896

SOMMARIO

1. Descrizione del progetto consegnato

a) Architettura generale

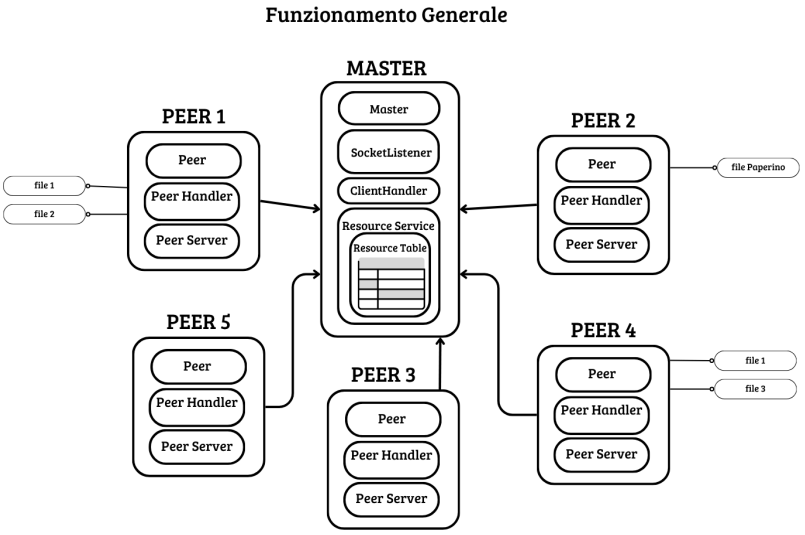
Visione di alto livello di quali sono le componenti in gioco, di come interagiscono tra loro e delle informazioni che si scambiano per far funzionare il progetto.

Il progetto realizza un'architettura peer-to-peer con la presenza di un nodo centrale Master, che funge da punto di coordinamento tra i vari nodi Peer.

Ogni Peer è in grado di condividere le proprie risorse, richiedere risorse da altri nodi e interagire con il Master per ottenere informazioni sulla disponibilità delle risorse nella rete.

Il Master gestisce una tabella aggiornata delle risorse nella rete, registrando quali risorse sono messe a disposizione da ogni peer.

Quando un peer si connette o esegue comandi, il Master aggiorna le informazioni interne.



Le comunicazioni tra Peer e Master avvengono in maniera mutualmente esclusiva, così da evitare conflitti tra aggiornamenti e richieste contemporanee.

b) Descrizione dettagliata delle singole componenti

1. Funzionalità interattive delle classi Peer e Master

Il progetto è strutturato in due cartelle principali:

- *Master/ → contiene la logica del nodo centrale.*
  Oltre a gestire le richieste dei Peer, include alcuni comandi interattivi per monitorare il sistema:
  1. listdata: mostra la tabella delle risorse conosciute e per ognuna i peer associati
  2. log: elenca le operazioni di download eseguite (sia con esito positivo, che negativo).
  3. quit: termina in modo controllato il Master.

- *Peer/ → contiene la logica dei nodi distribuiti.*
  Ogni Peer può:
  1. registrarsi al Master comunicando le risorse locali;
  2. richiedere la lista dei peer o delle risorse disponibili sulla rete;
  3. richiedere e scaricare risorse da altri nodi;
  4. aggiungere nuovi file locali notificando il Master.

Simulazione Comandi Interattivi

Comandi	Output
Master : PC C:\User\Progetto > Java Master.Master 9000	Master in ascolto sulla porta 9000
>listadata	- cartella - cartella1 - cartella2
>log	Risorse scaricate: [2025-11-07 10:25:57] file3 da: N/A a: peer2 [FALLITO - non disponibile] [2025-11-07 10:26:54] file2 da: peer2 a: peer3 [oK] [2025-11-07 10:27:22] file2 da: peer3 a: peer3 [FALLITO - già posseduta]
>quit	Master terminato.

I principali comandi interattivi sono:

01. listdata local = mostra le risorse locali, presenti nella cartella resources.
02. listdata remote = mostra le risorse associate ad ogni peer.
03. add = associa un peer ad una risorsa (già presente localmente o nuova)
04. download = scarica una risorsa remota (con gestione automatica dei fallimenti e nuovo tentativo su altri Peer).
05. quit =chiude la connessione del Peer.

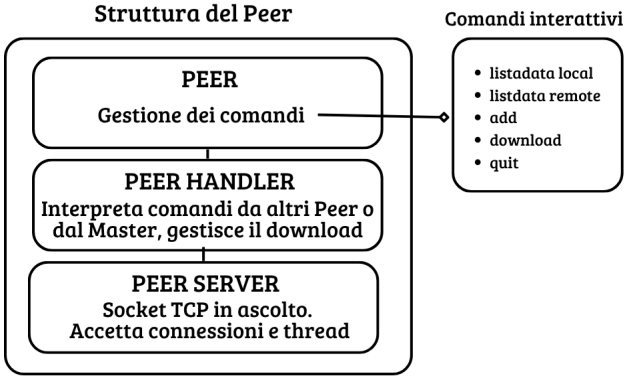
Simulazione Comandi Interattivi

Comandi	Output
Peer : PC C:\User\Progetto > java Peer.Peer 127.0.0.2 9000 peer1 6000	Connesso al master su 127.0.0.2:9000 come peer1 [PeerServer] In ascolto sulla porta 6000 REGISTERED peer1
>listadata local	Risorse locali: - cartella - cartella1 - cartella2
>listadata remote	cartella4: peer1 file 1 : peer1,peer2
>download cartella4	Tentativo di download da peer1 (127.0.0.1:6000) Risorsa 'cartella4' ora associata al peer 'peer2'. Download completato con successo da peer1
>add nuovoFile “ciao”	Risorsa nuovoFile creata localmente. Risorsa 'nuovoFile' ora associata al peer 'peer1'.
>quit	Client terminato. Socket closed

2. Struttura interna del Peer e il funzionamento logico

La parte del Peer è composta da più classi con ruoli distinti:

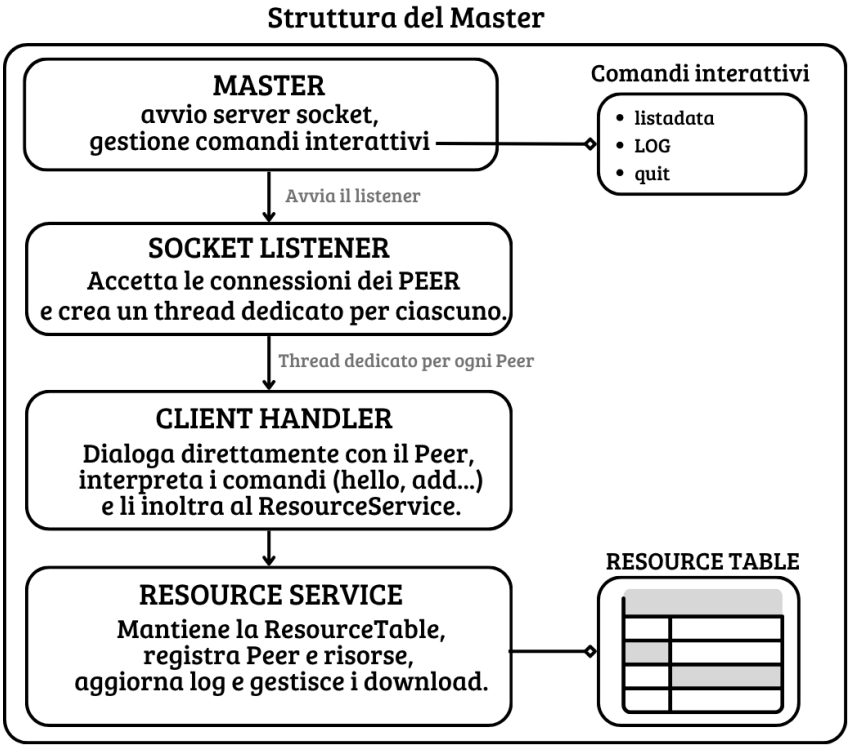
- Peer: gestisce i comandi interattivi e l’avvio del nodo.
- PeerHandler: riceve e gestisce le richieste provenienti da altri Peer, garantendo la mutua esclusione durante i download.
- PeerServer: mantiene attivo il socket in ascolto e gestisce la coda delle richieste.



3. Struttura interna del Master e il funzionamento logico

Come già anticipato, il Master è fondamentale per il funzionamento logico ed è composto da 4 classi:

- Master = classe principale, responsabile dei comandi interattivi, che avvia un server Socket in ascolto e inizializza le componenti necessarie al funzionamento del sistema
- SocketListener = è la classe responsabile di mantenere attivo l'ascolto delle connessioni in arrivo dai peer. Ogni volta che un peer si collega, viene creato un nuovo thread dedicato alla comunicazione con quel peer, garantendo l'elaborazione simultanea di più richieste. Inoltre, si occupa di chiudere tutte le connessioni attive quando il Master viene terminato.



- ClientHandler = è la classe che dialoga direttamente con i Peer.

  Riceve i comandi inviati (come hello, add, download, updatefail, ecc.) e li interpreta.

  Ogni comando viene elaborato e inoltrato al servizio principale (ResourceService) per essere eseguito.
- Resource Service= è la classe che si occupa di mantenere le informazioni riguardanti peer e risorse disponibili.

Tiene aggiornata una tabella condivisa, registra i peer collegati con i relativi IP e porta, gestisce i download e mantiene un log delle operazioni effettuate.

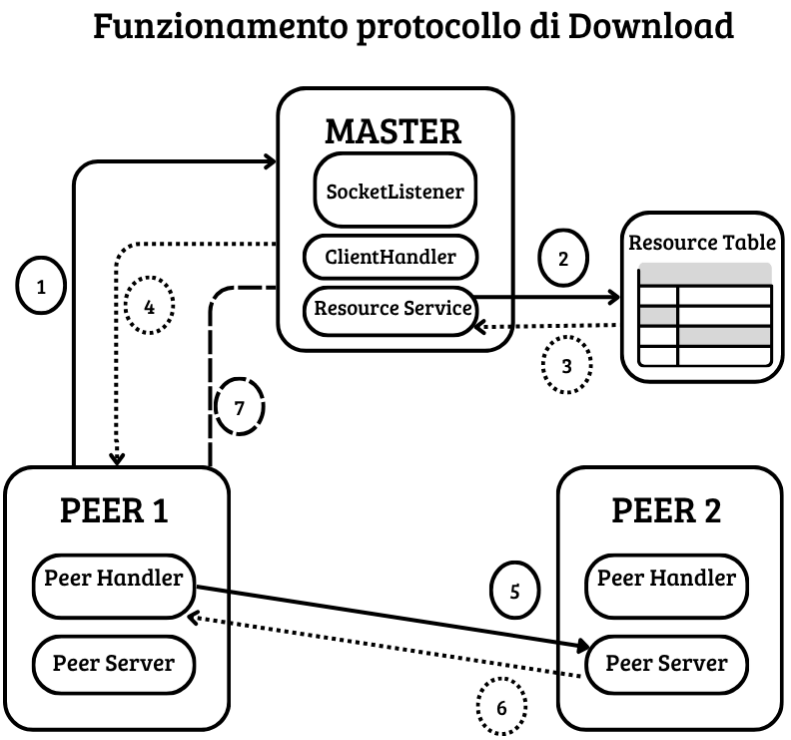
Tutti i suoi metodi sono sincronizzati per evitare conflitti tra richieste concorrenti.

4. Protocollo di download

caso 1.Funzionamento protocollo di download [Caso-OK]

Il funzionamento del download può essere interpretato graficamente, come mostrato nell'immagine, per facilitarne la comprensione.

Abbiamo progettato il funzionamento del download partendo proprio da questo schema.



In questo esempio troviamo due peer (Peer1 e peer2) e il master.

Peer1 ha bisogno del file "Paperino.txt" e non sa quale peer lo possieda .

FASE 1

Peer1 si connette al Master come Peer1.

Una volta stabilita la connessione, il Master accetta il peer attraverso il proprio ServerSocket, che rimane in ascolto costantemente in un thread dedicato (gestito dalla classe SocketListener).

Peer1 chiede al Master a quale Peer fare richiesta per scaricare la risorsa Paperino.txt.

In questa fase il Peer1 comunica al Master la propria identità e la porta locale su cui sarà disponibile per lo scambio di risorse, inviando il comando:

hello Peer1 6000

Il Master, tramite il metodo registerPeer() della classe ResourceService, memorizza il nome del peer, il suo indirizzo IP (ottenuto automaticamente dalla socket) e la porta su cui potrà essere contattato dagli altri peer.

A questo punto la connessione è stabile e il Master ha salvato le informazioni riguardanti il Peer1, che a questo punto richiede il download attraverso il comando :

download Paperino.txt Peer1

FASE 2 /3 /4

Nella seconda fase il Master ha ricevuto la richiesta di download da parte del Peer1 e la elabora interrogando la propria Resource Table attraverso il metodo HandleDownload :

public synchronized void handleDownload()

Questo metodo controlla a quale peer è associata la risorsa e restituisce ip e porta corrispondente.

Nel caso in cui la risorsa sia associata a più peer, vengono rilasciati i dati della prima corrispondenza trovata.

