Paolo Speziali

Condividere informazioni in modo sicuro combinando Git e Blockchain

Relatore:
Prof. Luca Grilli

Tesi di laurea in Ingegneria Informatica

Perugia, Anno Accademico 2020/2021

Università degli Studi di Perugia
Corso di laurea triennale in Ingegneria Informatica ed Elettronica
Dipartimento d'Ingegneria

0. Indice

1. Introduzione

2. Concetti Preliminari

Di seguito si introducono alcuni concetti per permettere al lettore di acquisire le nozioni necessarie alla corretta fruizione del materiale successivo.

2.1 VCS e Git

Un **Version Control System** (o anche VCS) è una tipologia di software per la condivisione, il controllo e la tracciabilità dei cambiamenti riguardanti determinati file e directory lungo un lasso di tempo e che permette agli utenti di recuperare rapidamente specifiche versioni dei loro documenti. Gli insiemi di file e cartelle gestite da questi sistemi sono suddivisi in **repository**, esse vengono trattate l'una isolata dalle altre. Spesso si considera una intera directory di lavoro, con il suo contenuto, un'unica repository, potendo però scegliere di escludere alcune risorse. Un VCS può essere centralizzato o distribuito. Nel primo caso è il server centrale che tiene traccia dei cambiamenti e che mantiene e distribuisce la versione più recente delle risorse richieste, gli utenti possono gestire le loro repository solo attraverso client lightweight che interagiscono con il server per riuscire a compiere una qualsiasi operazione. Nel secondo caso ogni client ha una copia precisa della repository e del suo storico salvata localmente, i server sono coinvolti solo per effettuare sincronizzazioni di repository tra i vari client.

Git [1] is the most widespread distributed version control system. It models every repository as a *sequence of snapshots* (or *stream of snapshots*) of a miniature file system. Whenever a user saves the state of their project (through a commit operation), Git takes a snapshot of all files and folders under version control at that moment, and stores the snapshot in its local database; files that have been modified (since the last commit) are entirely included in the latest snapshot, while every unchanged file is not duplicated, and just a link to the previous identical copy is inserted in the snapshot. Every resource in a repository is internally identified by its checksum (SHA-1), and not by its name, which allows Git to efficiently detect changes in files, and makes any undetectable change very unlikely. Also, nearly all Git operations only add data to its database, including the file

removal operation (rm command). This ensures that any change is reversible, with basically no risk of permanently losing data. Every file in a working directory can be in one of two main states: *untracked* or *tracked*. An *untracked* file is a file that was never added to the repository or that has been removed from the set of tracked files (rm command). A newly created file is therefore in the untracked state. All the other files are *tracked* and can in turn be in one of three states: *unmodified* (or *committed*), *modified* and *staged*. A tracked file is *unmodified* (or *committed*) when it coincides with its latest version in the local database. If any change occurs, the state switches to *modified*. An untracked file or a (tracked) modified file gets *staged* if it is explicitly marked by the user to go into the next snapshot with the add command. More specifically, the add command adds or updates the target resource in the *staging area* (or *index*) of the repository, which is a special hidden file containing information about what will be included in the next snapshot. Basically, the staging area contains all currently tracked files, with a flag for each file that has been modified since the last snapshot. A *commit* operation (commit command) creates a new snapshot that incorporates all the changes specified in the staging area, and stores this snapshot in the local database. As a result, the staging area will be *cleaned*, i.e. flags indicating modified staged files will be removed, and all the previously (modified) staged files will get committed (i.e. unmodified). Git users share information and collaborate with each other through remote repositories on servers, which can be synchronized with their own local repositories. Pushing and pulling data to and from remote servers are typical tasks when working on a shared project. The main commands to work with remote repositories are: clone, fetch, pull and push. The clone command creates a full-fledged working copy of the target repository, including a remote repository somewhere in Internet. The fetch command can be used to pull down all the resources from a remote project that are not yet in the (local) working directory, without modifying or merging the current local content with the downloaded data. While the pull command is similar to fetch, except that it also tries to automatically merge the downloaded data into the current local content. Finally, the push command can be used to push any local commit back up to the server, so as the local and remote projects are in sync.

