Laboratorio di Sistemi Operativi A.A. 2024-25

Documentazione Tecnica del Progetto P2P con Master Registry

<NOME GRUPPO>

Referente: referente@studio.unibo.it

21 ottobre 2025

Componenti del gruppo

Christian Toffalori	Matricola
Sara Alfieri	Matricola
Nome Cognome	Matricola

Indice

1 Introduzione

Questo documento descrive un sistema **peer-to-peer** con **registro centrale** (Master) che rispetta le specifiche del progetto di Laboratorio di Sistemi Operativi A.A. 2024–25. Il Master mantiene la mappatura tra risorse e peer; i peer si registrano, pubblicano risorse e scaricano risorse da altri peer tramite un protocollo testuale su socket.

Obiettivi principali

- Architettura P2P con Master centrale.
- Mutua esclusione sul Master durante aggiornamenti/consultazioni del registro.
- Mutua esclusione sul Peer (serve una richiesta alla volta).
- Download con token e retry robusto in caso di fallimenti.
- CLI per Master e Peer, con log delle richieste di download.

2 Architettura del sistema

2.1 Vista di alto livello

- **Master**: server TCP concorrente. Mantiene:
  - Registro `risorsa -> [peerId]` e `peerId -> PeerRef(host,port)`.
  - Emissione/revoca dei token di download.
  - Log centrale delle richieste di download (esito OK/FAIL).
- **Peer**: client del Master + server interno per servire file; mantiene una cartella risorse locale.

2.2 Struttura del repository

```
labso-p2p/
|-- pom.xml
|-- README.md
|-- run/
| |-- start-master.sh
| |-- start-peer-a.sh
| |-- start-peer-b.sh
| `-- windows/
| |-- start-master.bat
| |-- start-peer-a.bat
| `-- start-peer-b.bat
|-- data/
| |-- peer-A/
| | `-- R1.txt
| `-- peer-B/
| `-- R2.txt
|-- logs/
`-- src/
    `-- main/
        |-- java/
        | `-- it/unibo/labso/p2p/
        | |-- common/
```

```
| |-- master/
| `-- peer/
`-- resources/
```

2.3 Componenti principali

- **MasterServer**: ciclo `accept()` + thread-pool; passa ogni socket a `ClientHandler`.
- **ClientHandler**: interpreta comandi testuali (payload JSON).
- **Registry**: tabelle del registro con lock unico (mutua esclusione stretta).
- **TokenStore**: genera/gestisce `Token` con scadenza.
- **DownloadLog**: conserva eventi `(ts, risorsa, da, a, esito)`.
- **PeerServer**: server socket con `Semaphore(1)` per servire una richiesta per volta.
- **PeerClient**: effettua download byte-precise senza chiudere prematuramente il socket.
- **MasterClient**: API lato peer per parlare col Master.
- **PeerCli**: interfaccia interattiva utente (peer).

3 Protocollo di rete (Master <-> Peer)

I comandi sono line-based (UTF-8, terminati da `\n`). Formato: `<CMD> [JSON]`. Il Master risponde sempre con: `OK [JSON]` oppure `ERR <motivo>`.

Comandi supportati (Master)

CMD	Descrizione (JSON payload)
`REGISTER`	Registra/aggiorna peer e lista risorse.
`LISTDATA_REMOTE`	Restituisce mappa globale `risorsa -> [peerId]`.
`LIST_PEERS`	Restituisce mappa `peerId -> PeerRef(host,port)`.
`WHO_HAS`	`{"resource": "<nome>"}` -> lista di `PeerRef`.
`DOWNLOAD_TOKEN_REQ`	`{"resource": "<nome>", "requesterPeerId": "<id>"}` -> `{"token","resource","peer"}`.
`DOWNLOAD_TOKEN_REL`	`{"token","resource","fromPeerId","requesterPeerId"}` (rilascio token, log esito OK).
`DOWNLOAD_FAILED`	`{"resource","peerId","requesterPeerId"}` (rimuove associazione e logga FAIL).
`PEER_QUIT`	`{"peerId"}` (rimuove peer e risorse dal registro).
`QUIT`	Chiude la connessione corrente.

Comandi Peer -> Peer

CMD	Descrizione
DOWNLOAD <nome> <token>	Il peer server risponde OK <size> e invia esattamente size byte. Con **Semaphore(1)** serve una richiesta alla volta.

3.1 Sequenza di download e retry

1. Peer richiede token: DOWNLOAD_TOKEN_REQ{"resource","requesterPeerId"}.
2. Master risponde con OK {"token","resource","peer"}.
3. Peer si connette a peer.host:peer.port e invia DOWNLOAD <nome> <token>.
4. In caso di errore: DOWNLOAD_FAILED{"resource","peerId","requesterPeerId"} e ritorna al passo 1.
5. Se ok: salva file e invia DOWNLOAD_TOKEN_REL{"token","resource","fromPeerId","requesterPee

4 Comandi CLI (utente)

Master (console)

Comando	Descrizione
listdata	Stampa mappa globale risorsa -> [peerId].
inspectNodes	Stampa mappa peerId -> PeerRef.
log	Stampa elenco eventi di download con esito (OK/FAIL).
quit	Arresta il master (shutdown pulito).