Nel caso di errore, in cui il peer individuato come il peer sorgente nelle resourceTable in realtà non possiede più la risorsa, viene inviato dal Peer1 un comando che permette al Master di aggiornare l'informazione .

updatefail Paperino.txt Peer2

In questo caso, riparte il ciclo di ricerca di un peer associato alla risorsa richiesta.

FASE 5

Una volta ricevuta dal Master la risposta contenente le informazioni del peer sorgente, il Peer1 è pronto per stabilire una connessione diretta con quest'ultimo.

Questi dati vengono letti e interpretati dal Peer1 nel metodo downloadResource() della classe Peer.

All'interno di questo metodo, il Peer1 memorizza:

- il nome del peer sorgente (Peer2),
- l'indirizzo IP del peer,
- la porta TCP su cui è in ascolto il suo PeerServer.

Una volta ottenute queste informazioni, il Peer1 utilizza la classe PeerHandler per avviare la comunicazione diretta e scaricare il file.

FASE 6

Il Peer1, tramite il metodo downloadFromPeer() della classe PeerHandler apre un socket diretto verso il peer sorgente, quindi il peer2.

Il Peer1 sovrascrive il file nella cartella *resources*; il trasferimento termina quando viene ricevuta la stringa END. Se il file è stato ricevuto correttamente, la funzione restituisce true e il peer1 considera il download completato con successo.

FASE 7

Una volta concluso il download, il Peer1 notifica il Master di possedere la risorsa, inviando il comando:

add Paperino.txt Peer1

Il Master, attraverso il metodo addResource() di ResourceService, aggiorna la ResourceTable,aggiungendo il Peer1 alla lista dei peer associati al file.

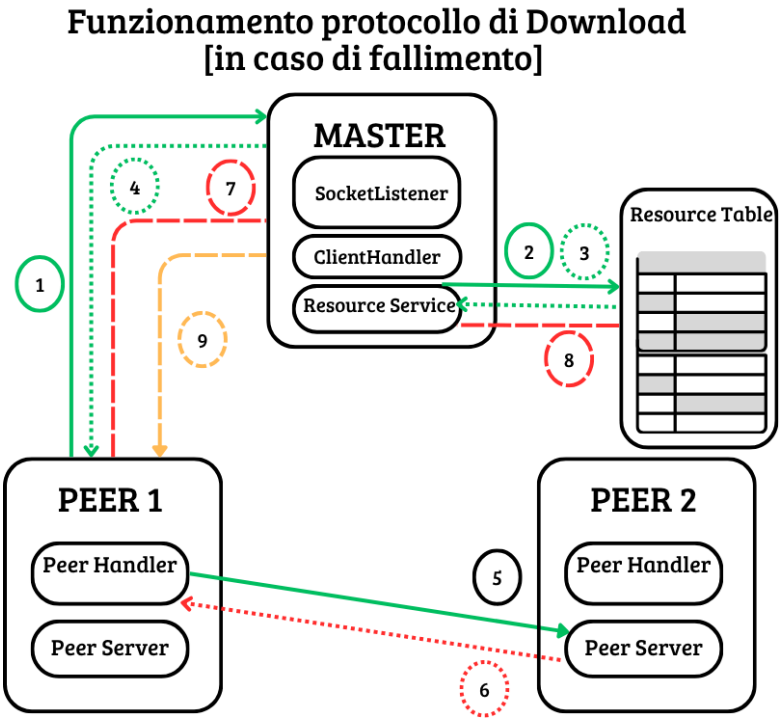
Il log del download viene registrato con esito positivo:

Es ; [2025-11-05 15:32:01] Paperino.txt da: Peer2 a: Peer1 [OK]

caso 2. Funzionamento protocollo di Download [Errore]

Nel caso in cui il Peer1 non riesca a completare correttamente il download della risorsa richiesta, il protocollo prevede una gestione dell'errore che permette di verificare con il Master se altri possiedono la stessa risorsa. Quindi, riparte con il protocollo di download: se ne esistono, il download avviene come sopra indicato (passaggi da 1 al 6), altrimenti risponde con "nessun peer disponibile " e termina il download.

Analizziamo i vari passaggi nel caso di errore:



PUNTO 6

Peer2 comunica a Peer1 di non possedere più la risorsa richiesta oppure di non essere più disponibile per il download.

Nel terminale del Peer1 viene segnalato il messaggio di errore:

Il peer peer2 non ha fornito la risorsa. Richiedo un altro peer...

PUNTO 7

Quando il download non va a buon fine, il Peer1 invia un messaggio di aggiornamento al Master per segnalare che il Peer2 non possiede più la risorsa.
Il messaggio inviato è:

updatefail Paperino.txt peer2

PUNTO 8

Il Master riceve questo comando e, attraverso il metodo updateFail() della classe ResourceService, aggiorna la Resource Table rimuovendo l'associazione tra il Peer2 e la risorsa Paperino.txt.
Questo evita che futuri download vengano indirizzati verso un peer che non possiede più il file.

PUNTO 9 -caso A

[Nuovo tentativo di download]

Dopo l'aggiornamento, il Peer1 ripete la richiesta di download al Master:

download file Paperino Peer1

Il Master controlla nuovamente la Resource Table e, se esiste un altro peer che possiede la risorsa, restituisce i nuovi dati (nuovo IP e porta) per tentare nuovamente il download.

Se invece non esistono altri peer che possiedono la risorsa, il Master invia la risposta:

Nessun peer disponibile per la risorsa 'Paperino.txt'

PUNTO 9 - CASO B

In assenza di altre sorgenti, il Peer1 termina il processo di download e registra nel log il tentativo fallito.

Il Master aggiorna il proprio log con l'esito negativo dell'operazione:

[2025-11-06 11:17:12] - Paperino.txt da: Peer1 a: Peer2 [tentativo fallito]

In questo modo il sistema mantiene la coerenza dei dati, assicurando che le informazioni sulle risorse disponibili nella rete P2P siano sempre aggiornate.

5. Funzionamento e Gestione Thread

Nel nostro progetto abbiamo utilizzato diversi thread per gestire in modo concorrente le connessioni tra il Master e i vari Peer e per permettere lo scambio simultaneo di risorse tra più nodi della rete.

Abbiamo organizzato i thread in modo che ogni componente del sistema potesse lavorare in parallelo, senza bloccare le altre.

Di seguito descriviamo in dettaglio come abbiamo gestito i vari thread.

1) Thread avviato nel Master

All'interno della classe Master, il thread principale viene avviato tramite il metodo main(). Questo thread si occupa di:

- avviare il server principale (ServerSocket);
- creare il servizio di gestione risorse (ResourceService);
- rimanere in attesa dei comandi digitati da console (come listdata, log, quit).