2.2 Blockchain ed Ethereum

Una **blockchain** è un registro condiviso e immutabile che facilita il processo di registrazione delle transazioni e di tracciamento degli asset in una rete di business. Un asset è un qualsiasi elemento avente un valore, esso può essere rintracciato e scambiato su una rete blockchain, riducendo i rischi e i costi per tutti gli interessati. Ogni volta che un asset viene registrato e ne viene dunque registrata la sua transazione, essa prende la forma di un blocco, il nome della tecnologia deriva dal fatto che questo registro è essenzialmente una serie di blocchi collegati sequenzialmente in maniera irreversibile, come una catena. Gli elementi chiave di una blockchain sono il registro distribuito e accessibile a chiunque

faccia parte della stessa rete dove le transazioni vengono registrate un'unica volta, dei record immutabili che non permettono ad alcun partecipante alla rete di poter essere modificati o manomessi, anche in presenza di errori, e infine, anche se assenti da alcune reti come quella Bitcoin, la presenza dei cosiddetti **Smart Contract** , ovvero dei set di regole programmabili che permettono anche la creazione di vere e proprie applicazioni che operano sul registro leggendo e aggiungendo nuovi blocchi alla catena.
I vantaggi di questa tecnologia sono di certo non trascurabili, soprattutto in tempi recenti con l'aumento esponenziale della quantità di transazioni che vengono eseguite sia nelle reti pubbliche sia nelle private. In un sistema centralizzato abbiamo uno spreco di risorse per la convalida di terze parti e per la memorizzazione di record duplicati, aspetti assenti in una rete dotata di blockchain dove chiunque è sicuro che nessuna transazione è stata eliminata o manomessa e si fida della validità dei dati confidando nel fatto che ogni partecipante ha la copia esatta e sincronizzata del registro nella sua macchina. Questa tecnologia può sicuramente essere implementata all'interno di un'azienda ed essere privata, l'incentivo a mantenere la sua esistenza è il profitto a cui un corretto utilizzo di essa può portare con un incremento della produttività e un minore utilizzo di risorse. Tuttavia il mantenimento di una blockchain pubblica e accessibile a tutti richiede un incentivo, un insieme distribuito di individui deve mettere a disposizione parte della potenza computazionale e dell'archivio della propria macchina per permettere a questo registro di esistere e alle transazioni di essere eseguite e calcolate correttamente. Non a caso la tecnologia della blockchain è nata assieme a quella delle criptovalute: essa doveva avere lo scopo di "libro mastro" proprio per Bitcoin, mantenendosi grazie a persone disposte a mettere a disposizione il proprio hardware in cambio di un compenso proprio nella criptovaluta della rete.

Nasce da queste tecnologie il progetto **Ethereum**, con l'obiettivo di mettere però più carne al fuoco: esso trascende il concetto di semplice criptovaluta andando a creare una piattaforma decentralizzata per la creazione e la pubblicazione peer-to-peer di Smart Contracts in linguaggi di programmazione Turing completi (ovvero capaci di risolvere ogni problema che gli si possa presentare): Solidity e Vyper. Con Ethereum è quindi possibile andare a creare vere e proprie applicazioni con vita propria all'interno della **Ethereum Virtual Machine**, ovvero la grande macchina virtuale simulata andando ad unire tutta la potenza computazionale messa a disposizione dagli utenti di Ethereum e usando la blockchain come archivio di informazioni permanenti. Ovviamente sia la pubblicazione di uno Smart Contract sia la registrazione in generale di informazioni sulla blockchain all'interno di reti pubbliche come quella thereum ha un costo pecuniario proporzionale alla quantità di informazioni che si andranno a registrare.

2.3 Accumulatori crittografici e Merkle Tree

I **Merkle Tree** sono una tipologia di **accumulatori crittografici**, ovvero strumenti che permettono di comprimere molti elementi informativi in una costante di dimensione fissa, in altre parole ci permettono di rappresentare più blocchi di dati con un singolo hash. Nel particolare, i Merkle Tree, nello specifico quelli binari, sono essenzialmente alberi binari in cui ogni foglia corrisponde all'hash di uno dei nostri elementi, risalendo verso la radice ogni nodo interno calcolerà il proprio hash concatenando gli hash dei nodi figli, infine si avrà una radice (**Merkle Root** o MR) il cui hash è univoco a quella lista di hash che l'albero ha come foglie, in quella sequenza. Inoltre, utilizzando degli hash generati con una funzione crittografica "forte", si ha un'assenza di collisioni tra le Merkle Root. Perciò sappiamo che, per una determinata sequenza di documenti, anche solo una piccola modfica ad un file causerebbe il cambiamento totale della MR. Possiamo quindi capire che c'è stato un cambiamento, tuttavia per capire anche quale dei documenti è stato cambiato bisogna ricorrere al concetto di **Merkle Proof**. Per effettuare una verifica tramite Merkle Proof sono tre gli elementi necessari:

1. L'elemento (foglia) che vogliamo verificare
2. La Merkle Root
3. La Merkle Proof, ovvero la lista degli hash dei fratelli lungo il cammino dall'elemento alla matrice

Andando a svolgere questa verifica su ogni documento riusceremo ad individuare i file modificati come quelli per cui non è possibile ricostruire il cammino verso la radice lasciandola inalterata.