Peer (console)

Comando	Descrizione
listdata local	Elenca file nella cartella del peer.
listdata remote	Mostra indice globale dal Master.
listpeers	Mostra tutti i peer registrati.
whohas <nome>	Mostra i peer che hanno <nome>.
add <nome> <contenuto>	Crea file locale e aggiorna il Master.
download <nome>	Scarica da un altro peer con token e retry robusto.
quit	Deregistra il peer presso il Master e chiude.

5 Concorrenza e sincronizzazione

5.1 Master

La struttura dati del registro e' protetta da un unico ReentrantLock (mutua esclusione stretta). In questo modo, durante gli aggiornamenti nessuna lettura procede in parallelo, aderendo alla specifica che richiede uso in mutua esclusione durante update/uso della tabella.

5.2 Peer

Il `PeerServer` utilizza `Semaphore(1)` per serializzare la gestione delle richieste di download: mentre due peer A e B comunicano, il peer B serve una sola richiesta per volta e le altre restano in attesa.

6 Istruzioni di build ed esecuzione

Prerequisiti

- Java LTS (testato con 21; la specifica menziona 24 LTS: adeguare `maven.compiler.release` se richiesto).
- Maven (o Maven bundled di IntelliJ).

Build

```
cd labso-p2p
mvn clean package -Pdist
# genera: target/labso-p2p-1.0.0-all.jar
```

Esecuzione (demo locale)

```
# Shell 1 - Master
java -cp target/labso-p2p-1.0.0-all.jar it.unibo.labso.p2p.master.MasterMain 9000

# Shell 2 - Peer A
java -cp target/labso-p2p-1.0.0-all.jar it.unibo.labso.p2p.peer.PeerMain 127.0.0.1
    9000 ./data/peer-A

# Shell 3 - Peer B
java -cp target/labso-p2p-1.0.0-all.jar it.unibo.labso.p2p.peer.PeerMain 127.0.0.1
    9000 ./data/peer-B
```

7 Test e casi attesi

Percorso consigliato

1. Peer A: `listdata local`, `add R3.txt hello`, `listdata remote`.
2. Peer B: `listdata remote`, `whohas R3.txt`, `download R3.txt`.
3. Master: `log` mostra evento OK, `inspectNodes` mostra peer attivi.
4. Retry: spegnere il peer sorgente durante `download`; il peer client mostra `retrying...` e invia `DOWNLOAD_FAILED`. Se non restano candidati, Master risponde `ERR NO_RESOURCE`.
5. Deregistrazione: `quit` su un peer; `inspectNodes` e `listdata` dal Master non devono piu' includerlo.

Mutua esclusione lato peer

Creare un file grande su `peer-A`, avviare due `download` paralleli da B: si osserva che uno parte solo quando l'altro termina (Semaphore(1)).

8 Copertura delle specifiche

Requisito	Implementazione	File principali
Gruppi/Repo/Consegna	Struttura Maven, repo privato, tag `Consegna`.	`pom.xml`, README
Master registra risorse	`REGISTER` con lock unico.	`Registry`, `ClientHandler`
Lista risorse globale	`LISTDATA_REMOTE`.	`ClientHandler`, `MasterMain`
Lista peer e WhoHas	`LIST_PEERS`, `WHO_HAS`.	`ClientHandler`, `MasterClient`
Download token-based	`DOWNLOAD_TOKEN_REQ/REL`.	`ClientHandler`, `TokenStore`
Retry robusto	`DOWNLOAD_FAILED` + loop nel peer.	`PeerCli`, `MasterClient`
Mutua esclusione Master	Lock unico su `Registry`.	`Registry`
Mutua esclusione Peer	`Semaphore(1)` nel server del peer.	`PeerServer`, `PeerRequestHandler`
Deregistrazione peer	`PEER_QUIT` su `quit`.	`PeerCli`, `ClientHandler`
CLI Master	`listdata`, `inspectNodes`, `log`, `quit`.	`MasterMain`
Log dei download	`DownloadLog` (OK/FAIL).	`DownloadLog`, `ClientHandler`
Messaggi d'errore chiari	Peer stampa errore se Master non raggiungibile.	`PeerMain`

9 Scelte progettuali e alternative

Lock unico vs ReadWriteLock (Master)

Scelta: lock unico per aderire rigorosamente alla specifica (nessuna lettura parallela durante aggiornamenti). Alternativa: ReadWriteLock (maggiore parallelismo in sola lettura) ma meno "stretta".

Validazione token lato peer

Non richiesta; possibile estensione: prima di servire `DOWNLOAD`, il peer valida il token chiedendo al Master. Pro: maggiore robustezza; Contro: overhead di round-trip.

Indice risorse filtrato

Per evitare indicizzazione di file scaricati (`downloaded_*`), si filtra lato peer in `ResourceManager.list()`.

10 Istruzioni di consegna (promemoria)

- Repo privato `LABSO <NOME GRUPPO>` su GitHub/GitLab.
- Aggiungere tutor come collaboratori (come da specifiche).
- Tag `Consegna` con messaggio non vuoto.
- Email di notifica con link repo e allegato `DOCUMENTAZIONE.pdf`.

11 Conclusioni

Il sistema soddisfa i requisiti della specifica:

- Registro centrale coerente, protetto da mutua esclusione.
- Peer server serializzato (Semaphore(1)).
- Download con token, retry robusto e log centralizzato.
- CLI utili per ispezione e diagnosi.

