SRS - Peer-to-Peer Distributed File System

Russo Antonio

2025-09-26

Contents

1 Introduzione

1.1 Scopo

Questo documento definisce i requisiti del sistema di File System Distribuito Peer-to-Peer (P2P FS). Il sistema consente la gestione di file e directory su

più nodi collegati in rete, con accesso trasparente tramite API remote. Ogni nodo (peer) agisce sia come server che come client.

1.2 Obiettivi

- Accesso uniforme ai file, sia locali che remoti, tramite RMI.
- Replica e condivisione dei dati tra peer.
- Operazioni tipiche di un file system: creazione, lettura, scrittura, cancellazione, link simbolici e gestione attributi.
- Robustezza: il fallimento di un peer non blocca l'intera rete.

1.3 Attori

- **Utente**: interagisce tramite il client a riga di comando `FileSystemClient`.
- **Peer**: nodo che ospita un'istanza di file system distribuito e un server RMI.
- **Rete P2P**: insieme dei peer connessi che replicano ed espongono le API.

2 Descrizione generale

2.1 Architettura

Il sistema adotta un modello a **grafo completo** per rappresentare la rete dei peer. Ogni nodo mantiene una connessione diretta con tutti gli altri peer della rete. In questo modo, le operazioni di ricerca e inoltro richieste non richiedono salti multipli, ma ogni nodo può interrogare direttamente i suoi vicini.

2.1.1 Vantaggi

- **Bassa latenza**: le richieste raggiungono il peer interessato in un solo hop.
- **Semplicità di implementazione**: non servono algoritmi complessi di routing. direttamente tra loro senza ricalcolare percorsi alternativi.

2.1.2 Svantaggi

- **Scalabilità limitata**: ogni nuovo peer deve stabilire connessioni con tutti gli altri, portando a un numero di connessioni pari a $O(n^2)$.
- **Overhead di gestione**: l'aggiunta o rimozione di un peer comporta l'aggiornamento delle tabelle dei vicini in tutti i nodi.
- **Consumo di memoria e risorse**: all'aumentare del numero di peer crescono le risorse necessarie per mantenere la lista dei vicini.

2.2 Peer

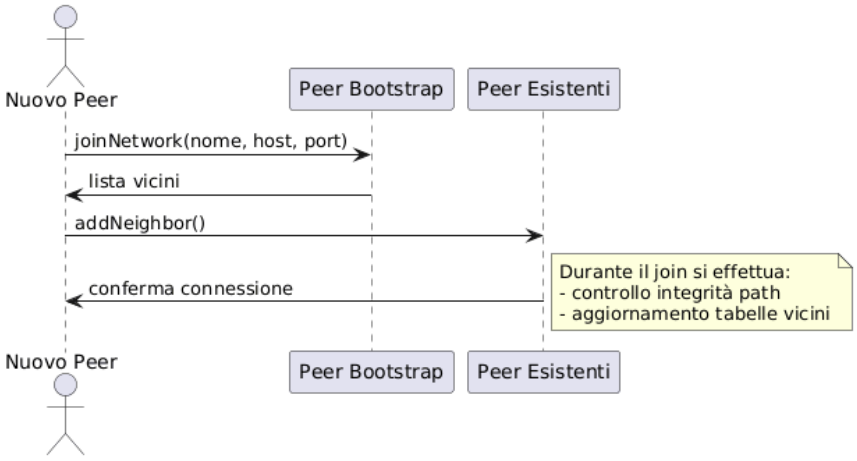
- Ogni peer possiede un `FileSystem` montato su una root reale del disco.
- Il peer espone un server RMI (`FileSystemServer`) che pubblica un oggetto remoto (`RemoteFileSystem`) conforme a `FileSystemInterface`.
- I client (`FileSystemClient`) invocano operazioni sui peer remoti come se fossero locali.

2.2.1 Inserimento di un nuovo peer

L'aggiunta di un nuovo nodo avviene tramite la procedura `joinNetwork`:

1. Il nuovo peer contatta un nodo bootstrap noto (specificato all'avvio).
2. Il bootstrap fornisce al nuovo nodo la lista dei vicini esistenti.
3. Il nuovo peer stabilisce connessioni con ciascun nodo della lista.
4. In parallelo, ciascun vicino aggiorna la propria tabella aggiungendo il nuovo peer.
5. Durante il join viene effettuato un **controllo di integrità**: si verifica che non ci siano conflitti di path già esistenti nei file system dei nodi coinvolti.
6. Se il controllo ha successo, il nodo viene accettato e partecipa a pieno titolo al grafo completo.

