Onix - Report Finale

Alessandro Marcantoni
alessandr.marcanton2@studio.unibo.it

Simone Romagnoli
simone.romagnoli10@studio.unibo.it

Marzo 2022

Indice

Introduzione

Il progetto consiste nella realizzazione di un sistema di controllo di versione distribuito (*DVCS*). Si tratta di un tipo di software che permette di tenere traccia delle modifiche e delle versioni apportate a codice sorgente. In questo modo gli sviluppatori possono collaborare individualmente e parallelamente non connessi su un proprio ramo di sviluppo, registrare le proprie modifiche ed in seguito condividerle con gli altri; il tutto senza bisogno di un server centralizzato [9]. Alcuni tra i DVCS più famosi ed utilizzati sono Git [5] e Mercurial [4] che si contrappongono a sistemi centralizzati come SVN [1].

Al giorno d'oggi l'utilizzo di tali sistemi per il controllo di versione del codice è divenuto pervasivo e di fondamentale importanza. Infatti, gli sviluppatori di tutto il mondo manifestano la necessità di collaborare tra loro e pertanto di condividere con altri il proprio lavoro anche a distanza. A tale scopo sono nati i CVCS (Centralized VCS). Questi sistemi sono caratterizzati da un server centrale che contiene tutti i file di un determinato repository dotati di tutte le rispettive versioni. Tutti i collaboratori possono quindi accedere a tali file e rimanere sempre aggiornati riguardo al lavoro svolto dagli altri. Tuttavia, questa soluzione presenta degli svantaggi: ad esempio, se il server centrale non è disponibile nessuno può accedere al contenuto o salvare nuove versioni durante tale lasso di tempo, oppure se i dati del server vanno persi non è possibile in alcun modo recuperare la storia ma solo gli eventuali snapshot dei singoli client. Per questi motivi sono nati i DVCS (Distributed VCS) in cui i singoli client non mantengono solamente uno snapshot del repository ma l'intera storia. Perciò, se un server cade, qualunque dei client può ripristinarlo a partire dalla propria storia.

La nostra versione sarà fortemente ispirata al DVCS Git e ne implementerà le funzioni descritte nella sezione 1. Nello specifico, gli utenti che accedono al client vedranno una linea di comando che offre semplici funzionalità, come creare un nuovo repository o visualizzare quelli a cui si ha accesso. Tramite la riga di comando sarà possibile scrivere istruzioni corrispondenti alle tipiche azioni di un DVCS precedute dalla parola chiave *onix*, omonima del progetto: con queste si potrà apportare modifiche a file, crearne delle versioni, caricarle o scaricarle.

1 Requisiti

1.1 Requisiti Funzionali

Si elencano di seguito i requisiti funzionali del sistema:

1. Realizzazione di un web service che mantenga uno spazio di tuple per ogni repository salvato da un utente.
2. Realizzazione di una interfaccia a linea di comando che permetta di interagire col sistema.
3. L'interfaccia a riga di comando deve permettere di utilizzare i seguenti comandi:
   - `onix init <port>` - inizializza il repository corrente creandone uno remoto sullo storage dispiegato sulla porta specificata;
   - `onix clone <user>/<repo-name>/<port>` - clona il repository specificato;
   - `onix cd <directory>` - permette di spostarsi all'interno della directory specificata;
   - `onix add <file>` - aggiunge il file specificato allo stage;
   - `onix status` - permette di stampare a video la *current working directory* e la lista di file aggiunti allo stage;
   - `onix commit <message>` - salva le modifiche apportate ai file aggiunti allo stage, associandoci il messaggio specificato;
   - `onix pull` - scarica le ultime versioni dei file del repository corrente;
   - `onix push` - manda i commit effettuati al repository remoto aggiornandolo con le corrispondenti modifiche;
   - `onix remote <port>` - crea una copia remota del repository locale sullo storage dispiegato sulla porta specificata;
   - `onix join <user>` - aggiunge l'utente specificato alla lista dei membri del repository;
   - `onix promote <user>` - aggiunge l'utente specificato alla lista degli admin del repository.
4. Realizzazione di un controllo degli accessi che permetta l'accesso in lettura al repository remoto solamente ai suoi membri e in scrittura solamente ai suoi admin.
5. Gestione dei file tramite percorsi relativi piuttosto che assoluti.

1.2 Requisiti Non Funzionali

Tra i requisiti non funzionali si evidenziano:

1. Il sistema finale deve essere più orientato alla consistenza dei dati rispetto alla disponibilità degli stessi: si ritiene infatti che l'integrità delle versioni dei repository sia più importante rispetto all'immediata reperibilità delle stesse.

2. Il sistema deve essere facilmente estendibile: l'integrazione di nuove funzionalità deve risultare poco onerosa.

3. L'architettura del sistema deve favorire la scalabilità dello stesso.

1.3 Requisiti Implementativi e Tecnologici

Si elencano di seguito gli aspetti riguardanti le tecnologie da utilizzare:

1. Il linguaggio di programmazione utilizzato per la realizzazione del sistema è *Java*, alla versione 15.

2. Come strumento di *build automation* viene utilizzato *Gradle*.

3. I servizi web del sistema sono implementati mediante l'utilizzo della libreria *Javalin*.

4. Come supporto per gestire operazioni su spazi di tuple viene utilizzata la libreria *TuCSoN* [6].

5. Per facilitare il confronto tra diverse versioni di file e quindi analizzarne le differenze, si fa utilizzo della libreria `DiffMatchPatch` [2].

6. Per facilitare il deployment e gestire in modo efficiente i servizi all'interno del sistema, questi verranno containerizzati con *Docker* [3].

1.4 Scenarios

Gli utenti possono usufruire del prodotto sviluppato tramite l'apposito applicativo *client*; tramite quest'ultimo, un utente sarà in grado di creare un *repository* remoto e gestirlo a suo piacimento, aggiungendo altri utenti e dando a questi diritti di *admin* per poter avere il permesso in scrittura. Genericamente, un utente si interfaccia al tipo di sistema sviluppato per tener traccia di diverse versioni di un insieme di file; quasi sempre, il contesto è quello di codice di programmazione.

Il tipico scenario d'utilizzo da noi immaginato è il seguente:

- Un utente `user1` crea un repository con l'intento di iniziare un progetto.
- Lo stesso utente inizia a lavorare sul progetto creando e modificando file di testo.
- Per salvare le modifiche apportate, l'utente esegue dei *commit* ed effettua infine un'operazione di *push*, affinché quest'ultimi vengano caricati in remoto.
- Un secondo utente `user2` vuole accedere al repository.