2.4 JSON

Il JavaScript Object Notation (**JSON**) è un semplice formato per lo scambio di dati, facile da interpretare e capire sia per i vari linguaggi di programmazione che per gli esseri umani. Esso, con le librerie apposite per ogni linguaggio, permette un semplice e rapido scambio di dati tra più applicativi e fornisce metodologie per la conversione di oggetti e collezioni di dati strutturati in stringhe da salvare in file e viceversa, un'ottima alternativa ai database per applicazioni che cercano di sviluppare architetture distribuite.

3. Il Problema e L'Obiettivo

Come già affermato in precedenza il sistema Git, nonostante la sua completezza e complessità, non fornisce ai suoi fruitori la possibilità di un controllo d'integrità sulle repository da esso create e gestite, un'operazione essenziale per alcune organizzazioni dove è cruciale che i dati non vengano manomessi o corrotti.
Tali organizzazioni potrebbero aver bisogno di tornare a versioni precedenti dei loro documenti con la certezza che essi siano stati ripristinati correttamente oppure di trasferire a terzi insiemi e sottoinsiemi di file e directory del loro file system facendo si che questi ultimi possano in ogni momento controllare l'integrità di ogni singolo documento.
La possibilità di appoggiarsi ad un sistema centralizzato è a questo punto quella che offrirebbe una verifica meno sicura: usando dei database le informazioni potrebbero essere eliminate o manomesse con maggiore facilità, da qui l'idea di appoggiarsi ad una struttura come quella della blockchain (sezione 2.2). Tutto ciò deve però essere implementato con un occhio di riguardo alla quantità di informazioni che verranno salvate su di essa: come sappiamo più byte vogliamo memorizzare più la nostra operazione sarà costosa dal punto di vista pecuniario.

3.1 L'obiettivo del nostro sistema

Il sistema progettato ha lo scopo di riuscire a fornire a chi ne usufruisce un livello di astrazione aggiuntivo sopra il software Git tramite un'interfaccia user-friendly che gli permetta non solo di gestire le sue directory come normali *repository* (sezione 2.1), ma fornisca anche degli utili strumenti di salvataggio di hash su blockchain, esportazione di sottoinsiemi di repository e verifica sia di singoli file che di moltitudini. Tutto ciò implementato con operazioni più o meno severe, a discrezione dell'utente, permettendo anche di impedire il ricalcolo di determinati insiemi di file con controllo sullo storico dei *commit* (sezione 2.1) o, volendo spendere di più, anche su Blockchain.

3.2 La questione della memorizzazione

Come già menzionato, la quantità di informazioni che andremo a memorizzare nella blockchain è direttamente proporzionale alla quantità di denaro che spendiamo per effettuare una registrazione. Occorre perciò trovare uno stratagemma per poter memorizzare una quantità estremamente ridotta di dati che contengano però intrinsecamente una grande quantità di informazione.
La soluzione al dilemma arriva sotto forma di accumulatori crittografici (sezione 2.3). Utilizzeremo delle strutture ad albero in cui le foglie saranno le nostre unità informative di interesse e da queste ne ricaveremo una radice unica. Ovviamente la loro struttura dovrà essere tale da permetterci di andare a reperire informazioni passate e già calcolate in un tempo che sia relativamente ragionevole.

4. Il Software PineSU

La concretizzazione della soluzione al problema esposto è l'applicativo **PineSU**.
PineSU si presenta come un software leggero scritto in Javascript e che sfrutta il runtime Node.js. Il software va a considerare gli insiemi di file come delle entità chiamate **Storage Unit** (SU) con cui va ad avvolgere logicamente una repository Git.
Queste SU sono le singole unità su cui si andranno ad effettuare le singole operazioni eccetto la registrazione su blockchain che si svolgerà collettivamente con l'ausilio di accumulatori crittografici.