In questo modo, la rete rimane sempre connessa e coerente, garantendo che ogni operazione possa essere inoltrata a qualsiasi nodo remoto senza ambiguità.



2.3 Vincoli

- Linguaggio: Java 17+
- Comunicazione: Java RMI
- Persistenza: write-through su file system locale
- Concorrenza: lock tramite `ReentrantReadWriteLock`
- Sicurezza: protezione da path traversal

3 Requisiti funzionali

3.1 Operazioni principali (API esposte)

3.1.1 Creazione

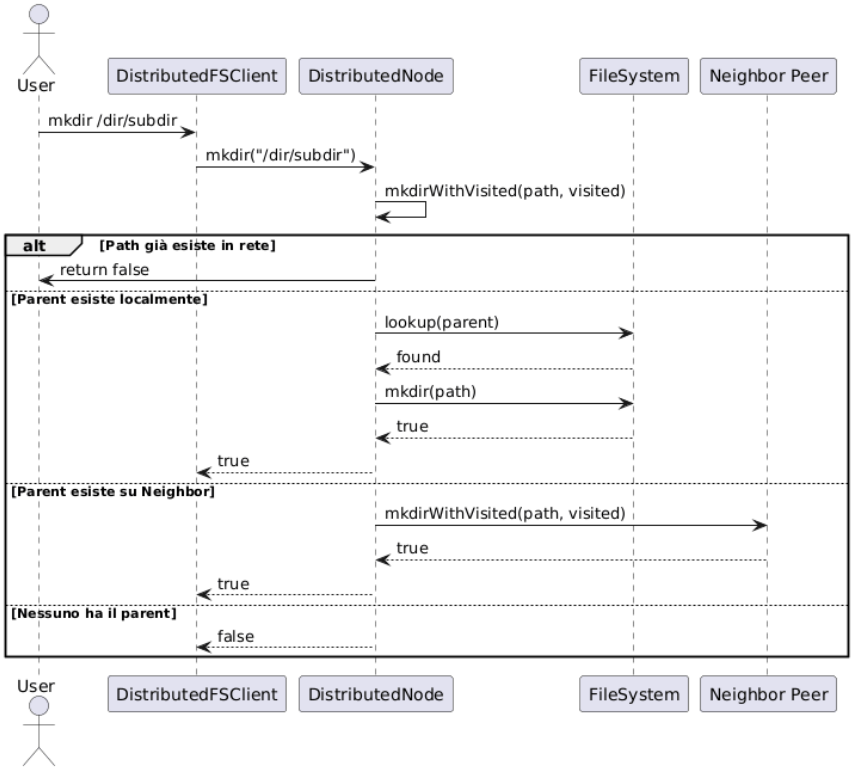
Le operazioni di creazione seguono il seguente pattern

1. Il client remoto invia una richiesta remota (es. mknod /path/$\text{file}_{\text{name}}$)
2. Se il path è presente localmente allora il file viene creato utilizzando la primitiva offerta dal file system locale.
3. Se il path non esiste localmente allora la richiesta viene inoltrata ricorsivamente ad un peer remoto.
4. FileSystem acquisisce il lock, aggiorna lo stato in memoria e riflette la modifica su disco locale del peer specificato.

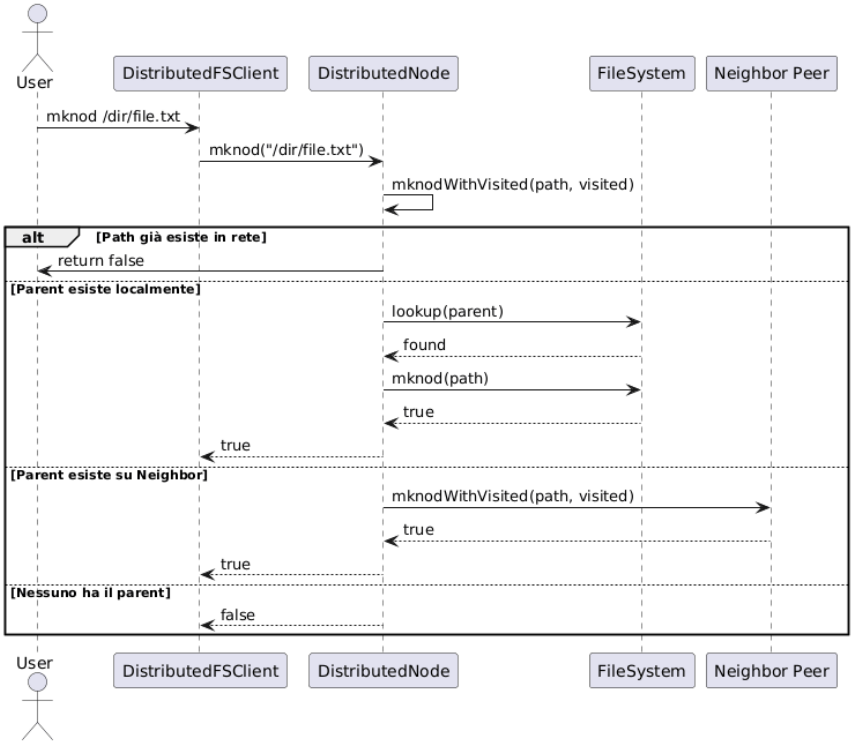
5. Il risultato viene restituito al client.