- Il primo utente concede i diritti in lettura o in scrittura (a sua discrezione) al secondo.
- L'utente `user2` può quindi effettuare una operazione di *clone* per ottenere una copia del progetto.
- Infine, gli sarà concesso di effettuare operazioni di *push* o *pull* a seconda dei diritti a lui concessi. Da notare che l'operazione di *pull* sovrascrive le modifiche apportate localmente che verranno perciò perse.

2 Analisi dei Requisiti

In questa sezione, vengono fatte considerazioni in seguito ad una attenta analisi dei requisiti *business* descritti nella sezione 1. Innanzitutto, un *distributed version control system* deve fornire diversi servizi, tra cui:

- Un'interfaccia che permetta all'utente di interagirci.
- Uno storage che mantenga i dati relativi ai repository.
- Un sistema di controllo degli accessi ai repository.
- Salvataggio dell'intera storia del repository in locale.

A partire da questi, si individuano le varie componenti del sistema sulla base delle loro funzionalità; nello specifico, si identificano un client per l'interazione con gli utenti, uno storage remoto che mantenga i repository e dei servizi intermedi che implementino il controllo degli accessi. La presenza dei servizi intermedi è utile anche per evitare eventuali *bottleneck*: evitando l'interazione diretta tra client e storage, si riescono a dividere le computazioni alleggerendo il carico su un unico componente.

Oltre alla definizione di un design soddisfacente per il tipo di sistema, un altro fattore importante è costituito dagli elementi statici del dominio: è necessario individuare un metodo di rappresentazione degli elementi logici all'interno del sistema; inoltre, tale modello deve essere interpretato da tutti i componenti del sistema. Le versioni di un file devono essere distinguibili sulla base del loro nome e del loro contenuto: quindi una versione viene identificata dall'hash digest della stringa composta dal nome del file concatenato al suo contenuto; in questo modo, se un file non è stato modificato, la sua versione rimane invariata. Le versioni di un file vengono create durante l'operazione di *commit*: quest'ultimo viene identificato dall'utente che ha eseguito il *commit* e dal *timestamp* del momento in cui si effettua l'operazione. I repository sono identificati dall'utente creatore e da un nome univoco solamente tra quelli appartenenti a lui; di conseguenza, possono esistere più repository con lo stesso nome, ma con proprietari diversi. Inoltre, un repository può esistere in diversi remoti in modo da poter gestire più copie dello stesso progetto. Infine, in ogni repository sono memorizzati tutti i membri con relativi privilegi di accesso e la lista dei commit che ne rappresenta la storia. Tale modello, permetterà la definizione di **rotte** in ogni componente che saranno vantaggiose per definire delle *API REST* utili per le comunicazioni interne al dominio.

3 Design

In questa sezione, viene descritto il design del sistema a tre livelli di astrazione: architettura, comportamento e interazione.

3.1 Architettura

Viste le specifiche del sistema, l'architettura ideata ha una natura di tipo client-server: gli utenti si interfacciano ad un client che comunica con un server (costituito da diversi componenti) che mantiene le versioni dei file dei repository. È stato deciso di modellare il lato server sfruttando un approccio a microservizi: questo garantisce scalabilità, flessibilità, facilità di distribuzione e una migliore *separation of concerns*. Inoltre, i microservizi che compongono il lato server sono organizzati con una struttura tale da evitare eventuali *bottleneck*:

- **Load Balancer** - costituisce il punto di ingresso del backend del sistema. Si occupa di dividere equamente il carico delle richieste ricevute dai diversi client; quando giunge una richiesta, quest'ultimo la ridireziona verso uno dei server disponibili.
- **Server** - gestisce le richieste dei client, effettua il controllo degli accessi e soddisfa tali richieste consultando e opportunamente manipolando i dati salvati su un apposito storage.
- **Storage** - mantiene i dati relativi ai repository creati dagli utenti; si occupa di gestire le richieste provenienti dai server che richiedono o inviano informazioni.

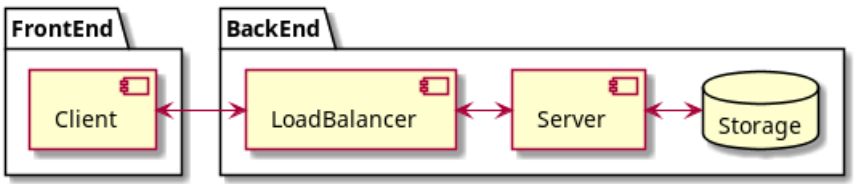

Figura 1: Diagrama UML delle componenti dell'architettura descritta.

Ogni componente descritto può essere contattato sfruttando l'apposita *API REST*. La potenzialità di individuare con precisione le funzionalità di ciascun componente è scaturita a partire dalla suddivisione delle diverse operazioni su più nodi: è stato quindi naturale esporre le API di ciascuno in modo da evidenziarne il comportamento. A riguardo, si riporta in figura 2 la specifica *Swagger* del sistema appositamente creata.

