Distributed Virtual File System (DVFS)

Russo Antonio

2025-09-22

Contents

1 Introduzione

Il **Distributed Virtual File System (DVFS)** è un progetto che implementa un file system distribuito secondo un modello **client-server**. L'idea di base è permettere a più client di accedere a un file system remoto come se fosse locale, con un'interfaccia semplice e coerente. Il sistema è sviluppato in **Java** ed utilizza **RMI (Remote Method Invocation)** come meccanismo di comunicazione, garantendo trasparenza delle invocazioni e modularità.

Il DVFS offre funzionalità classiche di un file system (creazione, lettura, scrittura, navigazione) e introduce una politica di **write-through**, che assicura che ogni modifica in memoria venga immediatamente riflessa anche sul file system reale montato sul server.

2 Architettura

L'architettura segue il modello **client-server centralizzato**:

- **FileSystem**: cuore del sistema, un file system virtuale in memoria strutturato come un albero. Ogni nodo può rappresentare directory, file o symlink. Le operazioni in memoria vengono sincronizzate su disco tramite write-through.
- **RemoteFileSystem**: oggetto RMI che funge da "ponte" tra i client e il VFS locale. Implementa l'interfaccia remota e inoltra le richieste al FileSystem.
- **FileSystemServer**: avvia e monta il VFS da una directory reale, pubblica lo stub RMI e resta in ascolto delle richieste.
- **FileSystemClient**: applicazione a riga di comando che permette di interagire col file system remoto. Supporta comandi familiari (**mkdir**, **ls**, **read**, **write**) e funzionalità avanzate come **edit**, che scarica un file remoto in un editor locale e lo risincronizza al termine della modifica.

Questa separazione isola le responsabilità: i client gestiscono l'interazione con l'utente, mentre il server centralizza la logica del file system e garantisce consistenza tra più richieste concorrenti.

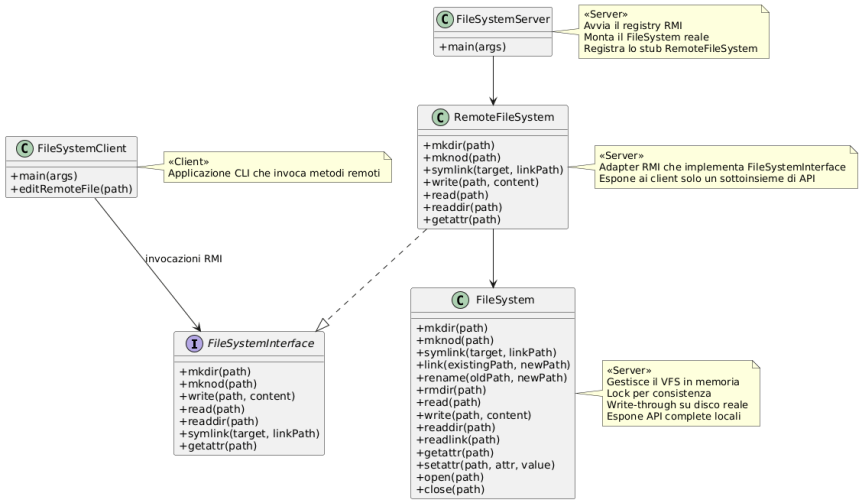

Figure 1: Architettura client-server DVFS

3 Consistenza

La consistenza è garantita dal **server**. Ogni operazione che modifica lo stato (scrittura, rinomina, rimozione) viene protetta da lock a livello di path (**ReentrantReadWriteLock**).

- Più client possono leggere contemporaneamente senza conflitti.
- Le scritture sono serializzate, impedendo **race condition**.
- Ogni modifica avviene in due fasi: aggiornamento in memoria e write-through su disco.

I client non gestiscono lock: tutta la concorrenza viene risolta dal server, che possiede l'unica copia "autorevole" dello stato.

4 Montaggio da directory reale

Il sistema può partire da zero o essere montato da una directory esistente. In questo caso, il contenuto viene caricato ricorsivamente:

- Directory → DirectoryNode.
- File → FileNode (contenuto letto in memoria).

- Symlink → SymlinkNode (target salvato).

La root del VFS viene rinominata "/", e ogni operazione successiva (scrittura, rinomina, rimozione) viene riflessa anche sulla directory reale tramite write-through.

5 Requisiti Funzionali

Il Distributed Virtual File System (DVFS) mette a disposizione un set completo di operazioni che riproducono le funzionalità tipiche di un file system UNIX, garantendo trasparenza tra accesso locale e remoto. Le API sono organizzate in cinque categorie principali:

5.1 Creazione

Le operazioni di creazione consentono di introdurre nuove entità nel file system:

- `mkdir(path)`: crea una nuova directory all'interno del percorso specificato.
- `mknod(path)`: crea un nuovo file vuoto, pronto per essere scritto.
- `symlink(target, linkPath)`: crea un link simbolico che punta ad un file o directory esistente.

Queste operazioni sono tutte soggette a controlli di integrità: prima della creazione viene verificato che il path non esista già e che il parent directory sia presente.