Vedremo come il ciclo di vita di una SU sia scandito dai **Blockchain Synchronization Point** (BSP), ovvero gli eventi che si generano quando si decide di andare a registrare lo stato e la presenza di una SU su blockchain inserendola, tramite gruppi di suoi simili chiamati **Storage Group** (SG), nella grande struttura chiamata **Merkle Calendar** (MC). Infine, la root di questo MC sarà salvata nella Blockchain, da qui in poi sarà possibile, in qualsiasi momento, ricostruire un MC in quel preciso istante in cui un nuovo BSP è stato creato e verificare se la sua root è presente.

4.1 Workflow

PineSU segue una filosofia di workflow che ricalca quella di Git, infatti tramite l'applicativo si può operare su singole Storage Unit, ognuna con uno stato potenzialmente diverso dall'altra. Ciò fa si che il workflow sia molto legato al ciclo di vita di una singola SU, la quale a seconda del suo stato permetterà di fare alcune operazioni anziché altre.

Possiamo vederne una rappresentazione in Fig. 4.1, si parte da una directory del nostro file system e, tramite PineSU, la si trasforma in una Storage Unit (*Create new Storage Unit*) calcolandone anche gli hash in modo da rendere già possibili registrazioni e verifiche. Dopo questa operazione potrebbero accadere i tre eventi che seguono:

1. Il contenuto della Storage Unit potrebbe venire modificato, a questo punto è necessario, se si vogliono registrare anche i nuovi cambiamenti, andare ad effettuare un ricalcolo (*Update SU*), dopo di ciò si torna alla possibilità dei tre eventi;

2. Potremmo scegliere di chiudere la nostra Storage Unit (*Close Storage Unit*), una SU è aperta (**Open**) se modificabile e ricalcolabile liberamente, chiusa (**Closed**) se la modifica, il ricalcolo e un'eventuale chiusura successiva non possono essere eseguiti, dopo la chiusura potranno essere eseguite in pratica le stesse operazioni eccetto, ovviamente, quella del ricalcolo;

3. Potremmo scegliere di sincronizzare la SU nella blockchain (*Synchronize with blockchain*), questo avviene in realtà tramite due fasi che vedremo in seguito, tuttavia questa scelta ci permette in seguito di eseguire qualsiasi operazione lo stesso, eccetto il ricalcolo nell'eventualità che questa SU sia stata chiusa precedentemente.

Una volta che una SU è stata registrata nella blockchain o chiusa si aprono altre due possibili azioni da poter compiere: la verifica d'integrità (*Integrity verification*), sia offline che online, l'esportazione di sottoinsiemi di file dalla SU (*Export verifiable (sub)bundle*), tutti muniti con i dati necessari per effettuare verifiche d'integrità da parte di terzi anche senza disporre dell'intera SU.

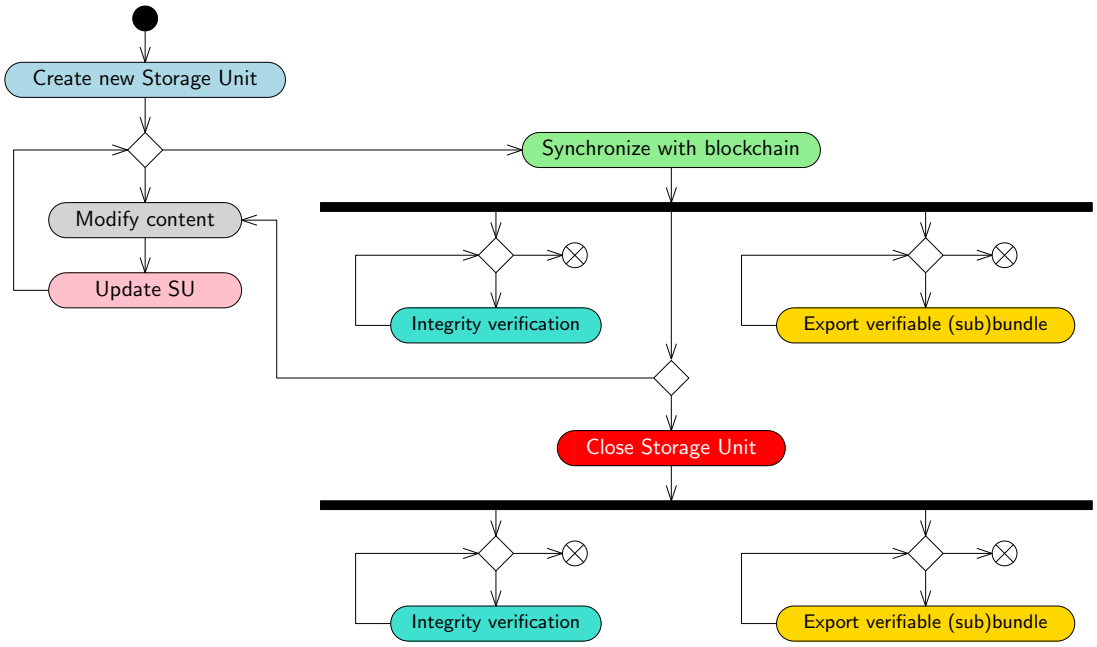

Figura 4.1: Workflow dell'applicativo sotto forma di Activity Diagram.