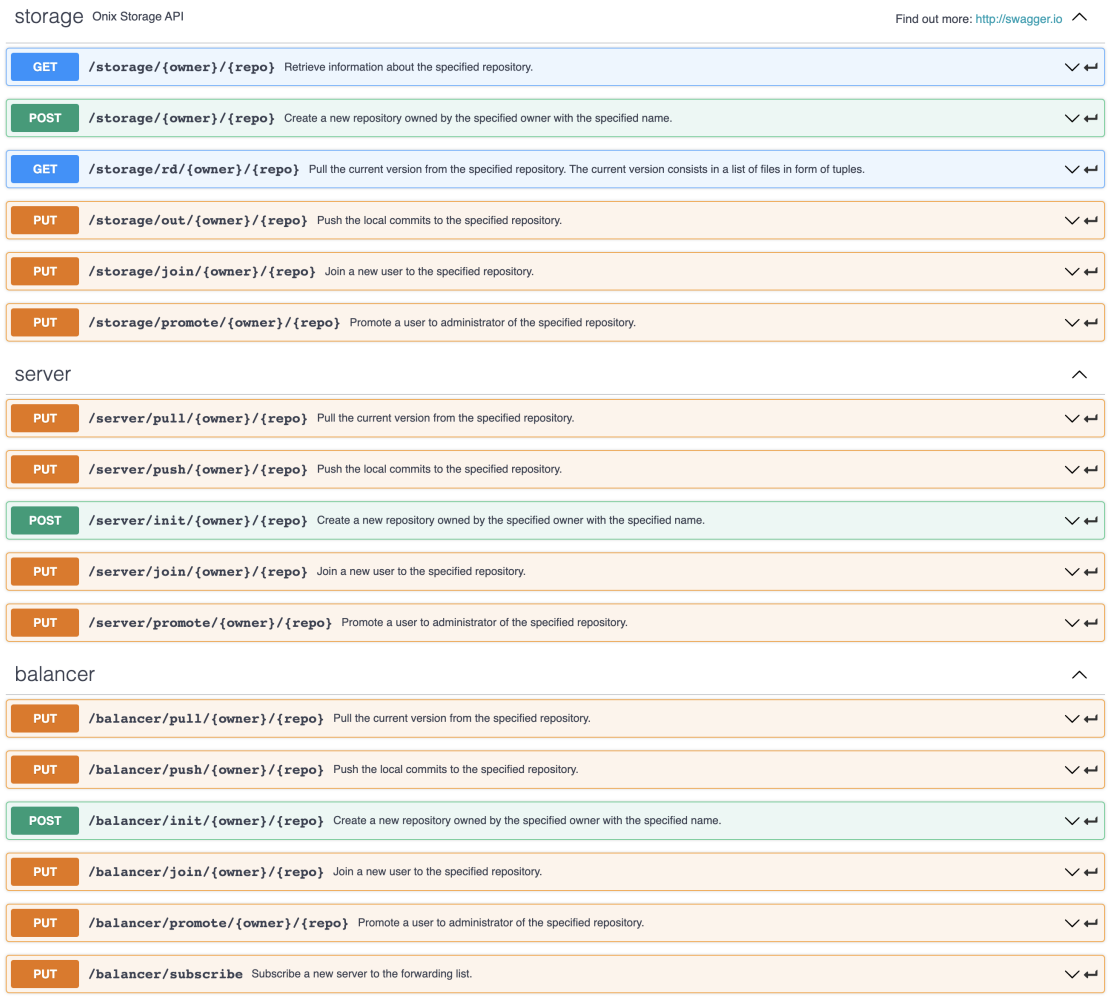

Figura 2: Specifica Swagger delle API del sistema.

Quindi, mentre il client opera su un proprio `Stage` dotato di un `Client HTTP`, tutte le componenti del backend sono dotate di un *service* costituito da un *server Javalin*; inoltre, ogni entità descritta incorpora le proprie API interne che permettono di far riferimento alle corrispettive funzionalità all'interno del rispettivo package (come mostrato in figura 3).

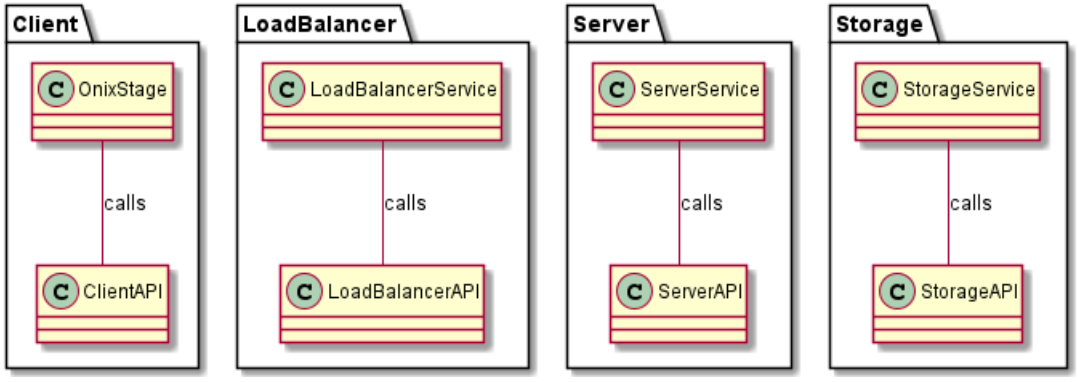

Figura 3: Diagrama UML dei package che compongono l'architettura descritta.

L'architettura appena descritta è ottimizzata per evitare *bottleneck* a partire dalle richieste degli utenti; tuttavia, si precisa che su ampia scala andrebbe rimodellata considerando una serie di altri componenti basati anche sulla dislocazione fisica di client e server. Inoltre, si vuole evidenziare che l'immediata evoluzione del sistema sviluppato include algoritmi di consenso (vedi Ricart Agrawala [8]) e di snapshot (vedi Chandy Lamport [7]) per mantenere un insieme potenzialmente infinito di dati consistenti.

3.2 Comportamento

In questa sezione si descrive il comportamento di ogni componente precedentemente descritto, enfatizzando il funzionamento interno e la sequenza di azioni che implementa, piuttosto che sulle interazioni tra componenti. Ogni entità del sistema implementa efficientemente un'attività di logging per individuare facilmente eventuali errori di rete o di comunicazione, ma anche per mostrare il flusso di informazioni agli amministratori del sistema.

3.2.1 Client

A livello comportamentale, il Client rappresenta la componente più ricca del sistema. Esso si compone di un'interfaccia a riga di comando che accetta un sottoinsieme di istruzioni utili ad interagire con il sistema. Alla ricezione di un'istruzione, questa viene validata da un `CommandExecutor` e, se valida, eseguita su uno `Stage`. Come mostra la figura 4, lo `Stage` permette di eseguire le operazioni descritte nella sezione 1 e sfrutta delle classi come `OnixCommit` e `OnixFile` appositamente sviluppate per gestire al meglio i dati relativi ai file locali e le corrispondenti conversioni in commit.