5.2 Navigazione

Le operazioni di navigazione permettono all'utente o al client di esplorare la struttura del file system:

- `lookup(path)`: risolve un percorso e restituisce il nodo corrispondente se esiste.
- `readdir(path)`: restituisce la lista di file e directory contenuti in una directory.
- `readlink(path)`: nel caso di un symlink, restituisce il target a cui il link punta.

Queste operazioni non modificano lo stato del sistema e sono protette da lock di lettura concorrente.

5.3 Manipolazione

Le operazioni di manipolazione permettono la modifica dello stato del file system:

- `read(path)`: legge il contenuto di un file e restituisce i byte.
- `write(path, content)`: scrive dati in un file, sovrascrivendo il contenuto precedente. È garantita la consistenza tramite lock e write-through su disco.
- `rename(oldPath, newPath)`: rinomina un file o directory, spostandolo eventualmente in un'altra directory.
- `rmdir(path)`: rimuove una directory vuota.

Queste operazioni sono serializzate tramite lock a livello di path per prevenire race condition. La scrittura è sempre atomica: avviene prima in memoria e subito dopo su disco.

5.4 Gestione attributi

Le operazioni sugli attributi forniscono informazioni di metadati o consentono modifiche limitate:

- `getattr(path)`: restituisce metadati come nome, tipo (file, dir, symlink), timestamp di creazione e modifica.
- `setattr(path, attr, value)`: modifica un attributo specifico, ad esempio il nome.

— L'insieme di queste operazioni rende il DVFS un file system distribuito completo, capace di supportare sia operazioni basilari (creazione e lettura) che funzionalità avanzate (gestione symlink, attributi, apertura/chiusura).

6 Requisiti non funzionali

6.1 Scalabilità

Il modello client-server centralizzato non è intrinsecamente scalabile. Il server rappresenta un collo di bottiglia: all'aumentare del numero di client

connessi cresce il carico di richieste che devono essere gestite da un singolo nodo.

6.2 Disponibilità

Il sistema presenta un **single point of failure**: se il server non è raggiungibile, l'intera rete di client perde accesso al file system. È necessaria la presenza di meccanismi di riavvio rapido o replica futura.

6.3 Prestazioni

- Le operazioni devono avere latenza comparabile ad accessi RMI standard.
- Le scritture sono serializzate: questo garantisce consistenza, ma può ridurre il throughput in scenari con molti client concorrenti.

6.4 Usabilità

Il client fornisce una CLI con comandi noti (mkdir, ls, read, write, edit), garantendo un'interazione familiare per l'utente, simile ad un file system UNIX.

6.5 Sicurezza

- Protezione da path traversal: il server impedisce accessi fuori dalla root montata.
- Gli errori vengono gestiti e propagati come eccezioni RMI.
- Non sono previsti meccanismi di autenticazione o autorizzazione: si assume

un ambiente controllato.

7 Protocolli

La comunicazione tra client e server avviene tramite **Java RMI**. Le invocazioni remote sono trasparenti: il client invoca metodi sull'interfaccia **FileSystemInterface**, che vengono eseguiti dal server sul VFS locale.

7.1 Flusso tipico di un'operazione

1. Il client invia una richiesta remota (es. `write("/foo", data)`).
2. Lo stub RMI inoltra la chiamata a RemoteFileSystem sul server.
3. RemoteFileSystem chiama il metodo corrispondente di FileSystem.
4. FileSystem acquisisce il lock, aggiorna lo stato in memoria e riflette la modifica su disco.
5. Il risultato viene restituito al client.

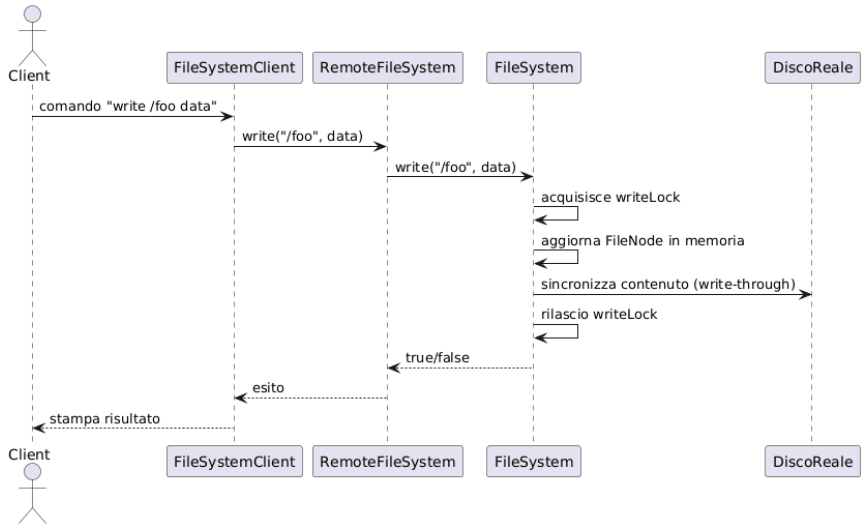

Figure 2: Flusso di una richiesta write

8 Sicurezza ed error handling

- Durante la risoluzione dei path, il server impedisce accessi fuori dalla root montata (protezione da path traversal).
- In caso di errori I/O durante il write-through, l'operazione resta valida in memoria, evitando perdita di dati.
- Gli errori lato server vengono propagati al client come eccezioni RMI.

8.1 Diagramma UML delle classi