Laboratorio di Sistemi Operativi
A.A. 2024-25

Gruppo: AJAF
Referente: jeffalan.shuyinta@studio.unibo.it
Componenti del Gruppo:

Alan Jeff Shuyinta	0001126988
James Tezo Ngoufack	0001123708
Anouar El Kihal	0001124429
Filippo Bucciarelli	0001115677

1 Descrizione del progetto

Il progetto realizza un sistema di condivisione di risorse basato su un'architettura peer-to-peer centralizzata. La condivisione di risorse avviene peer-to-peer, ma le conoscenze sulla distribuzione delle risorse vengono mantenute in maniera centralizzata.

Il **Master** rappresenta il nodo centrale del sistema: mantiene una tabella globale che associa ciascuna risorsa ai peer che la possiedono, registra i peer al momento della connessione e aggiorna dinamicamente la distribuzione delle risorse nella rete in base alle operazioni ricevute.
Il Master non memorizza i file, ma fornisce ai peer le informazioni necessarie per stabilire connessioni dirette tra loro e avviare lo scambio delle risorse. È in grado di gestire più connessioni concorrenti e mantiene un file centralizzato sulle operazioni di download eseguite, oltre a file di log per le singole componenti. Queste informazioni sono consultabili tramite comandi testuali direttamente dal Master.

I **Client**, a loro volta rappresentano i peer della rete e l'interfaccia tra l'utente e il sistema.
All'avvio, ciascun Client si connette al Master, comunicando le proprie risorse. La connessione viene mantenuta fintanto che il peer rimane disponibile sulla rete. All'avvio Client lancia un proprio server per rendere disponibili allo scaricamento le proprie risorse locali. Dopo la fase di avvio, Client mette a disposizione una serie di comandi testuali che permettono di consultare le risorse locali o remote, aggiungere file alla condivisione, scaricare risorse da altri peer oppure disconnettersi.

1.1 Architettura generale

1.1.1 Master

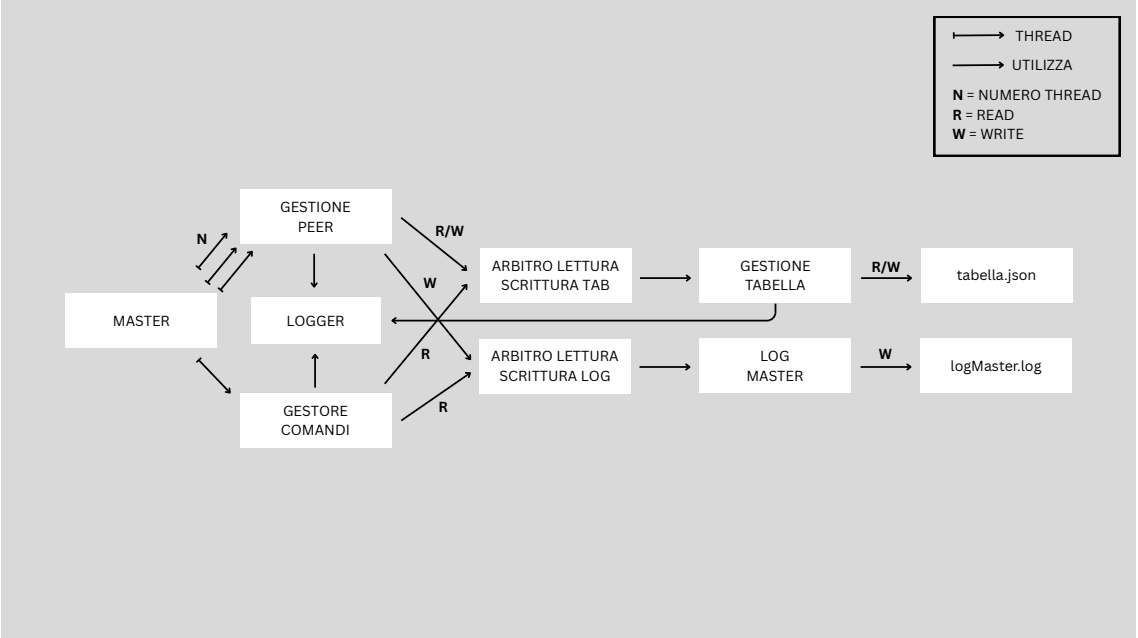

Figura 1: Schema dell'architettura generale del Master

Il **Master** gestisce le connessioni dei peer e i comandi della console attraverso l'avvio di più thread. Per ogni peer collegato viene avviato un thread `GestionePeer`, mentre per la console del Master viene avviato un unico thread `GestoreComandi`.