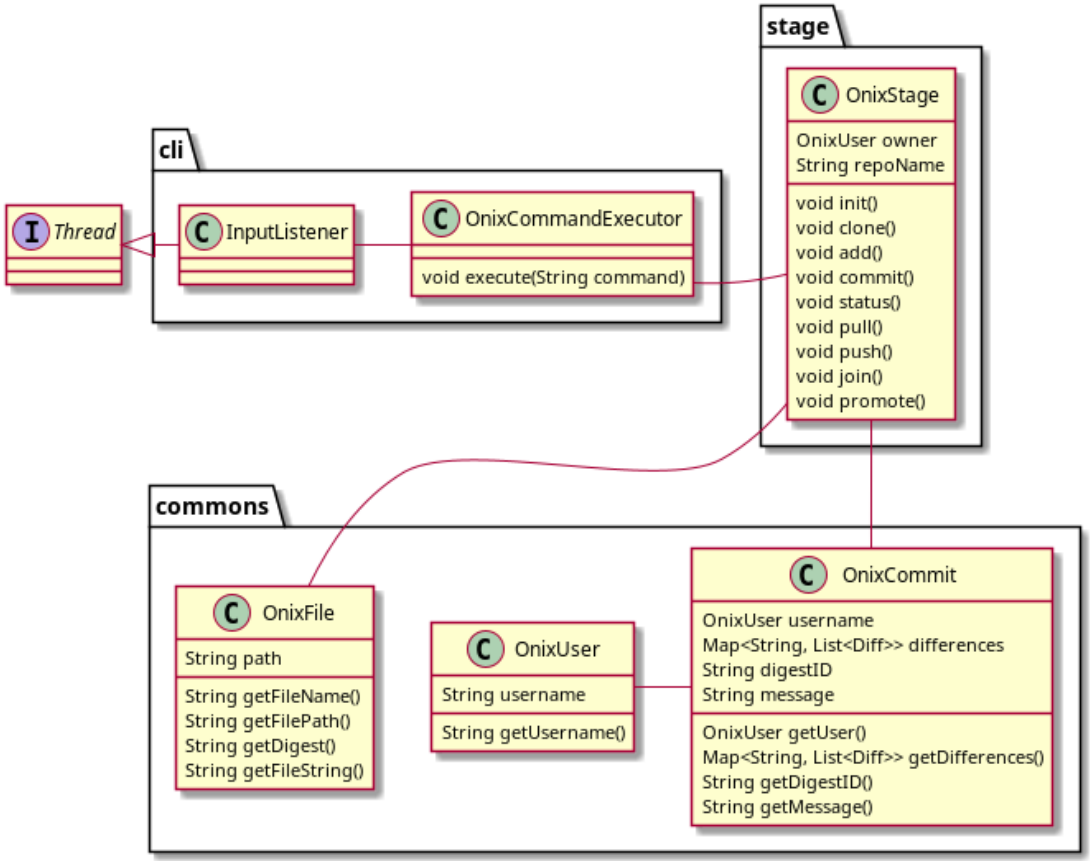

Figura 4: Diagrama UML delle classi che compongono il Client.

Le operazioni che il client deve gestire possono anche prevedere comunicazioni con il lato server del sistema: in tal caso, le risposte vengono processate tramite il completamento di una `Future`. Quando non sono richieste comunicazioni remote, il Client soddisfa spesso operazioni di gestione dei file, come l'aggiunta di questi allo `Stage` o la creazione di commit. Molte azioni dello `Stage`, infatti, prevedono una forte attività di input/output per la gestione del repository locale: sono stati quindi implementati degli algoritmi che permettono facilmente di creare, leggere, modificare e cancellare file localmente. Inoltre, per garantire compatibilità con molteplici macchine, esso supporta anche la gestione dei percorsi relativi dei file piuttosto che quelli assoluti: in questo modo, è anche possibile salvare le versioni dei file su spazi di tuple a prescindere dalle dipendenze locali dei sistemi operativi.

3.2.2 Load Balancer

Lo scopo del Load Balancer è quello di ridirezionare le richieste dei client verso server che non ricevono richieste da più tempo. Vista la sua natura, non processa in alcun modo i

dati ricevuti, tuttavia mantiene salvato l'insieme di server a lui sottoscritti e, per evitare lunghi accodamenti su questi, vi ridireziona le richieste con strategia a *round-robin*.

3.2.3 Server

Il Server si occupa di soddisfare le richieste dei client consultando lo Storage. Quando riceve una richiesta esegue il corrispettivo handler per richiamare la corretta procedura di lettura o scrittura sullo spazio di tuple dello 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. Per poter eseguire tali operazioni, il Server necessita di una prima interazione con lo Storage per ricevere informazioni generali sul repository, come i diritti di accesso degli utenti. A tale scopo, gli handler delle richieste sono composti da una prima richiesta in `GET`, seguita dal controllo degli accessi e l'eventuale esecuzione della corrispondente operazione sullo spazio di tuple (con una richiesta in `POST` per l'inizializzazione di un repository e una in `PUT` per le altre procedure).

3.2.4 Storage

Lo Storage è un componente fondamentale all'interno del sistema in quanto mantiene le versioni dei file all'interno di appositi spazi di tuple: uno per ogni repository. Oltre ad operare sugli spazi di tuple sfruttando le primitive di *read* e *out* supportate da *TuCSoN*, si occupa di mantenere le informazioni degli utenti, come l'insieme dei membri di un repository (che hanno solo accesso in lettura) e gli admin (che hanno anche accesso in scrittura).

Come già precedentemente descritto, lo *Storage* espone una API accessibile mediante HTTP. Alla ricezione di una richiesta, un apposito handler inserisce quest'ultima all'interno di una *Event Queue* gestita da un agente. Tale entità, considerata la sua natura, si occupa di soddisfare tali richieste in modo non bloccante così da essere sempre reattiva anche a fronte di un carico elevato. Per svolgere il suo compito, raccoglie informazioni dai repository immagazzinati al suo interno e si serve di operazioni su spazi di tuple per leggere e/o scrivere versioni di file.

L'architettura appena descritta è rappresentata in figura 5.