Estensioni possibili: validazione token lato peer, persistenza del registro su disco, policy avanzate di scelta del peer sorgente (round-robin, load-aware).

A Codice lato Master: classi, porzioni rilevanti e collegamenti

Tutte le classi del Master risiedono nel package `it.unibo.labso.p2p.master` e dipendono da DTO/utility comuni nel package `it.unibo.labso.p2p.common` (`PeerRef`, `Token`, `NetUtils`, `JsonCodec`). Di seguito il dettaglio per classe con estratti *illustrativi* del codice, collegamenti fra componenti e motivazioni progettuali basate sull'implementazione effettiva.

MasterMain.java

Punto di ingresso. Legge la porta (default 9000, sovrascrivibile dal primo argomento), avvia il server e presenta la CLI amministrativa minimale.

- Parsing robusto della porta con fallback e *usage* in caso di input non valido.
- REPL semplice: legge una riga, *switch* sul comando e delega a `MasterServer` tramite accessor.

```
int port = 9000;
if (args.length > 0) {
  try { port = Integer.parseInt(args[0]); }
  catch (NumberFormatException e) {
    System.err.println("usage: MasterMain <port>");
    return;
  }
}
MasterServer server = new MasterServer(port);
server.start();
while (true) {
  String cmd = in.readLine();
  if (cmd == null) break;
  switch (cmd) {
```

```
    case "listdata" -> System.out.println(server.registry().snapshotAll());
    case "inspectNodes" -> System.out.println(server.registry().snapshotPeers());
    case "log" -> System.out.println(server.downloadLog().snapshot());
    case "quit" -> { server.shutdown(); System.out.println("[MASTER] bye"); return;
        }
    default -> System.out.println("listdata | inspectNodes | log | quit");
  }
}
```

Scelta progettuale: la CLI vive nel `main` per tenerla indipendente dal protocollo e garantire visibilità immediata sullo stato interno (`Registry`, `DownloadLog`).

MasterServer.java

Accetta connessioni TCP e delega ogni socket a `ClientHandler` eseguito su un pool. Mantiene i componenti condivisi e fornisce accessor usati dalla CLI.

- Campi principali: `ServerSocket serverSocket`, `ExecutorService pool`, `Registry registry`, `TokenStore tokens`, `DownloadLog downloadLog`.
- `start()` avvia il ciclo di `accept()` e sottomette un `ClientHandler` per ogni connessione.
- `shutdown()` chiude ordinatamente socket e pool; `closeServerSocketQuietly()` gestisce eccezioni in chiusura.

```
/** Accetta connessioni e delega al ClientHandler. */
final class MasterServer {
  private final int port;
  private volatile boolean running;
  private ServerSocket serverSocket;
  private final ExecutorService pool = Executors.newCachedThreadPool();
  private final Registry registry = new Registry();
  private final TokenStore tokens = new TokenStore();
  private final DownloadLog downloadLog = new DownloadLog();

  Registry registry() { return registry; }
  TokenStore tokens() { return tokens; }
  DownloadLog downloadLog() { return downloadLog; }

  void start() throws IOException {
    running = true;
    serverSocket = new ServerSocket(port);
    while (running) {
      Socket s = serverSocket.accept();
      pool.submit(new ClientHandler(s, registry, tokens, downloadLog));
    }
  }

  void shutdown() {
    running = false;
    closeServerSocketQuietly();
    pool.shutdownNow();
  }
}
```

Scelta progettuale: server "sottile" e componibile; la logica del protocollo è tutta nel `ClientHandler`, lo stato condiviso resta centralizzato e facilmente ispezionabile.

ClientHandler.java

Implementa `Runnable` e gestisce il protocollo Master ↔ Peer: I/O line-based, payload JSON con `JsonCodec`. Interagisce con `Registry`, `TokenStore` e `DownloadLog`.

- Comandi: `REGISTER`, `LISTDATA_REMOTE`, `LIST_PEERS`, `WHO_HAS`, `DOWNLOAD_TOKEN_REQ`, `DOWNLOAD_TOKEN_REL`, `DOWNLOAD_FAILED`, `PEER_QUIT`, `QUIT`.
- DTO interni (record) per il parse del JSON: `RegisterReq`, `ListResp`, `PeersResp`, `WhoHasReq`, `TokenReq`, `TokenResp`, `TokenRel`, `DownloadFailedReq`, `PeerQuitReq`.
- Politiche: scelta del primo candidato da `Registry.whoHas(resource)`; token con TTL fisso; su `DOWNLOAD_FAILED` rimozione dell'associazione `peerId,resource` e tracciamento nel log.

```
switch (cmd) {
  case "REGISTER" -> {
    var req = JsonCodec.fromJson(json, RegisterReq.class);
    registry.upsertPeer(req.ref(), req.resources());
    NetUtils.sendLine(out, "OK");
  }
  case "WHO_HAS" -> {
    var req = JsonCodec.fromJson(json, WhoHasReq.class);
    var peers = registry.whoHas(req.resource());
    NetUtils.sendLine(out, "OK " + JsonCodec.toJson(new PeersResp(peers)));
  }
  case "DOWNLOAD_TOKEN_REQ" -> {
    var req = JsonCodec.fromJson(json, TokenReq.class);
    var candidates = registry.whoHas(req.resource());
    var chosen = candidates.isEmpty() ? null : candidates.get(0);
    if (chosen == null) { NetUtils.sendLine(out, "ERR NO_RESOURCE"); break; }
    var tok = tokens.issue(req.resource(), req.requesterPeerId(), Duration.
        ofSeconds(60));
    NetUtils.sendLine(out, "OK " + JsonCodec.toJson(new TokenResp(tok, chosen)));
  }
  case "DOWNLOAD_TOKEN_REL" -> {
    var rel = JsonCodec.fromJson(json, TokenRel.class);
    tokens.revoke(rel.token());
    downloadLog.addSuccess(rel.resource(), rel.fromPeerId(), rel.requesterPeerId())
        ;
    NetUtils.sendLine(out, "OK");
  }
  case "DOWNLOAD_FAILED" -> {
    var fail = JsonCodec.fromJson(json, DownloadFailedReq.class);
    registry.removePeerResource(fail.fromPeerId(), fail.resource());
    downloadLog.addFailure(fail.resource(), fail.fromPeerId(), fail.requesterPeerId
        ());
    NetUtils.sendLine(out, "OK");
  }
  case "PEER_QUIT" -> { /* deregistrazione */ }
```

```
  case "QUIT" -> { /* chiusura socket */ }
  default -> NetUtils.sendLine(out, "ERR UNKNOWN_CMD");
}
```