Risorse condivise e arbitri Le principali risorse condivise sono due:

- il file `tabella.json`, che contiene la mappatura delle risorse e dei peer che le possiedono;
- il file `logMaster.log`, che contiene lo storico di tutte le operazioni di download sulla rete, sia fallite che riuscite.

Per garantire l'accesso concorrente a queste risorse vengono utilizzate due istanze dell'arbitro `ArbitroLetturaScrittura`. Una è dedicata alla tabella (`tabella.json`), l'altro al file di log (`logMaster.log`).

Gestione delle richieste dei peer Il thread `GestionePeer` riceve le richieste dai peer e, per soddisfarle, utilizza la classe `GestioneTab` per accedere o modificare la tabella delle risorse (`tabella.json`).

Prima di poterlo fare, deve acquisire un lock tramite l'`ArbitroLetturaScrittura` della tabella, in modalità lettura o scrittura a seconda del tipo di operazione. Una volta ottenuto il lock, l'operazione viene eseguita e infine il lock viene rilasciato.

Oltre a questo, `GestionePeer` deve registrare nei log le proprie attività e gli eventi di download. Anche in questo caso l'accesso passa attraverso l'`ArbitroLetturaScrittura` dei log: per le scritture è necessario acquisire il lock in modalità scrittura.

Gestione dei comandi del Master Il thread `GestoreComandi` gestisce i comandi inseriti nella console e si occupa della chiusura dei thread che compongono il Master. Per soddisfare alcuni comandi da console può effettuare accesso in lettura a logMaster o alla tabella, in entrambi i casi avviene acquisendo il lock corretto tramite le due istanze `ArbitroLetturaScrittura`.

GestioneTab Gestisce la tabella contenente la mappatura tra risorse e peer, garantendone la persistenza nel file `risorse_rete/tabella.json`.

- **Struttura in memoria:** `Map<String, Set<String>>` (una risorsa → insieme di peer `IP:porta`).
- **Avvio:** crea la cartella se assente, carica il JSON o inizializza una tabella vuota.
- **Aggiornamenti:** aggiunta/rimozione peer e pulizia delle risorse senza peer; le modifiche aggiornano prima la memoria e poi il JSON; in caso di errore viene effettuato rollback da backup.
- **Letture:** elenco risorse o selezione del primo peer disponibile, effettuate sulla struttura in memoria.
- **Concorrenza:** la sincronizzazione è gestita dai chiamanti tramite `ArbitroLetturaScrittura`.
- **Formato JSON (esempio):**

```
{
  "file1": ["10.0.0.2:5000", "10.0.0.3:5001"],
  "file2": ["10.0.0.4:5002"]
}
```

Logger e LogMaster I componenti di logging principali sono due:

`Logger:` è un logger generico che ogni classe istanzia e utilizza per i propri messaggi informativi o di errore. Scrive su file dedicati per classe all'interno della cartella `.log/`, con nome del tipo `<NomeClasse>_<yyyy-MM-dd_HH-mm-ss>.log`. Non passa attraverso alcun `ArbitroLetturaScrittura`: non gestisce la concorrenza poiché ogni classe scrive sul proprio file separato, distinto dagli altri. Di conseguenza non esiste un file di log condiviso che richieda sincronizzazione.

`LogMaster:` è un logger specializzato e centralizzato per registrare gli eventi di trasferimento (download) fra peer. Scrive tutti questi eventi in un unico file condiviso: `.log/logMaster.log`. L'accesso concorrente a questo file è coordinato esternamente tramite un'istanza dedicata di `ArbitroLetturaScrittura`.