Questa logica garantisce che ogni operazione possa essere eseguita in maniera trasparente, sia che il path risieda in locale che in remoto.

- `mkdir(path)`: Creazione di una directory



- `mknod(path)`: Creazione di un file



1. Consistenza la consistenza è garantita grazie ai seguenti passi
   (a) Verifica che il nodo da creare (directory, file, symlink) non siano già presenti in locale
      i. Creo il nodo
   (b) Controllo di integrità `globale` del path (il nome file non deve essere presente in nessun peer con lo stesso path)
      i. Se il path già esiste allora genero un errore specificando che il file/dir esiste già nel path specificato
   (c) Se il path (parent) esiste localmente allora gli dò prioorità e lo creo
   (d) Se il parent esiste in remoto → delego la creazione al peer che lo possiede
   (e) Se nessuno ha il parent allora l'operazione fallisce con la generazione di un errore

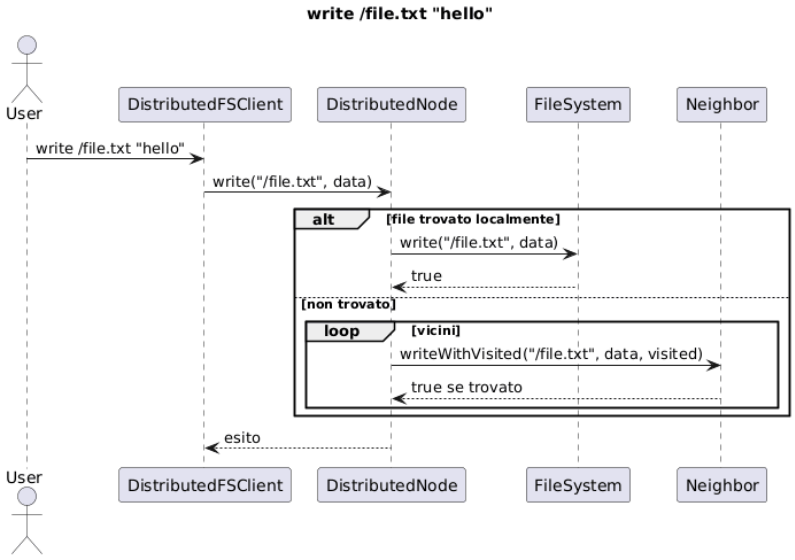
3.1.2 Manipolazione

Le operazioni di manipolazione (scrittura, lettura, rinomina, ecc.) seguono tutte un flusso generale comune, basato su tre fasi principali:

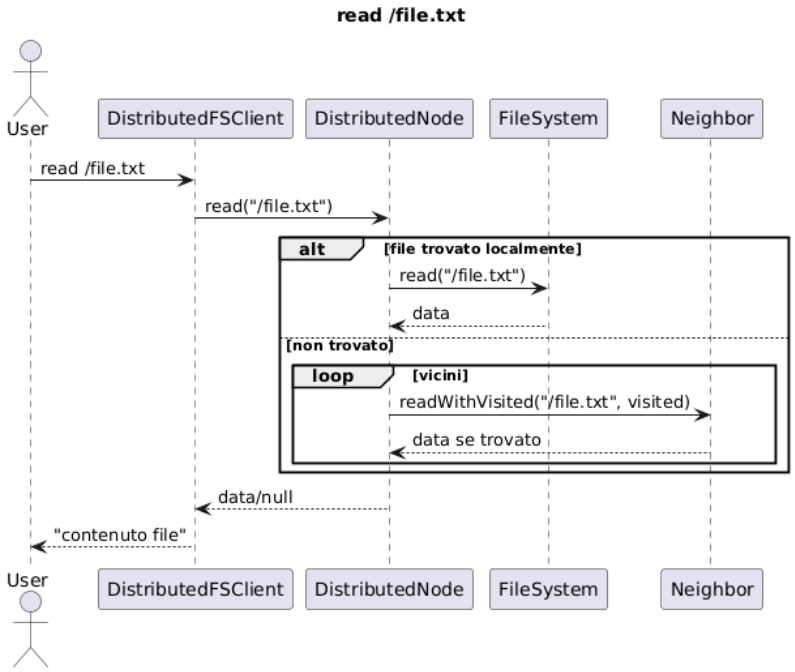
1. **Controllo locale** Ogni peer controlla prima se il path richiesto è presente nel proprio `FileSystem` locale.
    - Se il file o la directory esiste, l'operazione viene eseguita direttamente in locale, sfruttando i meccanismi di lock e il write-through su disco.
    - Le scritture avvengono in maniera atomica e consistente grazie a `ReentrantReadWriteLock`, garantendo che le letture concorrenti non entrino in conflitto.
2. **Inoltro ai vicini (propagazione remota)** Se il path non è presente localmente, il peer inoltra la richiesta ai vicini conosciuti. Ogni chiamata include una lista **visited** per evitare cicli infiniti: se un peer è già stato visitato nella catena della richiesta, la invocazione viene ignorata. In questo modo la ricerca termina sempre, anche in presenza di cicli.
3. **Risoluzione e risposta**
    - Se un peer remoto trova il path, esegue l'operazione e restituisce il risultato (contenuto letto, conferma di scrittura, lista directory, ecc.).
    - Se nessun peer possiede il path, l'operazione fallisce e viene restituito un valore di errore.

Questa logica garantisce che ogni operazione possa essere eseguita in maniera trasparente, sia che il path risieda in locale che in remoto.

- `write(path, content)`: scrive su file (locale o remoto)

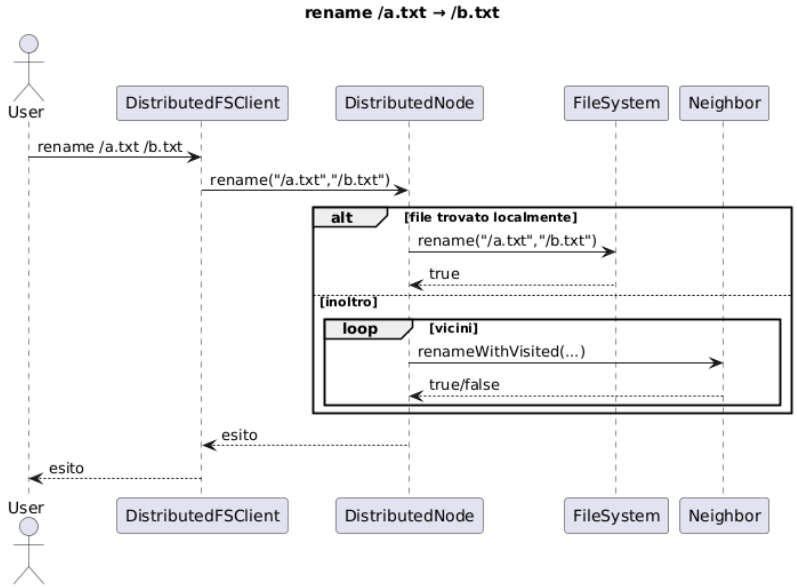


- `read(path)`: legge contenuto file (locale o remoto)



- `rename(oldPath, newPath)`: rinomina file o directory (locale o re-

moto).



1. Consistenza La consistenza per le operazioni di manipolazione sono garantite a livello di file system.

3.1.3 Gestione Attributi

- `readdir(path)`: lista contenuto directory (locale o remoto)
- `getattr(path)`: metadati (locale o remoto)
- `setattr(path, attr, value)`: modifica attributi

3.2 Funzionalità Client

- Interprete comandi interattivo (mkdir, ls, read, write).

4 Requisiti non funzionali

- Disponibilità: resilienza alla caduta di nodi.
- Trasparenza: uniformità tra file locali e remoti.
- Coerenza: propagazione aggiornamenti ai peer.