Onix

Progetto di Sistemi Distribuiti

Alessandro Marcantoni, Simone Romagnoli

Alma Mater Studiorum - Università di Bologna

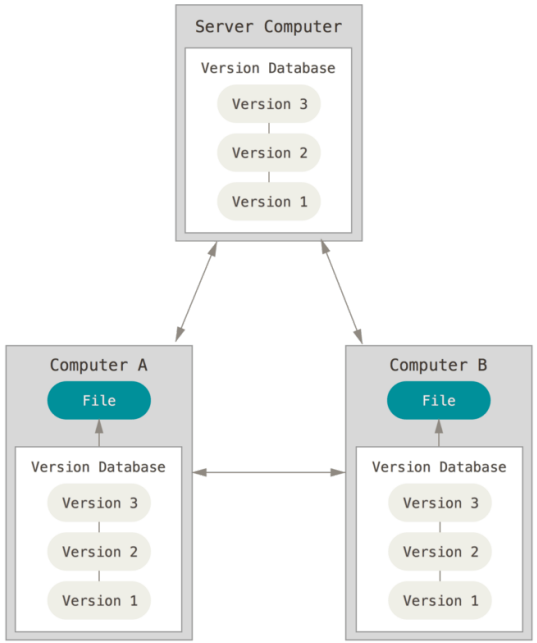
28 Aprile 2022

Introduzione

Il progetto consiste nella realizzazione di un DVCS (*Distributed Version Control System*) fortemente ispirato a Git.

I sistemi di controllo di versione consentono di:

- condividere il codice sorgente con altri sviluppatori;
- tenere traccia delle modifiche apportate a diversi file;
- lavorare parallelamente su medesimi rami di sviluppo o diversi.



A differenza dei *Centralized* VCS, i DVCS sfruttano meccanismi che non rendono indispensabile l'utilizzo di un server centralizzato.

Requisiti

I requisiti funzionali del sistema sono:

1. realizzazione di uno storage che mantenga i dati relativi ad ogni repository;
2. realizzazione di un'interfaccia a riga di comando che permetta di interagire col sistema grazie ad un insieme di comandi;
3. implementazione di un controllo degli accessi che conceda la lettura ai membri e la scrittura agli amministratori di repository.

Tra i requisiti non funzionali si evidenziano:

1. il sistema deve essere più orientato alla consistenza piuttosto che alla disponibilità;
2. il sistema deve essere facilmente estendibile;
3. il sistema deve essere quanto più scalabile.

Scenarios

Un tipico scenario d'utilizzo:

- l'utente `user1` crea un repository per lavorare ad un progetto;
- dopo aver modificato dei file, ne salva le modifiche eseguendo dei *commit*;
- carica le modifiche in remoto con un'operazione di *push*;
- a sua discrezione, concede l'accesso in lettura o in scrittura a un secondo utente `user2`;
- l'utente `user2` può effettuare un'operazione di *clone* per ottenere una copia del progetto;
- a questo punto, entrambi possono lavorare parallelamente sul repository.

Comandi

I comandi accettati dall'interfaccia a riga di comando sono:

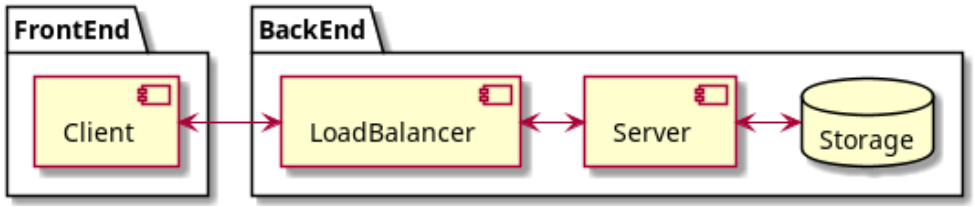
- `onix init <port>`
- `onix clone <user>/<repo-name>/<port>`
- `onix cd <directory>`
- `onix status`
- `onix add <file>`
- `onix commit <message>`
- `onix pull`
- `onix push`
- `onix remote <port>`
- `onix join <user>`
- `onix promote <user>`

Architettura

Viste le specifiche del sistema, l'architettura ideata ha una natura di tipo client-server.

Il lato server è stato modellato sfruttando un approccio a microservizi:

- **Client** - interfaccia a riga di comando;
- **Load Balancer** - punto di ingresso del backend, distribuisce il carico delle richieste;
- **Server** - gestisce le richieste, effettua il controllo degli accessi, consulta lo *storage*;
- **Storage** - mantiene i dati e gli spazi di tuple dei repository.