1.1.2 Client

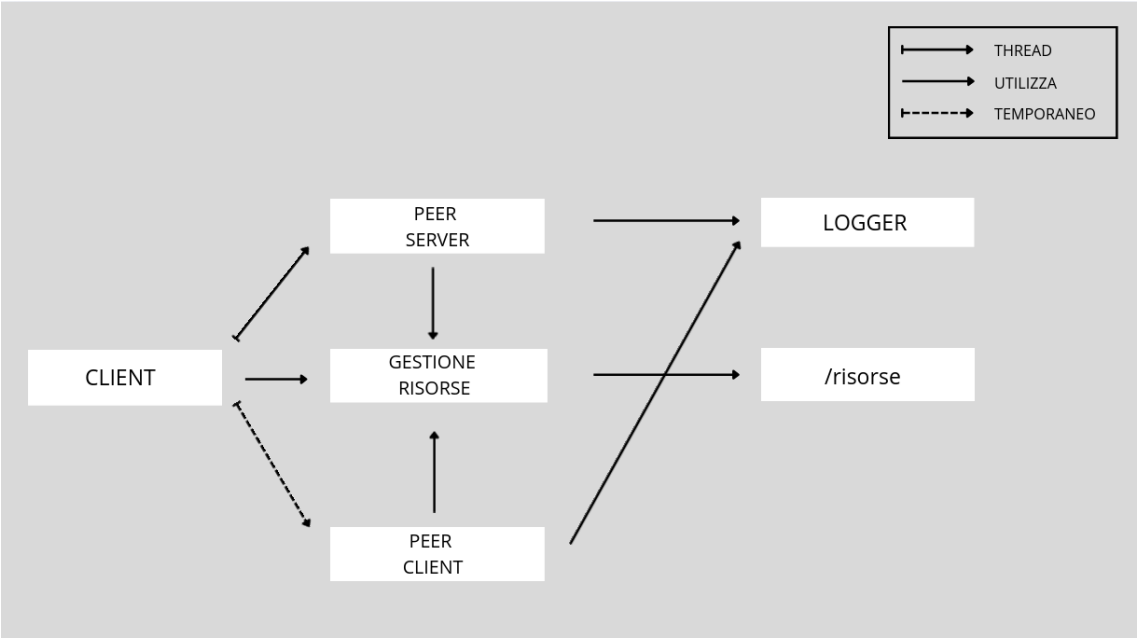

Figura 2: Schema dell'architettura generale del Client

La componente centrale è la classe `Client`, che rappresenta l'interfaccia con l'utente e il punto di partenza di tutte le operazioni. Oltre a gestire l'input da console, il `Client` avvia e coordina le altre componenti principali del sistema. In particolare:

PeerServer Dal Client viene avviato un thread dedicato, il `PeerServer`, che rimane in ascolto su una porta scelta automaticamente al momento dell'avvio. Questo thread ha il compito di permettere ad altri peer di scaricare le riorse locali condivise. La porta su cui il `PeerServer` è in ascolto viene determinata in autonomia e comunicata al Master tramite il Client

PeerClient Quando l'utente esegue il comando di download, il Client avvia un thread temporaneo, il `PeerClient`, che si connette al `PeerServer` remoto che possiede la risorsa richiesta, effettua il download e termina la propria esecuzione. Anche il `PeerClient` interagisce con la classe `GestioneRisorse`, ad esempio per salvare la risorsa scaricata nella cartella locale.

GestioneRisorse Il `Client` e il `PeerServer` si appoggiano ai metodi statici della classe `GestioneRisorse`, che funge da interfaccia verso il file system locale (`/risorse`). Attraverso questa classe è possibile elencare i file disponibili, aggiungere nuove risorse o verificare la presenza di un contenuto già esistente. In questo modo si centralizza la gestione dei file, mantenendo le operazioni sul disco separate dalla logica di rete.

Logger Sia il `PeerServer` che il `PeerClient` utilizzano la classe `Logger` per registrare eventi significativi e messaggi di errore all'interno di file di log. Questo per-

mette di avere uno storico delle operazioni (ad esempio, download riusciti o falliti) e di facilitare il debugging senza appesantire l'output su console.

1.2 Comunicazione Client-Master

La comunicazione tra Client e Master, avviene tramite connessioni TCP stabilite su socket.

Fin dall'avvio, ogni Client instaura un canale con il Master sulla porta specificata. Successivamente, lo scambio di informazioni segue un protocollo dedicato definito ad hoc, basato su stringhe testuali.

1.2.1 Fasi della comunicazione

1. **Connessione iniziale**
   Il Client apre una connessione TCP verso il Master. In parallelo, avvia il proprio `PeerServer`. Tramite un meccanismo di sincronizzazione (`CountDownLatch`), il Client attende che il `PeerServer` determini la porta. Solo in quel momento la comunica al Master.

2. **Trasmissione della porta del PeerServer**
   Il Client invia al Master un messaggio nel formato:

   ```
   PORTA:<numero_porta>
   ```

   Se la porta viene ricevuta correttamente, il Master risponde con:

   ```
   porta_ricevuta
   ```

   In caso contrario, trasmette un errore (`porta_non_ricevuta`) e chiude la connessione.