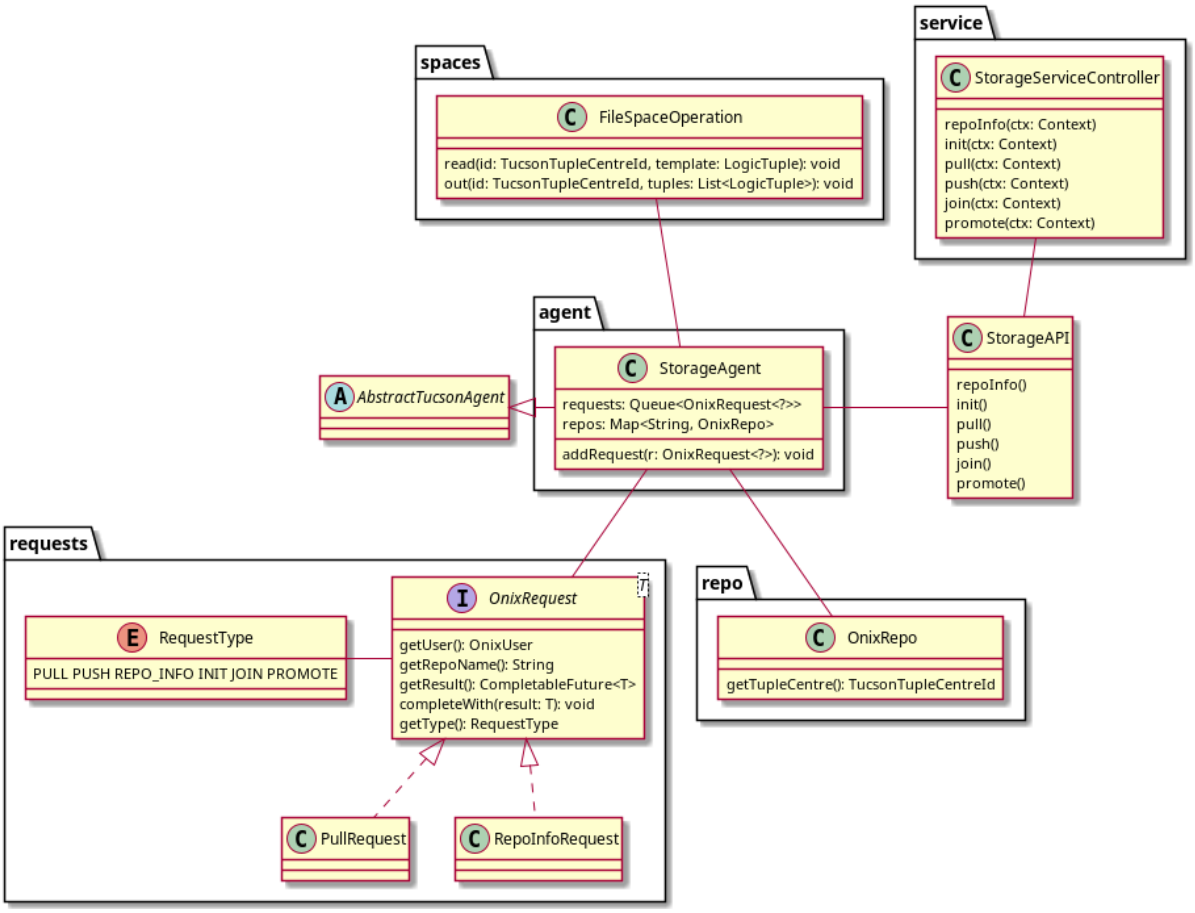

Figura 5: Diagramma UML delle classi che compongono lo storage.

Si vuole inoltre evidenziare come lo storage abbia il solo compito di immagazzinare al proprio interno i dati relativi ai repository e agli utenti: pertanto, non effettua alcun tipo di operazione sui dati stessi, bensì si limita ad inoltrarli in caso di richieste.

3.3 Interazione

L'interazione tra le componenti descritte è fondamentale anche a livello strutturale; infatti, è possibile che il flusso comunicativo possa essere già stato esplicitato a livello architetturale.

Il diagramma delle componenti in figura 6 mostra come avviene l'interazione nel sistema dispiegato con molteplici client e server: in particolare, si evidenzia come la funzione del LoadBalancer sia fondamentale per garantire scalabilità al sistema con l'aumentare del numero dei client e, di conseguenza, delle loro richieste.

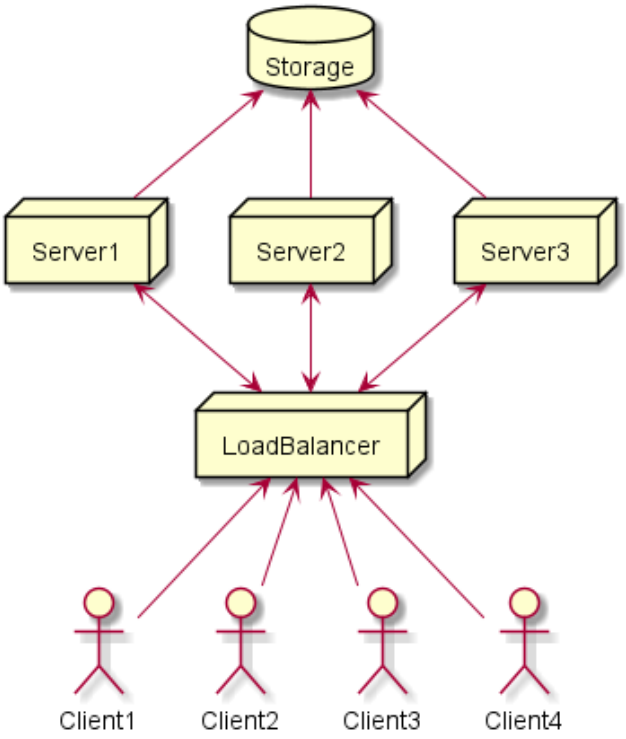

Figura 6: Diagrama UML delle componenti del sistema: il caso di deployment con 4 client e 3 server.

Il diagramma di sequenza un figura 7 mostra come avvengono le comunicazioni, concentrandosi sull'entità che le inizia, e come si trasforma l'informazione di una richiesta partendo dal client fino alla ricezione della risposta finale. Nello specifico, si vuole sottolineare il fatto che lo Storage sia stato liberato da operazioni onerose come il controllo degli accessi o la conversione delle tuple in modo da aumentarne la disponibilità con l'accrescere delle richieste.