In altre parole, con questo thread viene gestita l'interfaccia di controllo del Master, mentre la gestione delle connessioni dei peer è stata delegata a thread separati.

2) Thread di ascolto (Socket Listener)

Il secondo meccanismo di concorrenza che abbiamo inserito si trova nel Master (riga 36, 40 e 43).

```
SocketListener listener = new SocketListener(serverSocket,resourceService);
Thread listenerThread = new Thread(listener);
listenerThread.start();
```

Questo thread resta costantemente in ascolto sul ServerSocket.
Ogni volta che un peer si connette, il listener accetta la connessione (serverSocket.accept()) e crea un nuovo thread dedicato a quel peer.

In questo modo, il Master può gestire più peer contemporaneamente: mentre un thread comunica con un peer, gli altri thread possono continuare a ricevere o inviare dati ad altri peer.

3) Thread per ogni Peer che si connette al Master

Nel metodo run() del SocketListener, ogni volta che un peer si connette al Master, viene eseguita questa istruzione, creando un thread per ogni peer:

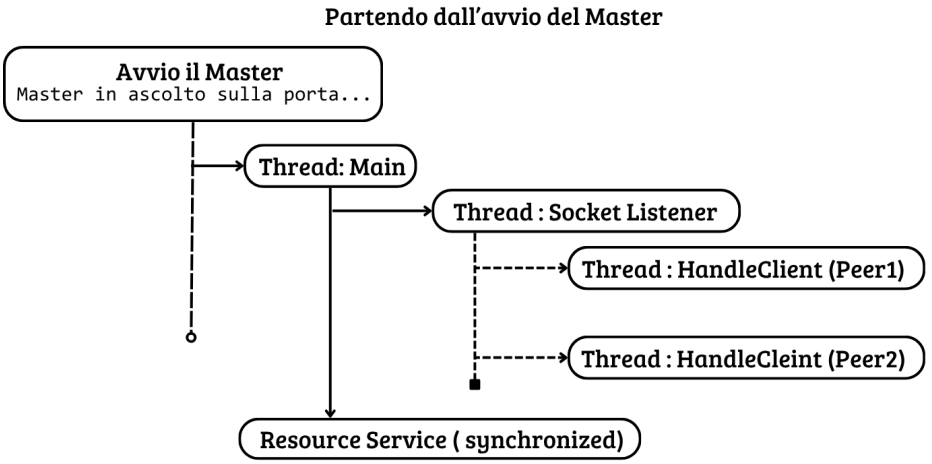
```
new Thread(() -> handleClient(clientSocket)).start();
```

Ogni thread esegue il metodo handleClient(), che si occupa di ricevere comandi dal peer (come add, download, check, ecc.) e di rispondere in modo indipendente dagli altri peer connessi.

Questa soluzione ci ha permesso di:

- gestire più peer in parallelo,
- evitare blocchi dovuti a un peer lento o disconnesso,
- mantenere aggiornate le risorse in tempo reale.

Tutti questi thread condividono l'oggetto ResourceService, che abbiamo reso thread-safe utilizzando il modificatore synchronized nei metodi.
In questo modo, anche se più thread accedono contemporaneamente alla stessa mappa di risorse, i dati rimangono coerenti e non si verificano conflitti.



4) Thread principale nella classe Peer

Anche la classe Peer ha un thread principale, che viene avviato nel suo main().
Questo thread gestisce:

- la connessione al Master;
- la lettura dei comandi inseriti dall'utente da tastiera.
- l'invio dei comandi (add, download, listdata, ecc.) al Master.

Abbiamo quindi separato la parte interattiva del peer (che comunica col Master) dalla parte che fornisce risorse ad altri peer.

5) Thread nel PeerServer

Ogni peer, una volta avviato, lancia un server interno chiamato PeerServer, che resta in ascolto su una porta locale.

Lo abbiamo avviato con questa istruzione:

PeerServer server = new PeerServer(localPort);
new Thread(server).start();

Questo thread si occupa di ascoltare eventuali richieste provenienti da altri peer che vogliono scaricare un file, trasformando ogni peer in un piccolo server indipendente.

6) Thread per ogni richiesta di download (nei peer)

All'interno del PeerServer, ogni volta che un altro peer si connette per chiedere una risorsa, viene creato un nuovo thread che gestisce quella richiesta:

```
new Thread(() -> handleClient(clientSocket)).start();
```

Questo thread legge il nome del file richiesto, lo apre e lo invia al peer richiedente.
Per evitare che più richieste simultanee accedano allo stesso file, abbiamo introdotto un Semaphore:

```
private final Semaphore mutex = new Semaphore(1);
```

Grazie al semaforo, abbiamo garantito che una sola richiesta alla volta possa leggere e inviare un file, evitando corruzioni o letture parziali.

c) Suddivisione del Lavoro

Il nostro gruppo era formato da persone che non avevano mai lavorato insieme in precedenza.
Fin dall'inizio abbiamo deciso di collaborare costantemente, lavorando spesso insieme per confrontarci, aiutarci e trovare soluzioni comuni ai problemi incontrati.

Tuttavia, come richiesto dal progetto, ognuna di noi si è concentrata in modo particolare su una parte specifica del lavoro, approfondendo quella sezione in autonomia per poi integrarla nel progetto complessivo. In questo modo siamo riuscite a mantenere una collaborazione continua, ma rispettando anche la divisione dei compiti e delle responsabilità.

- Sofia Viarani
  - Architettura generale comandi interattivi (e download)
  - Gestione dei thread lato Master
  - Stesura relazione Parte 1 Descrizione del progetto consegnato (testo e grafiche)
  - Gestione GitHub

- Lisa Furini
  - Debug del codice
  - Ottimizzazione della gestione degli errori durante le operazioni di download e aggiornamento delle risorse
  - Implementazione e test comando log lato master
  - Stesura relazione Parte 2 Descrizione e discussione del processo di implementazione

- Marica Paternicola
    - Stesura della relazione punto 3 “ Requisiti e istruzioni passo-passo per compilare e usare le applicazioni consegnate”
    - Sviluppo e implementazione condivisa della logica di download tra peer e master
    - Supporto al debugging e ai test di rete tra peer.
    - Strutturazione iniziale della relazione secondo il modello previsto dal docente

- Cristina Cei
    - Comandi interattivi lato Peer (listdata local, listdata remote, quit)
    - Comandi interattivi lato Master (listdata, quit)
    - Gestione dei thread lato Peer

2. Descrizione e discussione del processo di implementazione

a) Descrizione dei problemi

2.a.1 Problemi legati alla concorrenza

Analizziamo le risorse ritenute critiche perché condivise, contenute all’interno del nostro modello Client-Server, per cui è stato necessario gestire la concorrenza.