3. **Registrazione delle risorse locali**
   Dopo la fase di porta, il Client comunica la lista delle risorse disponibili in locale. La trasmissione è racchiusa tra token:

   ```
   REGISTRAZIONE_RISORSE
   <nome_risorsa_1>
   <nome_risorsa_2>
   ...
   FINE
   ```

   Il Master aggiorna la propria tabella globale delle riorse (`GestioneTab`) associando ciascun file all'indirizzo del peer e alla porta del suo `Peerserver`. Se l'operazione va a buon fine, il Master conferma con:

   ```
   aggiunto
   ```

1.3 Funzionamento componenti

1.3.1 Comandi Client

- `listdata local`

  Questo comando permette all'utente di consultare l'elenco delle risorse disponibili nella propria cartella locale (`/risorse`).

  - Il Client accede direttamente al file system attraverso la classe `GestioneRisorse`. In caso di assenza della cartella, viene creata.
  - Se la cartella è vuota, viene mostrato un messaggio di assenza di risorse. Altrimenti, vengono stampati i nomi dei file.

  Questo comando è utile per verificare rapidamente il contenuto locale e sapere quali file si stanno condividendo con il resto della rete.

- `listdata remote`

  Tramite questo comando, l'utente chiede al Master la lista delle risorse globalmente disponibili nella rete.

  - Il Client invia la stringa `LISTDATA_REMOTE` al Master tramite la socket.
  - Il thread dedicato al peer (`GestionePeer`) riceve il comando `LISTDATA_REMOTE` e invoca `listDataRemote()`. Per leggere in sicurezza la tabella globale, acquisisce il lock di lettura tramite `ArbitroLetturaScrittura`, consentendo letture concorrenti e impedendo conflitti con le scritture. All'interno della sezione critica richiama `GestioneTab.getRisorse()`, che scorre la mappa in memoria e costruisce una risposta su un'unica riga del tipo:

    ```
    nome_risorsa: [peer1, peer2]; nome_risorsa2: [peerX];
    ...
    ```

    Terminata la lettura, il lock viene rilasciato. Se la risposta è vuota (nessuna risorsa disponibile), il Master invia la stringa `"non_disponibile"`, altrimenti invia la lista formattata.
  - Il Client riceve la risposta e, attraverso il metodo `eseguiListDataRemote()` di `GestioneRisorse`, la mostra all'utente in maniera leggibile. In caso la risposta sia "non_disponibile" o non pervengano risposte, comunica che nessuna risorsa è disponibile.

  Questo comando fornisce una visione d'insieme della rete, consentendo di sapere quali file si possono scaricare e da quali peer.

- `add <nome_risorsa> <contenuto>`

  Il comando add consente di creare un nuovo file nella cartella locale delle risorse e di notificarne l'esistenza al Master.

  - L'utente specifica il nome del file e il contenuto da inserire.
  - Come prima cosa, il Client invoca `GestioneRisorse.eseguiAdd()`, che crea fisicamente il file nel file system locale.
  - Se l'operazione va a buon fine, il Client invia al Master il messaggio `ADD` <nome_ risorsa>.

- Il thread dedicato al peer (`GestionePeer`) riceve `ADD <nome_risorsa>`. Se il nome non è presente, risponde subito `"non_aggiunto"`. Altrimenti, per aggiornare la tabella globale acquisisce il lock di scrittura tramite `ArbitroLetturaScrittura` e invoca `GestioneTab.aggiungiPeer(...)`. Se la risorsa esiste, aggiunge il peer al relativo `Set`, altrimenti crea la risorsa e associa il peer. Infine il Master invia al Client l'esito: `"aggiunto"` se il salvataggio va a buon fine, altrimenti `"non_aggiunto"`.
- Infine, il Client mostra all'utente l'esito dell'operazione.-

Questo comando rappresenta il modo principale per estendere l'insieme delle risorse condivise all'interno del sistema.

- `download <nome_risorsa>`

  Il comando download permette di scaricare un file non presente in locale da un altro peer della rete.