Scelte progettuali:

1. Line-based + JSON per un parsing semplice e estendibile.
2. Token con TTL per evitare riusi accidentali/malevoli; revoca esplicita su `DOWNLOAD_TOKEN_REL`.
3. Pulizia del registro su `DOWNLOAD_FAILED` per favorire il retry verso altri peer.

Registry.java

Registro coerente con lock unico (`ReentrantLock`) su tutte le operazioni, incluse le letture *snapshot*, per rispettare il vincolo di mutua esclusione anche in consultazione.

- Strutture: `Map<String, Set<String>> resourceToPeers`, `Map<String, PeerRef> peerIdToRef`, `Map<String, Set<String>> peerToResources`.
- Operazioni principali: `upsertPeer(PeerRef, Set<String>)`, `removePeer(String)`, `removePeerResource(String,String)`, `whoHas(String)`, `snapshotAll()`, `snapshotPeers()`.

```
/** Registro risorse <-> peer con mutua esclusione "stretta". */
final class Registry {
  private final Lock lock = new ReentrantLock();
  private final Map<String, Set<String>> resourceToPeers = new HashMap<>();
  private final Map<String, PeerRef> peerIdToRef = new HashMap<>();
  private final Map<String, Set<String>> peerToResources = new HashMap<>();

  void removePeerResource(String peerId, String resource) {
    lock.lock();
    try {
      var res = peerToResources.get(peerId);
      if (res != null) { res.remove(resource); if (res.isEmpty()) peerToResources.
          remove(peerId); }
      var ids = resourceToPeers.get(resource);
      if (ids != null) { ids.remove(peerId); if (ids.isEmpty()) resourceToPeers.
          remove(resource); }
    } finally { lock.unlock(); }
  }
}
```

Scelte progettuali:

- Lock unico (anziché `ReadWriteLock`) per garantire assenza di letture durante update, come richiesto.
- Mantenimento simmetrico degli indici *forward* e *reverse* per query e stampe immediate (usate dalla CLI).

TokenStore.java

Gestione token thread-safe con `ConcurrentHashMap`. Ogni token ha valore (UUID), risorsa, destinatario e scadenza.

- `issue(resource,toPeerId,ttl)` crea e indicizza il token.
- `get(value)` valida la scadenza; rimuove i token scaduti.
- `revoke(value)` rimuove il token al termine del download.

```
final class TokenStore {
  private final Map<String, Token> byValue = new ConcurrentHashMap<>();

  Token issue(String resource, String toPeerId, Duration ttl) {
    Token t = new Token(/* uuid, resource, toPeerId, expiresAt */);
    byValue.put(t.value(), t);
    return t;
  }

  Optional<Token> get(String value) {
    Token t = byValue.get(value);
    if (t == null) return Optional.empty();
    if (Instant.now().isAfter(t.expiresAt())) { byValue.remove(value); return
        Optional.empty(); }
    return Optional.of(t);
  }
  void revoke(String value) { byValue.remove(value); }
}
```

Scelta progettuale: `ConcurrentHashMap` evita lock espliciti sui path veloci, mantenendo semplice la semantica.

DownloadLog.java

Log append-only degli esiti con timestamp e peer coinvolti. `List<Entry>` sincronizzata; `snapshot()` espone una vista immutabile per la CLI.

```
final class DownloadLog {
  static final class Entry {
    final Instant ts = Instant.now();
    final String resource, fromPeerId, toPeerId;
    final boolean ok;
    public String toString() {
      return ts + " " + resource + " da:" + fromPeerId + " a:" + toPeerId
            + " esito:" + (ok ? "OK" : "FAIL");
    }
  }
  private final List<Entry> entries =
      Collections.synchronizedList(new ArrayList<>());
  void addSuccess(String r, String from, String to) { entries.add(new Entry(r, from
      , to, true)); }
  void addFailure(String r, String from, String to) { entries.add(new Entry(r, from
      , to, false)); }
  List<Entry> snapshot() { return List.copyOf(entries); }
}
```

Scelta progettuale: lista sincronizzata sufficiente per append concorrente; `List.copyOf` evita esposizione di stato mutabile.