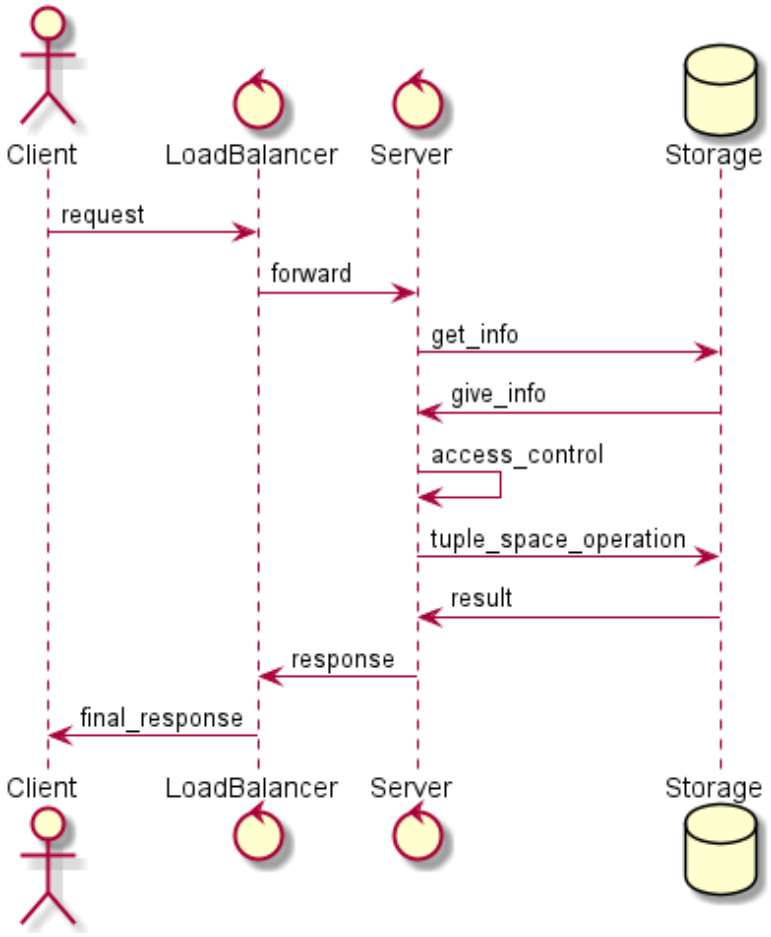

Figura 7: Diagrama UML di sequenza del flusso di informazioni a partire da una richiesta del client.

4 Dettagli Implementativi

In questa sezione vengono descritti i dettagli implementativi degni di nota facendo eventualmente riferimento a porzioni di codice. Per ulteriori dettagli, consultare l'apposita documentazione.

4.1 Individuazione delle differenze tra le versioni di file

Per quanto riguarda l'individuazione di differenze tra file, che costituisce un fattore fondamentale dei *VCS*, è stata sfruttata la libreria `DiffMatchPatch` [2]. Vista la complessità algoritmica dei sistemi come Git, sarebbe stato facile perdersi nei tentativi di implementazione di meccanismi di analisi del testo: si è quindi scelto di adoperare una libreria che li implementasse già. In particolare, la libreria permette di trovare una `List<Diff>` a partire da due file di testo: ogni `Diff` è costituito da una stringa e dal tipo di operazione che può essere di aggiunta o di rimozione. Di conseguenza, per aggiornare un file è necessario applicare la `List<Diff>` tra i due file, aggiornando il contenuto del più datato a quello del più recente. Il listato 1 mostra come viene fatto l'aggiornamento di un `OnixFile` a partire dal suo contenuto e dalla `List<Diff>`.

```
1  private static void applyDiffs(List<Diff> diffs, OnixFile of) {
2      StringBuilder newContent = new StringBuilder();
3      diffs.stream()
4          .filter(ServerAPI::isInsertOperation)
5          .forEach(diff -> newContent.append(diff.text));
6      of.setFileString(newContent.toString());
7      of.updateDigest();
8  }
9
10 private static boolean isInsertOperation(Diff diff) {
11     return diff.operation == DiffMatchPatch.Operation.EQUAL
12     || diff.operation == DiffMatchPatch.Operation.INSERT;
13 }
```

Listing 1: Applicazione delle modifiche ad un file.

Tale procedimento avviene durante l'operazione di *push* per creare la nuova versione a partire dall'ultima registrata nello spazio di tuple. Sarebbe opportuno utilizzare tale meccanismo anche per l'operazione di *pull*, tuttavia, per questioni di complessità, si è ritenuto sufficiente sovrascrivere tutte le versioni locali. Nonostante ciò, il mantenimento delle differenze tra le versioni nello storage favorisce l'estendibilità del sistema: ad esempio, non sarebbe difficile in futuro implementare la funzionalità di *checkout*.

4.2 Utilizzo di interfacce funzionali

Si evidenzia l'utilizzo delle interfacce funzionali in diverse aree del sistema: vista l'implementazione di diversi *controller* per i servizi sviluppati, alcune porzioni di codice risulterebbero ripetitive e particolarmente lunghe. Quindi è stato scelto di semplificare l'implementazione dei servizi laddove l'utilizzo del paradigma funzionale avrebbe conferito non solo eleganza al codice, ma anche una maggiore manutenibilità ed estendibilità.

Un esempio è l'implementazione del `ServerServiceController`, all'interno del quale si gestiscono gli *handler* delle chiamate del *service*.

```
/** Allows to call a specific api's functionality upon every request. */
@FunctionalInterface
private interface ServiceApiCaller<T> {
    CompletableFuture<T> call(String owner,
        String repoName, String body);
}

/** Call the specified action on every request. */
private static void callApi(final Context ctx, final ServiceApiCaller<
    String> caller) {
    var repoName = ctx.pathParam(REPO_NAME);
    var owner = ctx.pathParam(OWNER);
    var body = ctx.body();
    if (!body.isEmpty() && !body.isBlank()) {
        ctx.contentType(JSON).result(caller.call(owner, repoName, body));
    }
}

/** Call the api's pull handler. */
public static void pull(final Context ctx) {
    callApi(ctx, (owner, repoName, body) ->
        ServerAPI.pull(owner, repoName,
            Converter.deserialize().currentUser(body)));
}
```

Listing 2: Interfaccia funzionale utilizzata per gestire diversi handler in base alla chiamata effettuata.

Il listato 2 mostra come sia stato implementato un metodo `callApi` poi utilizzato in ogni metodo esposto alle rotte del *service* (ad esempio nel metodo *pull*) chiamando l'apposita funzionalità delle API interne.