  - Il Client invia al master la richiesta `DOWNLOAD <nome_risorsa>`.
  - Il thread dedicato al peer (`GestionePeer`) riceve `DOWNLOAD <nome_risorsa>`. Per ogni tentativo:
    1. Acquisisce il lock di lettura tramite `ArbitroLetturaScrittura` e interroga la tabella globale con `GestioneTab.getPrimoPeer(...)` per ottenere un peer disponibile, rilasciando il lock subito dopo. Se non ci sono peer associati, risponde `"non_disponibile"` e termina.
    2. Invia al Client l'indirizzo del peer selezionato e attende un feedback (`"true"`/`"false"`).
    3. Se riceve `"true"`: registra l'evento in `LogMaster` e aggiorna la tabella associando la risorsa al peer richiedente tramite `GestioneTab.aggiungiPeer(...)` e termina il comando.
    4. Se riceve `"false"`: registra il fallimento in `LogMaster` e rimuove dalla tabella il peer non raggiungibile per quella risorsa con `GestioneTab.rimuoviPeerInRisorsa(...)`. Se una risorsa rimane senza peer associati, viene eliminata dalla tabella. Quindi ripete la selezione proponendo un altro peer, se disponibile.
  - Il Client avvia una connessione TCP diretta verso il `PeerServer` del peer indicato e tenta il trasferimento.
  - Se il trasferimento va a buon fine, il Client comunica al Master l'esito positivo inviando la stringa `true`, altrimenti invia `false` e il Master propone un peer alternativo (vedi sopra).
  - Il ciclo continua fino a quando la risorsa non viene scartata con successo o fino a esaurimento dei peer disponibili.

    Se nessun peer risulta raggiungibile, viene stampato un messaggio di fallimento. In caso positivo, la risorsa viene stampata in locale e aggiunta alla cartella di condivisione.

- `quit`

  Con il comando quit, l'utente termina tutti i thread del peer in esecuzione.

  - Il Client invia al master il messaggio `QUIT`.

– Il thread dedicato al peer (`GestionePeer`) riceve `QUIT` e invoca `quitGestionePeer()`. L'operazione:

1. segna la connessione come terminata (`terminato = true`);
2. rimuove l'istanza di `GestionePeer` da `Master.listaGestoriPeer`;
3. chiude la socket TCP verso il peer (se ancora aperta), loggando l'esito della chiusura.

Non viene effettuata alcuna modifica alla tabella delle risorse (`GestioneTab`): le associazioni del peer non vengono rimosse al momento del quit e restano persistenti in `risorse_rete/tabella.json`. L'eventuale rimozione avviene successivamente tramite `GestioneTab.rimuoviPeerInRisorsa(...)` quando un download fallisce verso quel peer. Il Master resta in esecuzione per servire gli altri peer connessi.

– Parallelamente, il Client termina il thread PeerServer, rendendo indisponibili al download le proprie risorse.

– Infine, il programma termina l'esecuzione.

Questo comando è essenziale per garantire una chiusura ordinata della connessione e una corretta pulizia della tabella del Master.

1.3.2 Comandi Master

- `log`

  Con questo comando l'utente master può consultare lo storico delle operazioni di download avvenute nella rete.

  – Acquisisce il lock di lettura tramite l'istanza di `ArbitroLetturaScrittura` dedicata a `LogMaster`, ossia `arbitroLog`.

  – Stampa il log tramite `logger.stampa()`.

  – Per ogni riga vengono mostrati: la risorsa scaricata, il peer sorgente, il peer destinatario e l'esito (positivo o fallito).

  – Se non sono presenti voci, viene segnalata l'assenza di log.

  – Rilascia l'accesso in lettura con `arbitroLog.fineLettura()`.

  Questo comando è utile per monitorare l'andamento dei download e individuare eventuali errori o problemi di rete.

- `listdata`

  Tramite questo comando l'utente master può visualizzare la situazione aggiornata della tabella delle risorse gestita dal Master.

  – Acquisisce il lock di lettura tramite l'istanza di `ArbitroLetturaScrittura` dedicata a `GestioneTab`, ossia `arbitroTabella`.

  – Richiede l'accesso in lettura tramite `arbitroTabella.inizioLettura()`.

  – Ottiene i dati della tabella con `tabella.getRisorse()`.

  – Se la tabella è vuota, viene mostrato un messaggio dedicato all'assenza di risorse.

  – In presenza di dati, l'elenco viene presentato in modo leggibile e ordinato.

  - Infine, rilascia l'accesso in lettura tramite `arbitroTabella.fineLettura()`.

  Questo comando fornisce una visione immediata e globale dei file disponibili nella rete.

- `quit`

  Questo comando serve a chiudere in maniera ordinata l'esecuzione del Master.

  - Una volta digitato, la sessione viene terminata.
  - Imposta `Master.inEsecuzione = false`.
  - Chiude tutti i peer nella lista `Master.listaGestoriPeer` invocando `gp.quit()`. I peer collegati vengono disconnessi.
  - Chiude il `serverSocket` di Master e il server smette di accettare nuove connessioni.