1. Mappa Resource table

Definita in ResourceService,java, è la mappa che associa ad ogni risorsa la lista dei peer che la possiedono. È alla base di listdata remote e download, per cui l’accesso avviene sia in lettura, che in scrittura. Viene considerata critica perché, nel caso in cui più thread vi accedessero contemporaneamente per modificarla (per un download oppure per un add), si rischierebbe di provocare incoerenza nei dati.

2. Mappa Peers

Anch’essa definita in ResourceService,java, è la mappa contenente la lista dei peer attivi e per ognuno le relative informazioni (nome, IP e porta). Viene popolata tramite hello e usata dal master durante un download per fornire le coordinate di un SourcePeer.

Viene considerata critica perché è necessario garantire la coerenza dei dati: è importante che, nel caso in cui il Master decidesse di fornire le informazioni di un SourcePeer ad un peer richiedente, queste non vengano modificate o rimosse nel frattempo da un altro thread, rendendo quelle stesse informazioni obsolete o incomplete.

3. ArrayList downloadLog

Anche questo definito in ResourceService.java, memorizza tutti i download effettuati, con orario ed esito associato, essenziale per il comando log del master. Viene considerato critico perché se i dati dovessero essere compromessi a causa della modifica in contemporanea da parte di più thread, verrebbe compromessa l’intera tracciabilità dei download, rendendo il registro inutilizzabile per il debugging.

4. Lista clients

Definita in SocketListener, contiene i riferimenti a tutti i socket attivi. Quando il SocketListener accetta una nuova connessione, si crea un nuovo socket associato alla connessione e si aggiunge alla lista, che viene poi utilizzata per chiudere tutti i socket quando il master effettua il quit. Essendo modificata contemporaneamente da SocketListener (per l'aggiunta di nuove connessioni) e dal thread principale del Master (che itera per chiudere i socket in closeAllClient), viene considerata una risorsa critica perché, senza la mutua esclusione, si verificherebbe una race condition, in cui un thread tenta di iterare su una lista mentre un altro thread la sta modificando.

All'interno del Master, la soluzione adottata per la gestione di queste risorse critiche si basa sull'implementazione della mutua esclusione attraverso la keyword "synchronized".

È stata applicata attraverso due metodologie differenti:

a) A livello di metodo

Dichiarando un metodo come public synchronized, si impedisce che possa essere eseguito da più di un thread per volta sull'istanza corrente della classe. Quando un thread entra in un metodo così definito, acquisisce un lock sull'oggetto, impedendo ad altri di accedere contemporaneamente ad altri metodi synchronized della stessa istanza.

In ResourceService questo meccanismo è necessario: infatti, contiene gran parte delle strutture dati condivise. Se più thread potessero accedervi contemporaneamente, si potrebbero verificare inconsistenze o errori di sincronizzazione.

Ad esempio, se due peer tentassero contemporaneamente di chiamare addResource e handleDownload, il Master non potrebbe gestire la coerenza tra Peers e ResourceTable.
Il lock sull'istanza resourceService serializza questi accessi critici, garantendo la correttezza dei dati centrali.

b) A livello di blocco

Questa forma viene utilizzata per gestire un blocco di codice a cui può accedere un solo thread per volta. Nel caso di Clients, in SocketListener, la parte di codice presente all'interno del blocco permette di aggiungere alla lista dei socket attivi un nuovo socket, corrispondente ad una connessione appena creata. Il lock viene acquisito solo per il tempo minimo in cui viene creato il socket e aggiunto alla lista, permettendo al resto del metodo run() di procedere in modo concorrente. Il punto cruciale è che il lock viene acquisito sulla risorsa stessa e non sul metodo intero (caso precedente): questo garantisce che l'operazione di aggiunta di un nuovo socket da parte del thread di SocketListener sia mutualmente esclusiva rispetto all'operazione di iterazione che avviene in closeAllClient() nel Master e che quindi ogni thread (sia in closeAllClient, sia nel SocketListener) debba acquisire lo stesso lock prima di accedere alla lista Clients.

Ad esempio, se il thread SocketListener sta aggiungendo un elemento socket alla lista, il thread principale del Master non può accedere a closeAllClient nello stesso momento. Il lock è acquisito sulla risorsa: questo garantisce che, pur essendo in classi diverse, entrambi i thread devono acquisire lo stesso lock prima di accedere alla lista.

All'interno del Peer, per serializzare le richieste di download in ingresso, abbiamo implementato il meccanismo dei semafori. All'interno della classe PeerServer.java,

anziché utilizzare nuovamente synchronized, abbiamo ottenuto la serializzazione attraverso l'oggetto java.util.concurrent.Semaphore.
È stato inizializzato un semaforo binario, quindi con un singolo permesso, di cui si deve acquisire il lock all'inizio del metodo tramite mutex.acquire().Se un thread sta già eseguendo il trasferimento di un file, il semaforo non avrà permessi disponibili, facendo accodare e rimanere in attesa tutti coloro che cercano di acquisire il semaforo ed entrare nel metodo. La sezione critica protetta è l'intero metodo handleClient, che gestisce la logica di ricezione di una richiesta, la verifica della presenza del file e il trasferimento dei dati. Il rilascio del permesso è posizionato nel blocco finally, garantendo che il blocco venga rilasciato in ogni caso (anche con errori o eccezioni), evitando un deadlock e permettendo al peer di servire la richiesta successiva in coda.

Valutazione delle Alternative Considerate

Abbiamo valutato diverse alternative alla keyword synchronized, di seguito elencate.

1. Utilizzo di strutture dati concorrenti thread-safe

Abbiamo considerato di sostituire le mappe e le liste standard utilizzate (HashMap, ArrayList) con le strutture dati equivalenti ma thread-safe, fornite dal package java.util.concurrent (ad esempio, ConcurrentHashMap o CopyOnWriteArrayList).

Dopo un'attenta valutazione, abbiamo preferito mantenere la sincronizzazione manuale perché più coerente con la logica delle operazioni del sistema.

Le strutture dati più moderne garantiscono coerenza e sicurezza solo sulle singole operazioni e non su sequenze di operazioni legate logicamente, che devono essere eseguite in sequenza atomica. Nel nostro caso, molte sezioni critiche coinvolgono sequenze di operazioni, come ad esempio, durante un download: lettura della ResourceTable, successivamente consultazione di Peers e poi aggiornamento di DownloadLog.

Per questo motivo è stata preferita la sincronizzazione a livello di metodo o di blocco, che consente di proteggere l'intera sequenza di operazione come un'unica unità atomica.

2. Lock espliciti con ReentrantLock

Altra opzione valutata è stata l'inserimento di ReentrantLock, fornito anch'esso dal package java.Concurrency, classe che implementa l'interfaccia Lock.