MasterCli.java

Wrapper minimale che stampa un promemoria dei comandi; la logica del REPL resta nel `main`.

```
public class MasterCli implements Runnable {
  @Override public void run() {
    System.out.println("master> listdata | inspectNodes | log | quit");
  }
}
```

Collegamenti e flusso complessivo

1. `MasterMain` avvia `MasterServer` e presenta la CLI.
2. `MasterServer` accetta connessioni e istanzia `ClientHandler`, passando `Registry`, `TokenStore`, `DownloadLog`.
3. `ClientHandler` legge righe, fa il parse JSON con `JsonCodec`, invoca `Registry`/`TokenStore` e risponde con `OK ...` o `ERR ...`.
4. `DownloadLog` traccia successi/fallimenti; la CLI legge `server.downloadLog().snapshot()`.

Mappa comandi → componenti

Comando	Componenti coinvolti
`REGISTER`	ClientHandler → Registry.upsertPeer
`LISTDATA_REMOTE`	ClientHandler → Registry.snapshotAll
`LIST_PEERS`	ClientHandler → Registry.snapshotPeers
`WHO_HAS`	ClientHandler → Registry.whoHas
`DOWNLOAD_TOKEN_REQ`	ClientHandler → Registry.whoHas, TokenStore.issue
`DOWNLOAD_TOKEN_REL`	ClientHandler → TokenStore.revoke, DownloadLog.addSuccess
`DOWNLOAD_FAILED`	ClientHandler → Registry.removePeerResource, DownloadLog.addFailure
`PEER_QUIT`	ClientHandler → Registry.removePeer

B Codice lato Peer: classi, porzioni rilevanti e collegamenti

Le classi del peer risiedono nel package `it.unibo.labso.p2p.client` (o equivalente nel tuo progetto) e collaborano tra loro per: mantenere le risorse locali, esporre un piccolo server TCP serializzato (`Semaphore(1)`), comunicare con il Master per scoprire i sorgenti e ottenere i token, e fornire una CLI utente. Di seguito dettaglio per classe, con estratti illustrativi, collegamenti e motivazioni progettuali.

PeerMain.java

Punto di ingresso del peer. Legge host/porta del Master dalla riga di comando, costruisce i componenti e avvia `PeerServer` in un thread dedicato, quindi entra nella CLI.

- Argomenti attesi: `<masterHost> <masterPort>`.

- Inizializza `ResourceManager` (cartella dati locale), `MasterClient` per parlare col Master, `PeerServer` per servire file in ingresso, `PeerCli` per i comandi interattivi.

```
public static void main(String[] args) throws Exception {
  if (args.length != 2) { System.err.println("usage: PeerMain <masterHost> <
      masterPort>"); return; }
  String masterHost = args[0]; int masterPort = Integer.parseInt(args[1]);

  ResourceManager rm = new ResourceManager(Path.of("./data/peer"));
  MasterClient master = new MasterClient(masterHost, masterPort);
  PeerServer server = new PeerServer(rm /*, ownPort auto/bind */);
  new Thread(server, "peer-server").start();

  // registra le risorse iniziali presso il Master
  master.register(rm.localPeerRef(), rm.listResources());

  // avvia la CLI
  new PeerCli(rm, master, server, new PeerClient(rm, master)).run();
}
```

Scelta progettuale: *composition root* nel `main` per costruire e cablare tutti i componenti in modo esplicito; `PeerServer` su thread dedicato per non bloccare la CLI.

PeerServer.java

Piccolo server TCP che serve una sola richiesta alla volta, come da requisito. Trasferisce byte in modo deterministico dopo aver validato il token.

- Campi principali: `ServerSocket server`, `Semaphore gate = new Semaphore(1)`, `ResourceManager rm`, porta di ascolto (espone `PeerRef` con host/porta reali).
- `run()`: ciclo `accept()`; per ogni socket sottomette a `PeerRequestHandler` la gestione della singola richiesta *dopo* aver acquisito la `Semaphore(1)`.

```
/** Server del peer: serve una richiesta alla volta (Semaphore(1)). */
final class PeerServer implements Runnable {
  private final ServerSocket server;
  private final Semaphore gate = new Semaphore(1);
  private final ResourceManager rm;

  public void run() {
    while (!server.isClosed()) {
      try {
        Socket s = server.accept();
        // serializza: una sola richiesta alla volta
        gate.acquire();
        new Thread(new PeerRequestHandler(s, rm, gate), "peer-req").start();
      } catch (IOException ignored) {}
    }
  }
}
```

Scelta progettuale: `Semaphore(1)` garantisce mutua esclusione forte sull'erogazione dei contenuti, come richiesto, evitando interleaving di stream.

PeerRequestHandler.java

Gestisce il singolo dialogo Peer → Peer. Valida il formato del comando `DOWNLOAD <nome> <token>`, controlla l'esistenza della risorsa e invia `OK <size>` seguito da esattamente `size` byte.