  Tramite questo comando l'applicazione viene arrestata in modo sicuro, garantendo che non rimangano processi sospesi né porte di rete aperte.

1.4 Suddivisione del lavoro

All'inizio del progetto ci siamo organizzati in due gruppi distinti: da una parte Filippo Bucciarelli e Alan Jeff Shuyinta hanno lavorato sulla parte del client, mentre Anouar El Kihal e James Tezo Ngoufack si sono concentrati sullo sviluppo del master. Questa divisione iniziale ci è sembrata funzionale per poter lavorare in parallelo e per rendere equo il carico di lavoro.

In seguito, quando abbiamo iniziato a scrivere e testare il codice, ci siamo resi conto che l'approccio asincrono adottato stava generando difficoltà, in particolare per la necessità di dover attendere lo sviluppo di altre parti per poter procedere con il testing delle proprie, e le difficoltà di integrazione delle varie componenti.

A quel punto abbiamo deciso di ridefinire la strategia adottando un approccio orientato alla risoluzione dei problemi. In pratica, quando una difficoltà risultava circoscritta a una singola classe, ci siamo affidati al responsabile di quella classe, lasciando a lui il compito di trovare la soluzione più adatta. Nei casi invece in cui il problema interessava più parti del progetto, lo abbiamo assegnato a un membro del gruppo che si occupava di sviluppare una soluzione dedicata, aprendo un branch specifico per lavorarci in maniera ordinata e senza compromettere la stabilità del resto del codice. Questo metodo ci ha permesso di avere una visione più chiara della logica complessiva e di strutturare meglio il progetto, mantenendo solo le classi già implementate che si erano rivelate corrette e riscrivendo o completando le parti mancanti in modo coerente.

Tenendo conto di questo nuovo approccio, siamo arrivati a una suddivisione definitiva del lavoro nel seguente modo:

- **Filippo Bucciarelli**
  - ArbritoLetturaScrittura
  - PeerClient
  - PeerServer
  - Logger

- **Alan Jeff Shuyinta**
  - Client
  - Master
  - GestioneRisorse
- **Anouar El Kihal**
  - GestioneTab
  - GestorePeer
- **James Tezo Ngoufack**
  - LogMaster
  - GestoreComandi

2 Implementazione

2.1 Problemi riscontrati e soluzioni adottate

2.1.1 Accesso a `tabella.json` e `logMaster.log`

All'interno del progetto logMaster.log e tabella.json sono i due file ad accesso condiviso su cui si possono verificare race condition fra thread. Le operazioni di lettura e scrittura su questi due file, operazioni che consideriamo di sezione critica, vengono gestite dalle classi `LogMaster` e `GestioneTab`. I thread GestionePeer e GestioneComandi ottengono informazioni dai file attraverso i loro metodi richiedendo prima delle chiamate l'accesso ad un arbitro. La classe `ArbitroLetturaScrittura` è l'arbitro che gestisce gli accessi alle sezioni critiche. All'avvio di master vengono create due istanze, una per la gestione delle sezioni critiche per `GestioneTab` e una per `LogMaster`. La classe si basa un `ReentrantReadWriteLock`, un semaforo apposito per il problema di lettori e scrittori. Il meccanismo di sincronizzazione implementato consente a più lettori di accedere simultaneamente alla risorsa, ma garantisce che solo un scrittore possa accedervi alla volta, impedendo l'accesso ai lettori durante la scrittura. Al momento dell'inizializzazione, passando parametro true, viene impostata l'implementazione con fairness.

Fairness lettori-scrittori Quando il lock viene rilasciato, viene assegnato allo scrittore in attesa da più tempo, o in alternativa ad un gruppo di lettori che sommati abbiano tempo di attesa maggiore. Quando un lettore prova ad ottenere il lock, si blocca se è già posseduto da uno scrittore o se c'è uno scrittore in attesa, altrimenti lo acquisisce. Invece, quando uno scrittore prova ad acquisirlo, si blocca fino a quando tutti i lettori e scrittori rilascino il lock.

2.1.2 Download peer-to-peer

Per assolvere la richiesta di avere peer che forniscano risorse in maniera mutualmente esclusiva, l'implementazione di `PeerServer` è mono-threading. Questo significa che viene accettata una connessione per volta, mantenendo in attesa le richieste successive ma senza rifiutare le connesioni. In questo modo, supponendo che due peer B e C richiedano in contemporanea il download di una risorsa dal peer A, il

peer A assolverà prima il primo upload, e successivamente il secondo, in maniera sequenziale.