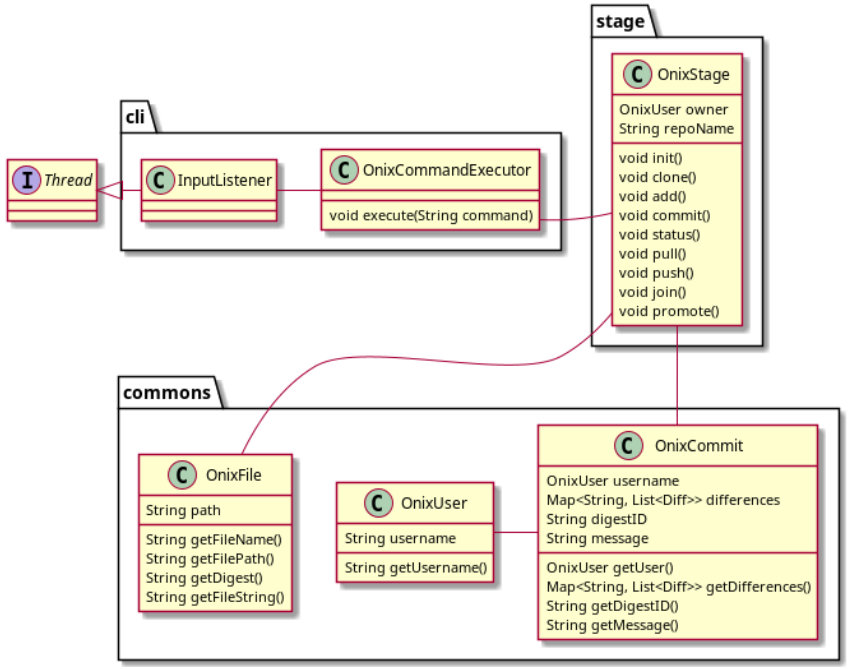


Client

Alla ricezione di un'istruzione, questa viene validata da un `CommandExecutor` e, se valida, eseguita su uno `Stage`.

Le operazioni con il lato server vengono processate tramite il completamento di una Future.

Per garantire compatibilità con molteplici macchine, supporta la gestione dei percorsi relativi dei file piuttosto che quelli assoluti.

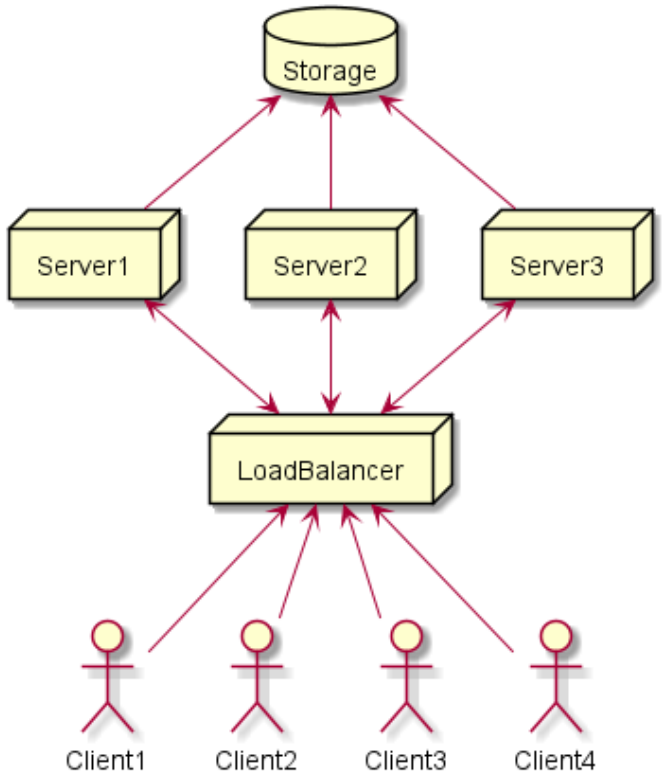


Load Balancer

Ridireziona le richieste dei client verso server che non ne ricevono da più tempo.

Non processa in alcun modo i dati ricevuti, li inoltra solamente.

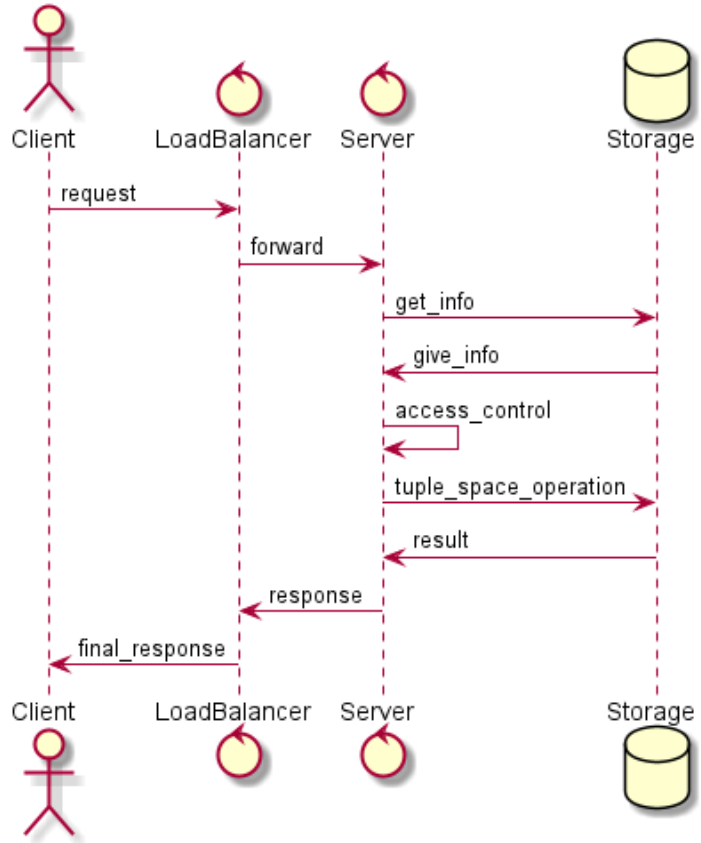
Mantiene salvato l'insieme di server a lui sottoscritti e, per evitare lunghi accodamenti, vi ridireziona le richieste con strategia a round-robin.