4.3 Gestione operazioni con agenti

Data la varietà di possibili operazioni eseguibili dallo storage, si è ritenuto opportuno fornire un'interfaccia comune ad ogni richiesta in modo che sia possibile processarle mediante un unico metodo `handleRequest(OnixRequest<?> r)`. Ogni operazione è quindi rappresentata da un'apposita implementazione di tale interfaccia: ad esempio, la richiesta di *pull* è gestita tramite la creazione di un'istanza di `PullRequest`. A questo punto, il corretto handler per ogni richiesta viene recuperato sfruttando una mappa statica a partire dal tipo dell'operazione (come mostrato nel listato 3).

```
private static final ImmutableMap<RequestType, RequestHandler> HANDLERS =
    ImmutableMap.<RequestType, RequestHandler>builder()
                .put(RequestType.PULL, StorageAgent::handlePull)
                .put(RequestType.PUSH, StorageAgent::handlePush)
                .put(RequestType.REPO_INFO, StorageAgent::handleRepoInfo)
                .put(RequestType.INIT, StorageAgent::handleInit)
                .put(RequestType.JOIN, StorageAgent::handleJoin)
```

```
                .put(RequestType.PROMOTE, StorageAgent::handlePromote)
                .build();

@FunctionalInterface
private interface RequestHandler {
    void handleRequest(
        final OnixRequest<?> request,
        final Map<String, OnixTupleCentre> repos
    );
}

@Override
protected void main() {
    while (this.running) {
        if (!requests.isEmpty()) {
            final var request = requests.poll();
            HANDLERS.get(request.getType()).handleRequest(request);
        }
        waiting();
    }
}
```

Listing 3: Gestione degli handler delle richieste dello storage.

Ogni `OnixRequest` presenta al suo interno una `CompletableFuture` che sarà completata con il risultato dell'operazione stessa. Ciò permette ai metodi che le utilizzano di ritornare immediatamente la *future* senza bisogno di attendere la fine dell'operazione. È possibile quindi affidare a degli `Executor` i task da svolgere che richiedono tempo non trascurabile per essere eseguiti. Tali thread secondari, una volta pronto il risultato della computazione, completeranno appunto la *future* e verrà solo a quel punto eseguita la callback ad essa associata.

5 Valutazioni

Per valutare l'effettivo riscontro del comportamento descritto nell'analisi dei requisiti all'interno del software prodotto, sono stati sviluppati degli Unit Test. In particolare, sono stati testati alcuni aspetti principali:

- **Service** - per ogni web service sviluppato, sono state testate le API effettuando delle richieste alle rotte specificate dal corrispondente *service* e verificando il risultato atteso. Ciò ha permesso non solo di verificare la corretta implementazione del *service* in sé, ma anche di controllare l'effettivo funzionamento delle API interne di ogni componente: ad esempio, alla chiamata di una `init` è stato possibile controllare che le componenti ricevessero la richiesta alla rotta indicata e ritornassero il risultato atteso (in questo caso con il corretto *status code* a seconda del repository).

- **Marshalling & Unmarshalling** - per evitare errori di conversione indesiderati, sono state testate serializzazione e deserializzazione dei dati che vengono inviati all'interno dei messaggi http. Vista la complessità delle strutture dati sviluppate, è stato fondamentale verificare che non venissero persi dati nella serializzazione e successiva deserializzazione, ma soprattutto che i dati ricevuti corrispondessero a quelli inviati, senza subire alterazioni.

- **Client** - per verificare la corretta ricezione degli input da riga di comando, ma anche per testare l'effettivo funzionamento della logica *push-pull* senza dover dispiegare manualmente il sistema, sono stati eseguiti dei test che simulassero i principali casi d'uso del client del sistema. Questi test, oltre ad accertare il corretto funzionamento dello *stage*, hanno permesso di convalidare l'intero sistema di tracciamento sviluppato, verificando anche che i file degli utenti venissero correttamente condivisi assieme ai relativi contenuti.

Sviluppare tali valutazioni ha permesso di verificare il corretto funzionamento del sistema senza doverne effettuare il deployment: ciò ha consentito di lavorare in maniera efficiente, senza perdere tempo effettivo nel testing, e di validare la robustezza del sistema realizzato. Inoltre, avendo sviluppato delle *suite* di test, è stato possibile verificare il funzionamento del sistema a fronte di modifiche, sia a livello implementativo, sia a livello di design.

Si fa notare che, per eseguire i test relativi allo storage, **non** è necessario lanciare alcun servizio di support *TuCSoN* in quanto viene eseguito autonomamente in *background* dallo storage. Per il corretto dispiegamento delle componenti del sistema, si fa rifermento alla sezione 6.

6 Deployment

In questa sezione vengono descritte le modalità con cui può essere effettuato il deployment del sistema.

6.1 Docker

Ogni componente del sistema è stato containerizzato con *Docker*: di conseguenza, è sufficiente aver installato tale software sulla propria macchina per riuscire a lanciare il programma. È possibile generare l'immagine *Docker* del sistema sfruttando il `Dockerfile` appositamente creato: esso sfrutta anche l'eseguibile `run` a cui vengono passati gli argomenti relativi alle varie componenti del sistema. Il comando per creare l'immagine è `docker build -t onix-image .` da lanciare all'interno della root directory del progetto. Una volta ottenuta l'immagine, è possibile lanciare un container che esegue un componente del programma con un comando specifico.
Le opzioni del comando variano a seconda di quale componente si vuole eseguire. Di seguito, si descrivono i comandi con le relative opzioni per ogni componente e cosa avviene dentro il container:

- Client - `docker run onix-image -it client user1 <local-ip-addr>`. Con questo comando viene lanciata con *Gradle* un'istanza del *client* con nome utente `user1`. Sarà successivamente possibile interagire con la linea di comando lanciata utilizzando i comandi specificati nella sezione 1.

- LoadBalancer - `docker run -p 3000:3000 onix-image loadBalancer`. Con questo comando viene lanciata con *Gradle* un'istanza del *load balancer*; il service *Javalin* viene istanziato in *localhost* alla porta 3000. Il programma lanciato non prevede interazioni con l'utente, tuttavia viene mostrata l'attività di logging che stampa le ricezioni di richieste da parte dei *client* e le connessioni coi server sottoscritti.

- Server - `docker run -p 5000:5000 onix-image server 5000 <local-ip-addr>`. Con questo comando viene lanciata con *Gradle* un'istanza del *server* del sistema; il service *Javalin* viene stanziato in *localhost* alla porta specificata, in questo caso la numero 5000. Il programma lanciato non prevede interazioni con l'utente, tuttavia viene mostrata l'attività di logging che stampa le ricezioni di richieste da parte del *load balancer*. Tale programma va eseguito dopo aver dispiegato un'istanza del *load balancer* in quanto è necessario per sottoscriversi ad esso e notificarlo della sua presenza: in questo modo il *load balancer* riesce a inoltrare ai *server* le richieste ricevute dai *client*.