- Alla fine rilascia sempre la `Semaphore` passata dal server, per sbloccare la prossima richiesta.
- In caso di errore, risponde con `ERR <motivo>` e chiude la socket.

```
final class PeerRequestHandler implements Runnable {
  private final Socket s;
  private final ResourceManager rm;
  private final Semaphore gate;
  public void run() {
    try (var in = new BufferedReader(new InputStreamReader(s.getInputStream(),
        StandardCharsets.UTF_8));
         var out = new BufferedWriter(new OutputStreamWriter(s.getOutputStream(),
             StandardCharsets.UTF_8));
         var raw = s.getOutputStream()) {

      String line = in.readLine(); // es. "DOWNLOAD R1.txt <token>"
      var parts = line.split("\\s+");
      if (parts.length != 3 || !"DOWNLOAD".equals(parts[0])) { out.write("ERR
          MALFORMED\n"); out.flush(); return; }

      String name = parts[1], token = parts[2];
      if (!rm.exists(name)) { out.write("ERR NO_RESOURCE\n"); out.flush(); return;
          }
      long size = rm.size(name);
      out.write("OK " + size + "\n"); out.flush();

      try (var file = rm.newInputStream(name)) { file.transferTo(raw); }
    } catch (IOException e) {
      // log locale opzionale
    } finally { gate.release(); try { s.close(); } catch (IOException ignored) {} }
  }
}
```

Scelte progettuali: protocollo line-based minimale; header `OK <size>` per consentire al client di ricevere esattamente i byte attesi; rilascio `gate` in `finally` per evitare deadlock.

ResourceManager.java

Astrazione del filesystem locale del peer: crea/legge file, espone dimensioni e lista delle risorse, costruisce il `PeerRef` (host, porta) da registrare sul Master.

- `add(name, bytes)` crea/aggiorna file.
- `exists(name)`, `size(name)`, `newInputStream(name)` per il serving.
- `listResources()` restituisce l'elenco per `REGISTER`.

```
final class ResourceManager {
  private final Path root;
  ResourceManager(Path root) { this.root = root; Files.createDirectories(root); }
```

```
  boolean exists(String name) { return Files.isRegularFile(root.resolve(name)); }
  long size(String name) { return Files.size(root.resolve(name)); }
  InputStream newInputStream(String name) { return Files.newInputStream(root.
      resolve(name)); }
  List<String> listResources() { try (var s = Files.list(root)) {
    return s.filter(Files::isRegularFile).map(p -> p.getFileName().toString()).
        toList();
  } }
}
```

Scelta progettuale: incapsulare I/O su disco per separare la logica di rete dalla gestione file, rendendo testabili i componenti.

MasterClient.java

Client lato peer verso il Master. Implementa i comandi del protocollo: `REGISTER`, `LISTDATA_REMOTE`, `LIST_PEERS`, `WHO_HAS`, `DOWNLOAD_TOKEN_REQ`, `DOWNLOAD_TOKEN_REL`, `DOWNLOAD_FAILED`, `PEER_QUIT`.

- `register(PeerRef, List<String>)` invia `REGISTER` con payload JSON.
- `whoHas(resource)` ritorna la lista dei candidati `PeerRef`.
- `requestToken(resource, requesterPeerId)` restituisce `Token` e `PeerRef` scelto.
- `releaseToken(...)` e `notifyFailed(...)` tracciano esito al Master.

```
TokenResp requestToken(String resource, String requesterPeerId) throws IOException
    {
  var req = new TokenReq(resource, requesterPeerId);
  NetUtils.sendLine(out, "DOWNLOAD_TOKEN_REQ " + JsonCodec.toJson(req));
  String line = in.readLine();
  if (!line.startsWith("OK ")) throw new IOException(line);
  return JsonCodec.fromJson(line.substring(3), TokenResp.class);
}
```

Scelta progettuale: centralizzare qui il protocollo verso il Master mantiene `PeerClient` privo di logica di controllo "globale".

PeerClient.java

Implementa il download attivo: data una risorsa, chiede al Master il token, si connette al peer sorgente, verifica l'header `OK <size>` e riceve i byte, con retry su errore.

- `download(resource)`: ciclo retry su lista candidati (se fallisce un candidato, invia `DOWNLOAD_FAILED` e prova il successivo).
- Su successo: salva su disco e invia `DOWNLOAD_TOKEN_REL`.

```
Path download(String resource) throws IOException {
  // 1) token + peer sorgente scelto dal Master
  var resp = master.requestToken(resource, rm.localPeerId());
  var token = resp.token().value(); var src = resp.sourceRef();

  try (var s = new Socket(src.host(), src.port());
       var in = new BufferedReader(new InputStreamReader(s.getInputStream(),
           StandardCharsets.UTF_8));
       var out = new BufferedWriter(new OutputStreamWriter(s.getOutputStream(),
           StandardCharsets.UTF_8));
```

```
      var raw = s.getInputStream()) {

    out.write("DOWNLOAD " + resource + " " + token + "\n"); out.flush();
    String header = in.readLine(); // "OK <size>"
    if (header == null || !header.startsWith("OK ")) throw new IOException("bad
        header");
    long size = Long.parseLong(header.substring(3).trim());

    Path dst = rm.resolve(resource);
    try (var file = Files.newOutputStream(dst, StandardOpenOption.CREATE,
        StandardOpenOption.TRUNCATE_EXISTING)) {
      raw.transferTo(file);
    }
    master.releaseToken(resource, src.peerId(), rm.localPeerId(), token);
    return dst;
  } catch (IOException e) {
    master.notifyFailed(resource, src.peerId(), rm.localPeerId());
    throw e;
  }
}
```

Scelte progettuali:

1. Verifica header e dimensione per ricezione deterministica.
2. Retry guidato dal Master: su fallimento si "pulisce" il registro (`DOWNLOAD_FAILED`) e si tenta un altro candidato.
3. Separazione netta: `MasterClient` per il controllo globale, `PeerClient` per il trasferimento dati.