La sua caratteristica principale è il fatto di essere rientrante: questo significa che permette ai thread di riacquisire il lock più volte. È un lock esplicito e deve essere gestito manualmente (al contrario di synchronized): si chiama il metodo lock() per acquisirlo e il metodo unlock() per rilasciarlo. Nel progetto, la necessità di garantire sempre il rilascio del lock, anche in casi di eccezione, e il rischio di deadlock in caso di errore, ci ha portato a preferire la semplicità e la sicurezza automatica di synchronized alla flessibilità di ReentrantLock.

3. Scelta del modello di sincronizzazione

Nonostante la maggior parte delle operazioni che coinvolgono risorse condivise riguardino la lettura delle stesse (e non la modifica), il modello di sincronizzazione adottato sul Master non permette l'accesso a più lettori contemporaneamente; implementa, di fatto, una mutua esclusione totale.

Nel modello ideale, con lo scopo di massimizzare il throughput, si permette a più lettori di accedere alla risorsa condivisa contemporaneamente, a patto che nessuno scrittore la stia modificando in quel momento. Secondo questo principio, i metodi di sola lettura sarebbero potuti essere eseguiti in parallelo, migliorando le performance del Master.

Tuttavia, la complessità richiesta da questo tipo di implementazione e la priorità data alla garanzia del rilascio del lock, si è deciso di non implementare questa tipologia di meccanismo, che avrebbe richiesto l'uso della classe ReentrantReadWriteLock e la conseguente introduzione di lock espliciti, da gestire manualmente (come detto in precedenza, non garantiscono il rilascio del lock automatico, al contrario di synchronized). Per i motivi già sopra indicati, abbiamo preferito la sincronizzazione basata su synchronized.

2.a.2 Problemi legati al modello client-server

a) Come vengono instaurate le connessioni

La connessione tra Peer e Master segue il classico modello Client-Server basato su socket TCP in java.

Step 1. Avvio e creazione del socket

Obiettivo: creare il canale di comunicazione e registrare il peer in Peers

Il Master, all'avvio, crea un'istanza di ServerSocket, associata ad una specifica porta: questo oggetto mette il Master in uno stato passivo di ascolto.

Il thread principale del Master (che opera in SocketListener) rimane bloccato nel metodo accept() finché non riceve un tentativo di connessione da parte di un peer.

Il Peer crea un'istanza di Socket, specificando l'indirizzo IP e la porta del master: questo invia una richiesta di connessione TCP al Master.

Quando il Master riceve la richiesta, il metodo accept() viene sbloccato e restituisce un nuovo oggetto Socket; da qui il controllo viene passato ad un thread dedicato (ClientHandler), così da permettere al thread principale del Master (SocketListener) di tornare immediatamente in attesa su ServerSocket per accettare nuove connessioni.

Questo nuovo Socket restituito crea un canale bidirezionale, con cui Peer e Master comunicano. Se la connessione ha successo, i due socket (quello del Peer e quello appena creato dal Master) sono collegati e possono scambiarsi comunicazione utilizzando i rispettivi stream di I/O, che rimarranno aperti durante tutta la comunicazione.

Step 2. Registrazione logica del nuovo Peer

Dopo l'instaurazione del canale fisico, è necessario che il Peer venga registrato all'interno di Peers: il Peer invia immediatamente il comando hello <peerName> <peerPort> al Master, che tramite ClientHandler parsa il comando e chiama resourceService.registerPeer per registrare il peer e le sue relative informazioni nella mappa Peers, completando la registrazione.

b) Mantenimento delle connessioni

1. ***Lato Master***

Il thread ClientHandler entra in un ciclo bloccante di lettura, in attesa dei comandi inviati dal Peer; il ciclo termina quando viene chiuso lo stream di input.

Ogni riga letta corrisponde ad un comando: ClientHandler è responsabile di parsare questo comando e chiamare la funzione corrispondente nel ResourceService, delegando l'esecuzione della logica del protocollo e della gestione dello stato (ad esempio addResource, handleDownload). Questa delega permette al ClientHandler di mantenere la connessione attiva finché il peer è nella rete, processando i comandi e inviando le risposte tramite il canale I/O dedicato.

Parallelamente, il thread che esegue SocketListener è ancora in ascolto e attende l'arrivo di nuove connessioni su accept(): il suo ruolo è cruciale per garantire la continuità del servizio Master, accettando nuove connessioni senza essere bloccato dall'elaborazione dei comandi dei peer già connessi.

2. ***Lato peer***

Il peer ha una molteplice funzione nel mantenimento della connessione attiva: svolge sia il ruolo di Client (comunicando con il Master), che il ruolo di Server (comunicando con gli altri peer).

A) Ruolo del thread principale Peer.main

Il thread principale della classe Peer ha il ruolo di mantenere attiva la comunicazione in uscita verso il master. Questo thread gestisce:

- il ciclo di lettura dei comandi digitati dall'utente da tastiera(attraverso lo Scanner)
- traduce i comandi presi in input e li invia attraverso il socket verso il master

Rimane attivo per tutta la durata della sessione del Peer, in attesa di input locale.

La risposta del Master viene letta immediatamente, dopo l'invio del comando.

B) Ruolo del PeerServer

Il thread del PeerServer viene avviato dal Peer nel momento in cui il peer stesso viene creato e avviato. Garantisce che il Peer sia pronto a soddisfare le richieste provenienti dagli altri peer, relative a download di risorse.

- Ascolto passivo

Resta costantemente in ascolto su una porta specifica, separata dalla connessione al Master.

- Gestione dei download

Quando un peer si connette per chiedere una risorsa, il PeerServer accetta la connessione e crea un nuovo thread dedicato per ogni richiesta. In questo modo, la classe Peer può gestire più peer contemporaneamente senza bloccare la comunicazione con il Master.

C) Protezione delle risorse

Nel thread dedicato alla gestione dei download in ingresso (creato dal PeerServer), il mantenimento e l'elaborazione del trasferimento di file sono protetti dal Semaphore, che garantisce che:

- solo una richiesta di download possa accedere al file e inviarlo

- si eviti la corruzione dei dati o letture parziali nel caso di richieste simultanee dello stesso file

c) Chiusura delle connessioni

a) chiusura del Peer

Quando un peer decide di uscire dalla rete, invia un comando di protocollo: quando l'utente esegue il comando "quit", il peer invia lo stesso comando al master tramite lo stream di output. Dal lato del master, il ClientHandler riceve il quit ed esce dal suo ciclo di lettura bloccante. Prima di procedere con la chiusura fisica, il ClientHandler chiama ResourceService per rimuovere il peer dalla resourceTable: sebbene le risorse rimangano all'interno della tabella, questo assicura che il master non possa più fornire le coordinate del peer, rendendo di fatto le risorse non accessibili per il download. Infine, il ClientHandler entra nel blocco "finally" e chiama clientSocket.close(), che chiude il canale TCP bidirezionale. La chiusura del socket provoca la terminazione del thread del ClientHandler, liberando la porta e la struttura di rete che il sistema operativo utilizzava per mantenere quella specifica connessione.

b) chiusura del Master

Quando l'utente chiude il Master, tutte le connessioni devono essere interrotte forzatamente. Il thread principale esegue il comando quit e imposta il flag running su false per uscire dal while. Chiama il metodo listener.closeAllClient definito nel SocketListener, che itera sulla lista dei socket attivi e li chiude tutti chiamando s.close().