Server

Soddisfa le richieste dei client consultando lo *storage*.

Per alleviare lo *storage* da computazioni onerose, i Server eseguono operazioni come il controllo degli accessi e la serializzazione e deserializzazione delle informazioni, delegando ad esso solo le interazioni con gli spazi di tuple.

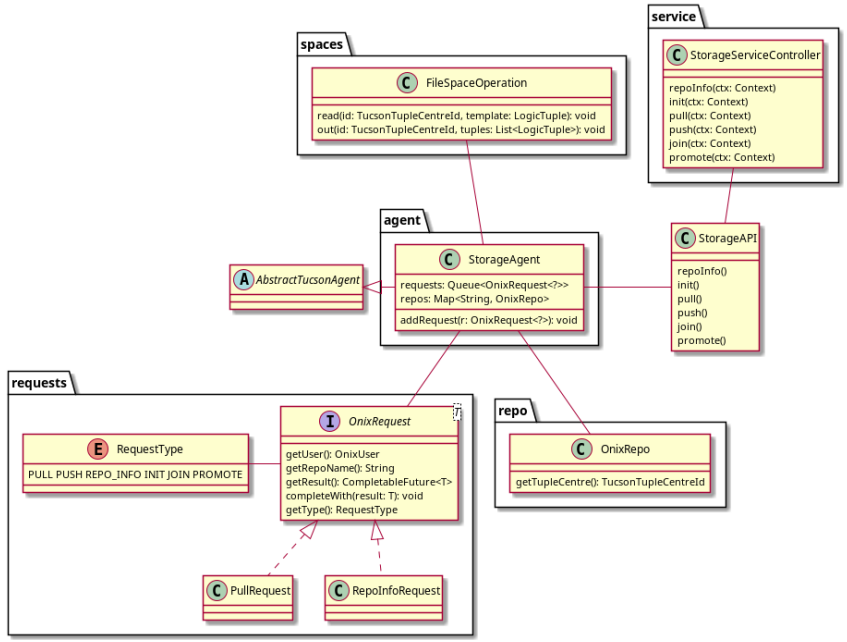


Storage

Mantiene le versioni dei file all'interno di appositi spazi di tuple: uno per ogni repository; immagazzina i dati al proprio interno senza alterarli in alcun modo.

Sfrutta un servizio *TuCSoN* per effettuare operazioni di *read* e *out* sugli spazi di tuple.

Le richieste vengono accodate in una *event queue* e soddisfatte in modo non bloccante da un apposito agente.



Valutazioni

Per valutare la correttezza del software, sono stati sviluppati dei test che verificassero:

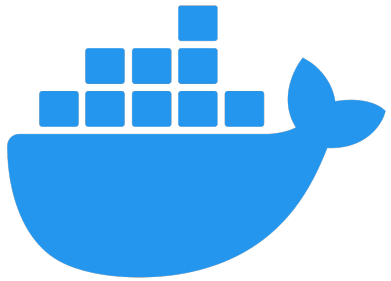
1. **Web Service**;
2. **Marshalling & Unmarshalling**;
3. **Client**.

I test hanno permesso di validare la robustezza del sistema senza effettuarne il deployment e di verificarne il funzionamento anche a fronte di modifiche.

Build Automation & Deployment

Come strumento di *build automation* è stato utilizzato ***Gradle***: infatti, si può far uso del *gradle wrapper* per lanciare velocemente istanze delle varie componenti del sistema.

Inoltre, ogni componente del sistema è stato containerizzato con ***Docker***.



Sviluppi futuri e Conclusioni

I possibili sviluppi futuri del sistema sono:

- inserimento dei meccanismi di *branching* per permettere di lavorare su diversi rami di sviluppo;
- implementazione dell'operazione di *checkout* per tornare a versioni precedenti del repository;
- realizzazione di un servizio web (*GitHub*, *GitLab*, etc) per accedere ai repository con un'interfaccia grafica.

Il sistema soddisfa i requisiti individuati in fase di analisi e il suo sviluppo ha permesso di approfondire le tematiche del corso e di sviluppare delle competenze utili sia in campo accademico che in ambito lavorativo.