- Storage - `docker run -p 8000:8000 onix-image storage 8000`. Con questo comando viene lanciata con *Gradle* un'istanza dello *storage*; tale comando lancia anche in *background* il servizio *TuCSoN* (alle porte 20504 e seguenti), necessario per il mantenimento dello spazio di tuple; il service *Javalin* viene stanziato in *localhost* alla porta specificata, in questo caso la numero 8000. Tale programma costituisce

quindi la base della persistenza del sistema totale: infatti, senza tale componente dispiegato il sistema non funzionerebbe.

6.2 Gradle

Nel caso non si disponesse del software *Docker*, o semplicemente non si ritenesse necessario il suo utilizzo, è comunque possibile dispiegare le componenti del sistema sfruttando *Gradle* nel seguente modo:

- Client - `./gradlew runClient --args="user1" --console=plain`
- LoadBalancer - `./gradlew runLoadBalancer --console=plain`
- Server - `./gradlew runServer --args="5000" --console=plain`
- Storage - `./gradlew runStorage --args="8000" --console=plain`

Si ricorda nuovamente che **non** è necessaria l'esecuzione in *background* del servizio *TuCSoN* per avviare correttamente lo *storage*.

7 Esempi d'uso

Per poter interagire con il *client* ed ottenere risultati concreti, è necessario eseguire un'istanza dell'intero backend: in particolare, bisogna innanzitutto lanciare lo *storage*, seguito dal load balancer e dal numero di server desiderato (ognuno eseguito con una porta diversa). Sarà successivamente possibile eseguire un'istanza del client con un username a scelta. Una volta eseguito il client, questo lancerà una riga di comando che accetta i comandi descritti nella sezione 1. Supponendo di aver lanciato il sistema con nome utente `user1` e trovandosi in una cartella `project42` vuota, con il comando `onix init 8000` verrà inizializzato il repository remoto vuoto `user1/project42` sullo Storage registrato alla porta 8000.
Successivamente, supponendo di aver creato un file `example.txt` all'interno della cartella, con il comando `onix add example.txt` si aggiunge tale file allo *stage*, così come le sue eventuali modifiche; con il comando `onix add .` si aggiungono allo *stage* tutti i file contenuti all'interno della cartella.
Una volta che i file desiderati sono stati aggiunti allo *stage*, si può verificare lo stato di quest'ultimo con `onix status` oppure è possibile eseguire un *commit*, con relativo messaggio, con il comando `onix commit "this is a message"`. Il *commit* comprende tutti i file aggiunti allo *stage*: in questo caso, riguarda solo il file `example.txt`.
Le precedenti operazioni possono ripetersi diverse volte, il sistema salva i *commit* per disfarsene quando vengono caricati in remoto; per effettuare questa operazione è necessario digitare il comando `onix push`.
Affinché un secondo utente `user2` possa scaricare il contenuto del repository, il primo deve necessariamente aggiungerlo tra i membri con il comando `onix join user2`. Come indicato nella sezione 1, in questo modo si concedono i diritti in lettura, mentre, per ottenere anche quelli in scrittura (ovvero la possibilità di effettuare una *push*), è necessario eseguire il comando `onix promote user2`. Una volta ottenuti i permessi, il secondo utente può clonare il repository con `onix clone user1/project42` e, in futuro, aggiornare il repository locale con `onix pull`.
È possibile infine salvare una copia del repository su un altro remoto tramite il comando `onix remote 8001`, dove 8001 è la porta a cui è registrato lo Storage remoto su cui si salva la copia.

8 Conclusioni

L'idea del progetto è partita da un interesse personale di entrambi i membri del gruppo; portando avanti il lavoro, oltre ad essersi consolidata la cooperazione dei due membri del team, sono state rafforzate le conoscenze di ciascuno riguardanti l'ambito del progetto. Inoltre, sono state consolidate le conoscenze riguardanti gli argomenti del corso in questione e le nuove tecnologie che questo richiedeva di saper utilizzare. Infine, ci si ritiene soddisfatti del risultato ottenuto, consapevoli di poter riutilizzare quanto imparato in futuro, sia in ambito accademico, sia in ambito lavorativo.

8.1 Lavori Futuri

Il sistema progettato prende ispirazione dalle caratteristiche di uno dei più famosi e utilizzati *DVCS*. L'evoluzione immediata del sistema sviluppato consisterebbe nell'arricchimento delle sue funzionalità, sempre ispirandosi a *Git*: una feature che non può mancare, ad esempio, sarebbero i *branch* sia locali che remoti. Inoltre, è risaputo che l'algoritmo di gestione della storia di *Git* è molto complesso e permette svariate operazioni, grazie anche al mantenimento di strutture dati chiamate *tree-ish*. Una forte limitazione del sistema sviluppato si nota nell'operazione di *pull*: in presenza di modifiche locali queste vengono completamente sovrascritte. Un'altra possibile espansione potrebbe essere la realizzazione di un servizio web ispirato a *GitHub*, *GitLab*, etc. grazie al quale sia possibile accedere ai repository con un'interfaccia grafica e gestirne proprietà come il controllo degli accessi.

Riferimenti bibliografici

[1] *Apache Subversion*. URL: https://subversion.apache.org/.

[2] cowwoc. *Diff Match Patch*. URL: https://mvnrepository.com/artifact/org.bitbucket.cowwoc.diff-match-patch.

[3] *docker*. URL: https://www.docker.com/.

[4] *Mercurial*. URL: https://www.mercurial-scm.org/.

[5] Linus Torvalds. *Git*. URL: https://git-scm.com/.

[6] unibo. *TuCSoN*. URL: https://apice.unibo.it/xwiki/bin/view/TuCSoN/.

[7] Wikipedia. *Chandy Lamport Algorithm*. URL: https://en.wikipedia.org/wiki/Chandy-Lamport_algorithm.

[8] Wikipedia. *Ricart Agrawala Algorithm*. URL: https://en.wikipedia.org/wiki/Ricart-Agrawala_algorithm.

[9] Wikipedia. *Sistema di Controllo Versione Distribuito*. URL: https://it.wikipedia.org/wiki/Controllo_versione_distribuito.