Questo provoca l'interruzione della connessione con ogni peer, perché vengono forzatamente terminati tutti i thread ClientHandler attivi, indipendentemente dal loro stato di esecuzione. Una volta chiusi i canali individuali, l'oggetto ServerSocket viene chiuso, così da impedire che possa accettare nuove connessioni. Solo dopo la chiusura di tutti i client e del ServerSocket, il thread principale chiama listenerThread.interrupt(), svegliando il thread nel caso in cui fosse rimasto bloccato in attesa sul metodo serverSocket.accept() prima che serverSocket.close() venisse chiamato o per garantire che esca dal ciclo while (running).

d) Interruzioni anomale

1. Da parte del Peer

Nel progetto abbiamo implementato un meccanismo di tolleranza ai guasti basato sulla verifica on-demand dello stato delle risorse.

Nel caso in cui un peer richieda il download di una risorsa associata ad un peer non più attivo (a causa di un crash o perchè ha effettuato un quit), il peer richiedente non fallisce l'intera operazione, ma ritorna al Master. Al ritorno del download fallito, il peer richiedente notifica al master updatefail <risorsa> <peerSorgente>: a riceverlo è il ClientHandler, che chiama il metodo resourceService.unregisterResource(), che rimuove l'entry in resourceTable in cui quella risorsa è associata a quel peer sorgente, correggendo il dato obsoleto nel momento in cui la sua incoerenza è stata verificata. A questo punto, il peer richiedente ripete il ciclo di download: se ci sono altri peer disponibili e proprietari della risorsa, il Master ne suggerisce un altro, altrimenti informa il peer che la risorsa non è disponibile sulla rete.

2. Da parte del Master

Il crash del Master implica che l'unico nodo che detiene lo stato centrale della rete smette di funzionare: questo significa che i Peer non possono più eseguire operazioni che richiedono il

suo intervento, come add, listdata remote o download. Qualsiasi tentativo di inviare comandi al master si traduce in un errore di connessione fallita.Il thread principale dei Peer rileva l'errore sul socket e notifica l'utente che il Master non è più disponibile.

Non implementando la persistenza, i dati attuali vengono definitivamente persi.

Però, avendo un server interno (PeerServer), ogni peer mantiene una tolleranza al guasto parziale:

- Funzionalità locale

Può continuare ad eseguire il comando "listdata local", perché non richiede alcuna comunicazione di rete.

- Servizio P2P continuo

Il thread PeerServer continua a funzionare e rimane in ascolto.

b) Descrizione degli strumenti utilizzati per l'organizzazione

Per lo sviluppo del progetto abbiamo adottato un approccio misto per la scrittura, compilazione e debug del codice, utilizzando sia IntelliJ che Visual Studio Code.

La scelta di ambienti diversi ha permesso a tutti i membri di poter operare massimizzando la propria produttività all'interno di un ambiente familiare.

Pur avendo a disposizione strumenti di versionamento come GitHub, data la natura del progetto e il fatto che il gruppo abbia lavorato costantemente in presenza per la quasi totalità dello sviluppo, abbiamo preferito un metodo di collaborazione diretto, caratterizzato da una sincronizzazione diretta e immediata.

La comunicazione tra membri è avvenuta tramite Whatsapp, sia nei casi che richiedevano una comunicazione più semplice e veloce, sia nei casi in cui invece veniva richiesto un impegno e un impiego di tempo maggiore, garantendo una comunicazione fluida e in tempo reale.

Per pianificare le attività, la suddivisione dei compiti, gli incontri e le task da completare abbiamo utilizzato Google Docs, immediato, semplice e facilmente accessibile a tutti i membri del gruppo, garantendo aggiornamenti in tempo reale.

3. Requisiti e istruzioni passo passo per compilare

1. Esempi di output e comandi di avvio progetto

Il nostro progetto di sistemi operativi è stato sviluppato in JavaSE 21 ed è compatibile con qualsiasi ambiente dotato di jdk 17 o superiore.

Il sistema si compone in due moduli principali (Master e Peer) quali comunicano tramite socket TCP con scambio di messaggi testuali.

Dopo aver scaricato il codice sorgente o clonato la repository di GitHub, è necessario compilare il progetto direttamente dalla directory principale.

Tutte le classi sono organizzate in package distinti, perciò la compilazione deve includere entrambe le cartelle.

Per compilare entrambe le cartelle da terminale occorre lanciare i comandi

→ javac Master/*.java Peer/*.java.
Questo comando genera i file .class nelle rispettive cartelle dei package.
Per testare l'intero flusso (Peer1 -> Master -> Peer2 -> Peer1) occorre aprire tre terminali separati (split) in modo da simulare i tre attori principali e quindi lanciare i comandi seguenti nella directory principale:

Comandi Master

- java -cp . Master.Master <porta>: Avvia il master sulla porta<>.
- listdata: Mostra le risorse dei Peer associati
- log: Elenca i download registrati
- quit: Termina il master

Comandi Peer1 (richiedente)

- java -cp . Peer.Peer 127.0.0.1 8000 peer1 6000 : Avvia il peer1 che si connette sulla porta 8000 al master e apre la porta locale 6000
- add <file><contenuto>: Crea e registra una risorsa
- listdata local: Mostra le risorse locali
- listdata remote: Mostra le risorse globali note al Master
- download <file>: scarica una risorsa dal peer sorgente
- quit: Disconnette il Peer1 dal Master e chiude il server locale

Comandi Peer2 (sorgente)

- java -cp . Peer.Peer 127.0.0.1 8000 peer2 7000 : Avvia il peer2 che si connette sulla porta 8000 al master e rimane in ascolto sulla porta 7000
- add <file><contenuto>: Crea una risorsa locale e la registra presso il Master
- Listdata local: Mostra le risorse locali
- Listdata remote: Mostra le risorse globali
- quit: Disconnette il Peer2 dal Master e chiude il server locale