2.1.3 PeerServer su porta dinamica

Nell'implementazione iniziale `PeerServer` si metteva in ascolto sempre su una porta fissa. Questa scelta, seppur semplificasse lo sviluppo, impediva il lancio di più peer da localhost e confidava nella continua disponibilità della porta su dispositivi differenti in momenti differenti. Abbiamo quindi deciso di far determinare dinamicamente la porta alla `ServerSocket` passando come parametro porta 0, così che venga determinata dinamicamente all'avvio in base a quelle dichiarate disponibili dal sistema operativo. Per implementare questo cambiamento si è resa però necessaria la comunicazione della porta del `PeerServer` al momento della connessione fra `Client` e `Master`, e quindi la necessità di sincronizzare `Client` con `PeerServer`. A questo scopo Client crea un oggetto `CountDownLatch` che viene passato all'istanza `PeerServer`. Al momento della comunicazione della porta, il latch tramite il comando wait() mette `Client` in attesa, sino a quando `PeerServer` non determina la porta e sblocca il latch con il comando countDown().

2.1.4 Quit del master

Inizialmente, alla chiusura del Master veniva terminato solamente il thread della console, mentre i thread che eseguono le istanze di `GestionePeer` rimanevano attivi. Per risolvere questo problema è stata introdotta una lista che mantiene il riferimento a tutti i gestori `GestionePeer` attualmente connessi, così da procedere alla terminazione dei rispettivi gestori attivi al momento del quit tramite un'iterazione. Quando un nuovo peer si connette, il relativo gestore viene aggiunto alla lista. Se un peer si disconnette, il gestore corrispondente viene rimosso. L'accesso a questa struttura dati è reso mutualmente esclusivo tramite l'uso di `synchronized`, così da evitare l'insorgere di problemi di concorrenza.

2.2 Strumenti utilizzati

2.2.1 Sviluppo

Per il versionamento del codice e la gestione dei conflitti abbiamo utilizzato `git`. La repository git è hostata su **GitHub** per consentire la collaborazione. Degli strumenti messi a disposizione da GitHub abbiamo fatto uso del sistema di `Issues` grazie al quale mantenevamo traccia dei problemi, li categorizzavamo tramite tag e assegnavamo la risoluzione.

Tipologie di issue (tag):

- **bug**: qualcosa che non funziona.
- **documentation**: miglioramenti o aggiunte alla documentazione.
- **enhacement**: nuove funzionalità da introdurre o proposte.
- **help wanted**: non capisco cosa non stia funzionando, ho bisogno che anche qualcun altro controlli.

- **invalid**: l'implementazione è sbagliata (magari funziona, ma non è così che dovrebbe essere, oppure non dovrebbe proprio esserci).
- **question**: necessità di chiarimenti o informazioni aggiuntive su un aspetto specifico, richiesta di pareri.
- **wontfix**: issue segnata come non risolvibile o da non risolvere, solitamente di seguito l'issue viene chiuso.
- **master**: issue relativi al master.
- **peer**: issue relativi al peer.
- **in progress**: issue su cui qualcuno sta già lavorando, è in corso di risoluzione.

Questo metodo ci ha permesso di tenere sempre sotto controllo lo stato di avanzamento e la suddivisione dei compiti, soprattutto nel momento di sviluppo intensivo del progetto. Per legare in maniera chiara gli issue alla loro risoluzione abbiamo spesso creato branch appositi dedicati alla risoluzione di un singolo issue. Questo ha portato all'utilizzo di un numero notevole di branch, ma ha permesso una gestione dei conflitti più agevole e uno sviluppo simultaneo più chiaro.

Abbiamo inoltre fatto inizialmente uso di una semplice GitHubAction che compilasse tutti i codici prima di ogni pull request, tuttavia il risultato non era soddisfacente, per questo l'abbiamo successivamente disattivata.

2.2.2 Comunicazione interna

Per la comunicazione e il coordinamento quotidiano abbiamo utilizzato un gruppo **Telegram**, che abbiamo suddiviso in diversi topic per evitare confusione e mantenere ordinata la discussione:

- **General**: comunicazioni generali, scadenze, aggiornamenti.
- **Documentazione**: discussione sulla relazione e organizzazione dei contenuti.
- **GitHub**: gestione repository, issue, pull request.
- **Codice**: problemi tecnici.
- **Master**: sviluppo server.
- **Concorrenza**: problematiche di gestione degli accessi concorrenti.
- **Client**: sviluppo peer.