PeerCli.java

Interfaccia testuale interattiva per l'utente del peer. Fornisce i comandi richiesti dalla specifica.

- `listdata local`: elenca i file locali (`ResourceManager.listResources()`).
- `listdata remote`: stampa l'indice globale (`MasterClient.listdataRemote()`).
- `listpeers`, `whohas <nome>`, `add <nome> <contenuto>`, `download <nome>`, `quit`.

```
switch (cmd[0]) {
  case "listdata" -> {
    if ("local".equals(cmd[1])) rm.listResources().forEach(System.out::println);
    else if ("remote".equals(cmd[1])) System.out.println(master.listdataRemote());
  }
  case "whohas" -> System.out.println(master.whoHas(cmd[1]));
  case "add" -> { rm.add(cmd[1], cmd[2].getBytes(StandardCharsets.UTF_8));
                  master.register(rm.localPeerRef(), rm.listResources()); }
  case "download" -> { Path p = peerClient.download(cmd[1]); System.out.println("OK
      " + p); }
  case "quit" -> { master.peerQuit(rm.localPeerId()); server.shutdown(); return; }
  default -> System.out.println("listdata local|remote | listpeers | whohas <r> |
      add <r> <txt> | download <r> | quit");
}
```

Scelta progettuale: la CLI aggiorna il Master dopo `add` per mantenere l'indice coerente; `quit` invia `PEER_QUIT` e arresta il `PeerServer`.

Collegamenti e flusso complessivo lato peer

1. `PeerMain` costruisce `ResourceManager`, `MasterClient`, `PeerServer`, `PeerClient` e avvia la `PeerCli`.
2. `PeerServer` serializza le richieste in ingresso con `Semaphore(1)` e usa `PeerRequestHandler` per spedire i byte.
3. `PeerClient` usa `MasterClient` per ottenere token e sorgente; scarica i dati e notifica esito (`DOWNLOAD_TOKEN_REL` o `DOWNLOAD_FAILED`).
4. `PeerCli` orchestra i comandi utente, aggiornando il Master dove necessario (`REGISTER`, `PEER_QUIT`).

Mappa comandi CLI → componenti

Comando	Componenti coinvolti
`listdata local`	PeerCli → ResourceManager.listResources
`listdata remote`	PeerCli → MasterClient.listdataRemote
`listpeers`	PeerCli → MasterClient.listPeers
`whohas <nome>`	PeerCli → MasterClient.whoHas
`add <nome> <txt>`	PeerCli → ResourceManager.add; MasterClient.register
`download <nome>`	PeerCli → PeerClient.download; MasterClient.releaseToken/notifyFailed
`quit`	PeerCli → MasterClient.peerQuit; PeerServer.shutdown

Rischi, limiti e alternative considerate

- **Serializzazione server**: `Semaphore(1)` rispetta il requisito "una richiesta alla volta", limitando throughput; alternativa: coda con back-pressure e finestra di 1 richiesta (semantica identica) o validazione token lato peer con round-trip al Master.
- **Retry lato client**: oggi *best-effort* sul primo candidato dal Master; alternative: round-robin sui candidati o politica LRU.
- **Header fisso `OK <size>`**: semplice e interoperabile; alternativa: framing binario con length-prefix su 4 byte, ma non necessario per la specifica.

C Codice comune e utility: strutture dati, codec e I/O di rete

Questa sezione documenta le classi condivise tra Master e Peer, responsabili di serializzazione JSON, messaggistica, utilità di rete e descrizione dei peer. Le decisioni progettuali puntano a mantenere un protocollo line-based semplice, con payload JSON tipizzati e strutture dati minime ma esplicite.

PeerRef.java e PeerInfo.java

`PeerRef` descrive come contattare un peer: `peerId`, `host`, `port`. `PeerInfo` (se presente) incapsula metadati non strettamente necessari alla connessione (per esempio stato o ultimo seen).

```
/** Riferimento contattabile a un peer. */
public final class PeerRef {
  private final String peerId;
  private final String host;
  private final int port;

  public String peerId() { return peerId; }
  public String host() { return host; }
  public int port() { return port; }

  @Override public String toString() {
    return peerId + " " + host + ":" + port;
  }
}
```

Collegamenti:

- Nel Master: `Registry` mantiene mappe `peerId` → `PeerRef` e `resource` → {`peerId`}.
- Nel Peer: `MasterClient` scambia `PeerRef` per `REGISTER`, `WHO_HAS` e `DOWNLOAD_TOKEN_REQ`.

Scelta: `PeerRef` immutabile, adatto a snapshot e a essere serializzato in JSON.