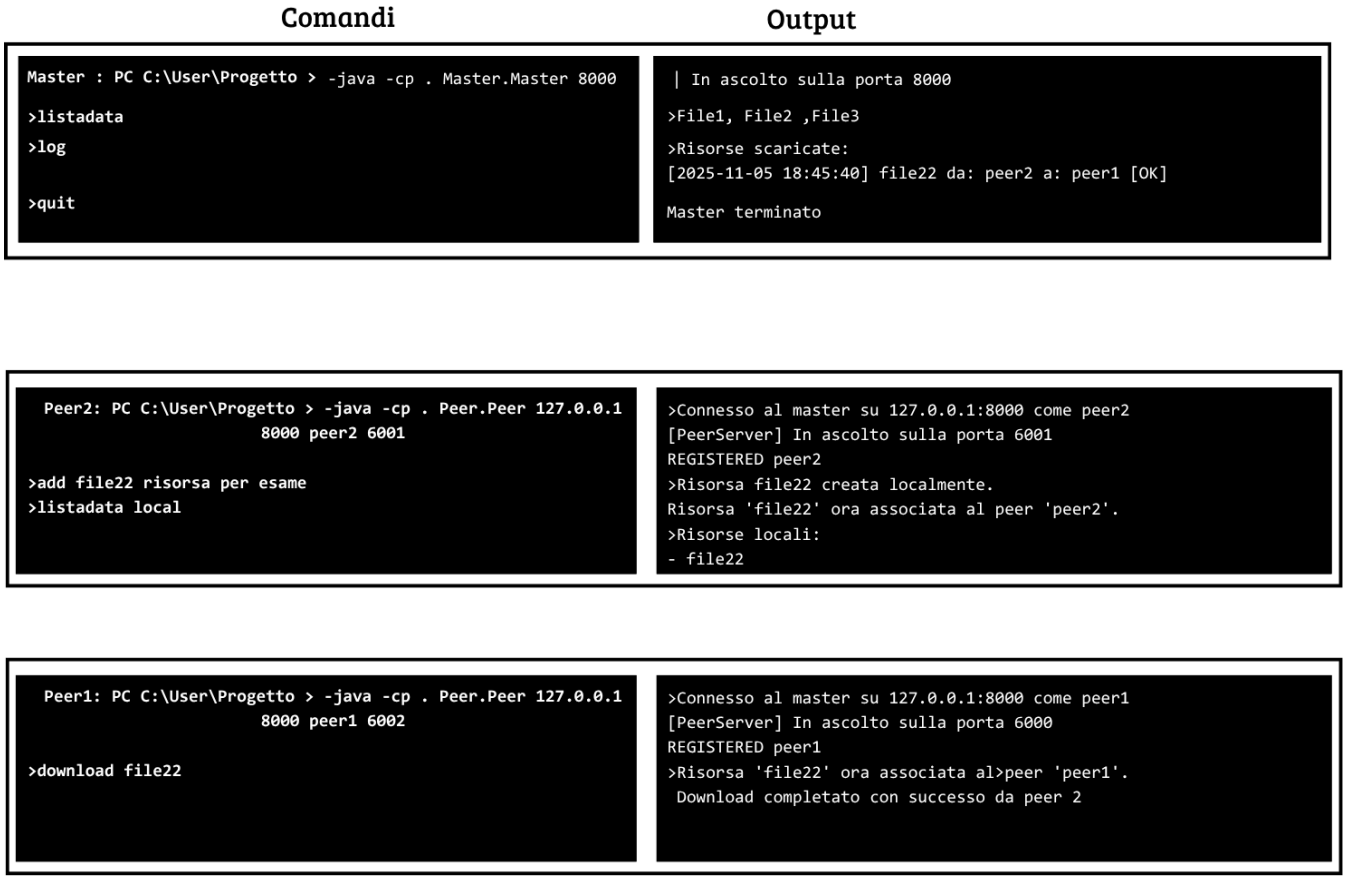
Simulazione completa del flusso Peer1 -> Master -> Peer2 -> Peer1:

La seguente simulazione dimostra il corretto funzionamento del meccanismo P2P, in particolare il flusso di trasferimento di una risorsa ("file22" in questo caso).
Il modello implementato è quello centralizzato dove il Master funge da coordinatore per lo scambio diretto tra i Peer.
Sono coinvolti i tre attori principali:

1. Master: in ascolto sulla porta 8000

2. Peer 2: Registrato come peer2 con server P2P sulla porta 6001
3. Peer 1: Registrato come peer1 con server P2P sulla porta 6000

Simulazione completo del flusso



2. Estensioni implementate

Durante lo sviluppo del progetto per rendere il sistema più robusto e coerente con i principi di affidabilità di una rete p2p, abbiamo implementato diverse estensioni al protocollo di comunicazione tra Master e Peer per migliorare la robustezza del sistema P2P.

In particolare, è stato implementato il comando updatefail, che consente al Peer richiedente di notificare al Master il fallimento di un tentativo di download da un altro peer.

Questa estensione nasce dall'esigenza di gestire in modo dinamico e affidabile le situazioni in cui un Peer indicato dal Master non possiede più la risorsa o non risponde correttamente alla richiesta.

Dal punto di vista implementativo, il comando updatefail viene inviato dal Peer richiedente al Master direttamente nel metodo downloadResource() della classe Peer.java, nel punto in cui il download fallisce.

Il Master, ricevuta la segnalazione, aggiorna la propria ResourceTable eliminando il riferimento al peer che non ha fornito la risorsa, in modo che alla successiva richiesta venga scelto un nodo alternativo.

L'utente può verificare gli effetti di questa estensione, che è trasparente a livello di comando utente, attraverso i comandi di monitoraggio del Master: listdata (che riflette l'aggiornamento in tempo reale della disponibilità delle risorse) e log (che visualizza lo storico dettagliato degli esiti P2P).

Questo approccio consente di realizzare una gestione automatica e resiliente dei fallimenti, garantendo che il Peer continui a tentare il download finché esiste almeno un altro nodo disponibile.

4. Conclusioni

Tramite questo progetto abbiamo realizzato un'architettura P2P in grado di gestire la comunicazione e lo scambio di risorse. L'aspetto più sfidante, la concorrenza, ha richiesto l'impiego sinergico di synchronized nel Master e dei semafori nel PeerServer, oltre al multi-threading applicato per elaborare simultaneamente le richieste.

Integrando meccanismi come updatefail per correggere proattivamente i dati obsoleti sulla rete, il protocollo si è dimostrato affidabile.

Nonostante i limiti del design attuale, come la mancanza di persistenza o del modello di sincronizzazione lettori-scrittori, il progetto rappresenta una solida dimostrazione dell'applicazione pratica dei principi dei sistemi operativi.