Questa suddivisione ci ha permesso di discutere in modo mirato e di non mescolare argomenti diversi.

2.2.3 Documentazione

La documentazione è stata realizzata in LaTeX e gestita in maniera collaborativa tramite **Overleaf**.

3 Compilazione e avvio

3.1 Scaricamento progetto

3.1.1 Download repository

Effettuare il download della repository da Github scaricando lo zip ed estrarlo. La struttura della directory scaricata è la seguente:

```
LABSO_AJAF
├── doc
│   └── RelazioneLABSO_AJAF.pdf
├── lib
├── Makefile
├── README.md
├── specifiche-progetto.pdf
└── src
    ├── client
    │   ├── Client.java
    │   ├── GestioneRisorse.java
    │   ├── PeerClient.java
    │   └── PeerServer.java
    ├── common
    │   └── Logger.java
    └── master
        ├── ArbitroLetturaScrittura.java
        ├── GestionePeer.java
        ├── GestioneTab.java
        ├── GestoreComandi.java
        ├── LogMaster.java
        └── Master.java
```

3.1.2 Download jar

La tabella di master è salvata in formato json e il suo funzionamento con java richiede il download di tre pacchetti jar: jackson-annotation, jackson-core e jackson-databind. Per farlo, spostarsi dentro la cartella /lib, scaricarli con wget e tornare nella directory principale tramite i seguenti comandi:

Listing 1: download jar

```
cd lib
wget https://repo1.maven.org/maven2/com/fasterxml/jackson/core/jackson-annotations/2.17.0/jackson-annotations-2.17.0.jar
wget https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-core/2.17.0/jackson-core-2.17.0.jar
wget https://repo1.maven.org/maven2/com/fasterxml/jackson/core/jackson-databind/2.17.0/jackson-databind-2.17.0.jar
cd ..
```

Per lo sviluppo abbiamo utilizzato le versioni 2.17.0 di questi pacchetti, ma anche le successive dovrebbero essere compatibili. In alternativa all'utilizzo di wget è possibile effettuare il download da browser ai seguenti link:

- jackson-annotations
- jackson-core
- jackson-databind

3.2 Compilazione

3.2.1 Compilazione codice

Abbiamo costruito un `Makefile` per la compilazione semplificata del progetto.

Listing 2: compilazione

```
make compile
```

Il comando prevede la creazione di una cartella /bin e la compilazione all'interno di essa delle classi java presenti dentro / src aggiungendo alla path la cartella /lib, contenente i jar, e la cartella /bin stessa dove sono già state compilate le altre classi.

3.2.2 Rimozione compilati

Per rimuovere le classi compilate utilizzare il comando apposito

Listing 3: pulizia repo

```
make clean
```

Il comando comporta anche la rimozione dei log presenti all'interno della cartella `/.log`.

3.3 Avvio

3.3.1 Master

Per avviare il master utilizzare il comando seguente, specificando la porta su cui mettersi in ascolto.

Listing 4: avvio master

```
make master PORT=[porta]
```

Nel caso in cui la porta non venga specificata verrà utilizzata di default `7000`.

3.3.2 Peer

Per avviare un peer utilizzare il comando seguente, specificando l'indirizzo ip e la porta del master a cui connettersi.

Listing 5: avvio peer

```
make client MASTER_IP=[ip master] PORT=[porta]
```

Nel caso in cui indirizzo ip e/o porta non vengano specificate verranno utilizzate di default `localhost` e `7000`.

3.3.3 Informazioni generali

È possibile testare il progetto sia su rete locale che su localhost. Non è funzionante su ALMAWIFI, sulla rete univesitaria pare infatti che siano bloccate le connessioni interne su porte non standard.

In caso di testing su localhost invece tenere presente che l'avvio di master e peer deve preferibilmente avvenire da cartelle differenti, seppur l'avvio dalla stessa cartella non ne comprometta il funzionamento. In caso di avvio di due client dalla stessa cartella alcune funzionalità risulteranno compromesse, in quanto la cartella risorse dei due peer risulterà condivisa.

Nel caso si vogliano preallocare risorse al peer è possibile creare la cartella /risorse e inserire le risorse desiderate all'interno. Altrimenti, al primo avvio del peer, la cartella verrà creata in automatico.