Token.java

Rappresenta il permesso temporaneo a scaricare una risorsa, con valore (UUID), risorsa associata, destinatario e scadenza.

```
public final class Token {
  private final String value; // UUID
  private final String resource; // nome risorsa
  private final String toPeerId; // destinatario
  private final Instant expiresAt; // scadenza

  public String value() { return value; }
  public String resource() { return resource; }
  public String toPeerId() { return toPeerId; }
  public Instant expiresAt(){ return expiresAt; }

  @Override public String toString() { return value; }
}
```

Collegamenti:

- Emesso/validato/revocato da `TokenStore`.
- Trasportato in `TokenResp` tra Master e Peer tramite `JsonCodec`.

Scelta: modello minimale e immutabile, favorisce validazioni semplici e passaggio sicuro tra thread.

Message.java

Convenzioni per i payload JSON del protocollo; tipicamente record/dto interni a `ClientHandler` e a `MasterClient`. In alcuni casi il progetto definisce contenitori generici (per esempio `ListResp`, `PeersResp`, `TokenReq`, `TokenResp`, ecc.).

```
// Esempi di DTO/record per il payload JSON.
public record RegisterReq(PeerRef ref, List<String> resources) {}
public record WhoHasReq(String resource) {}
public record PeersResp(List<PeerRef> peers) {}
public record TokenReq(String resource, String requesterPeerId) {}
public record TokenResp(Token token, PeerRef sourceRef) {}
public record TokenRel(String token, String resource, String fromPeerId, String
    requesterPeerId) {}
public record DownloadFailedReq(String resource, String fromPeerId, String
    requesterPeerId) {}
```

Collegamenti:

- Master: `ClientHandler` usa questi dto per il parse dei payload e la costruzione delle risposte.
- Peer: `MasterClient` li costruisce e li decodifica su risposta del Master.

Scelta: record Java per verbosità ridotta, immutabilità e serializzazione diretta.

JsonCodec.java

Adapter centralizzato per serializzare/deserializzare i DTO in JSON, astratto dalla libreria concreta (Jackson/Gson).

```
public final class JsonCodec {
  private static final ObjectMapper M = new ObjectMapper();

  public static String toJson(Object o) {
    try { return M.writeValueAsString(o); }
    catch (Exception e) { throw new IllegalArgumentException("json-encode", e); }
  }

  public static <T> T fromJson(String s, Class<T> type) {
    try { return M.readValue(s, type); }
    catch (Exception e) { throw new IllegalArgumentException("json-decode", e); }
  }
}
```

Collegamenti:

- Master: `ClientHandler` decodifica richieste e codifica risposte `OK <json>`.
- Peer: `MasterClient` costruisce richieste e decodifica le risposte del Master.

Scelte:

- Centralizzare qui evita dipendenze dirette dalla libreria JSON in tutto il codice.
- Eccezione unchecked per semplificare i flussi line-based e fallire veloce su input malformati.

NetUtils.java

Utility I/O per protocollo line-based su socket, con charset fissato a UTF-8 e *auto-flush* esplicito.

```
public final class NetUtils {
  private static final Charset CS = StandardCharsets.UTF_8;

  public static void sendLine(Writer out, String line) throws IOException {
    out.write(line); out.write("\n"); out.flush();
  }

  public static BufferedReader newReader(InputStream in) {
    return new BufferedReader(new InputStreamReader(in, CS));
  }

  public static BufferedWriter newWriter(OutputStream out) {
    return new BufferedWriter(new OutputStreamWriter(out, CS));
  }
}
```

Collegamenti:

- Master: `ClientHandler` invia `OK ...` o `ERR ...`.
- Peer: `PeerClient` e `PeerRequestHandler` scambiano header e comandi.

Scelte:

- Separare `Reader`/`Writer` dall'`InputStream` crudo usato per i byte del file.
- `flush()` esplicito dopo ogni linea per evitare deadlock con buffer residui.

Integrazione tra i moduli e contratti di validazione

Contratti lato Master

- Ogni risposta usa il formato `OK <json>` o `ERR <motivo>`.
- `TokenStore.get(value)` invalida i token scaduti; `DOWNLOAD_TOKEN_REL` revoca sempre il token.
- `Registry.removePeerResource(peerId, resource)` viene invocato su `DOWNLOAD_FAILED` per mantenere coerenti gli indici.

Contratti lato Peer

- `PeerClient` verifica l'header `OK <size>` prima di ricevere il flusso binario.
- `PeerServer` serve una sola richiesta alla volta (`Semaphore(1)`); `PeerRequestHandler` rilascia la semafora in `finally`.
- Dopo `add` o modifica risorse, la CLI invia `REGISTER` per aggiornare il Master.

Motivazioni complessive

- Line-based + JSON: parsing semplice, estendibile, adatto a log e debug.
- Modelli immutabili (`PeerRef`, `Token`, DTO record): riducono bug di concorrenza e facilitano snapshot.
- I/O separato testo/binario: header umani e trasferimento byte-preciso.

D Appendice A: Esempi di comandi

Master

```
listdata
inspectNodes
log
quit
```

Peer

```
listdata local
add R3.txt hello
listdata remote
listpeers
whohas R3.txt
download R3.txt
quit
```

E Appendice B: Suggerimenti di debug

- Se `mvn clean` fallisce: chiudere tutti i `java.exe`, poi ripetere.
- Se `REGISTER` fallisce: controllare che il Master risponda `OK` (anche senza payload).
- Se `download` resta appeso: controllare che il peer sorgente sia avviato e che risponda `OK <size>`.
- Per simulare concorrenza: avviare due download in parallelo verso lo stesso peer.