Possiamo osservare il workflow appena descritto anche in Fig. 4.2, dove il diagramma degli stati di una singola storgae unit riflette le operazioni appena descritte.

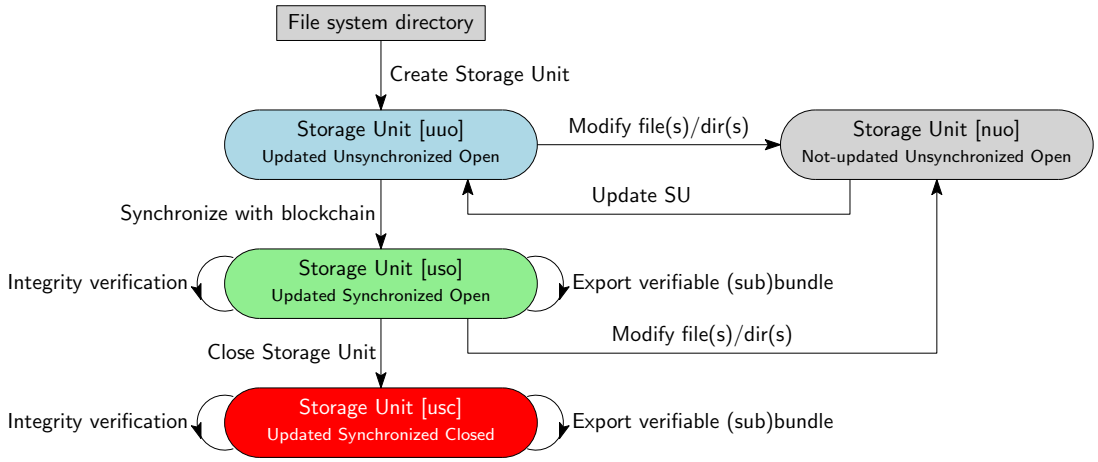

Figura 4.2: Ciclo di vita di una Storage Unit sotto forma di State Diagram.

4.2 Architettura

Il sistema va ad interfacciarsi con il client Git e con l'API web3.js per la comunicazione con la blockchain Ethereum, possiamo descrivere la sua architettura come in Fig. 4.3, dove troviamo i componenti principali:

- *PineSU* ***CLI*** *(Command Line Interface)*. L'interazione con PineSU da parte degli utenti avviene attraverso un emulatore di terminale di una macchina avente NodeJS installato[1], analogamente a come avviene con Git, tuttavia con un'interazione più guidata. Oltre al permettere l'uso delle normali funzioni di PineSU, questo modulo permette anche l'inserimento di un qualsiasi comando di Git in modo da rendere l'utilizzo diretto di quest'ultimo non necessario durante una tipica sessione di lavoro.

- *PineSU* ***BEL*** *(Back End Logic)*. Questo componente è il nucleo di PineSU. Gestisce tutte le SU e controlla la comunicazione con la blockchain e il client Git locale. Il client Git viene utilizzato inoltre per interagire indirettamente con i server Git remoti. Il modulo BEL, inoltre, si occupa dei due Storge Group, Open (OSG) e Closed (CSG), e mantiene il Merkle Tree dinamico chiamato Merkle Calendar che permette di recuperare efficientemente l'hash registrato in blockchain per un qualsiasi BSP. La gestione del salvataggio remoto del Merkle Calendar avviene tramite una repository Git scelta dall'utente.

[1] L'interfaccia si basa sul modulo npm Inquirer.js `https://github.com/SBoudrias/Inquirer.js`.

- *PineSU **EC** (Ethereum Connector)* Si interfaccia con il modulo *web3.js*[2].
- *PineSU **GC** (Git Connector)* Si interfaccia con il modulo *simple-git*[3].
- *PineSU **SM** (Smart Contract)* Questo modulo entra in gioco solamente nel caso di una registrazione "forte" di una Storage Unit nella blockchain, in una fruizione standard dell'applicativo non entrerà probabilmente in azione.

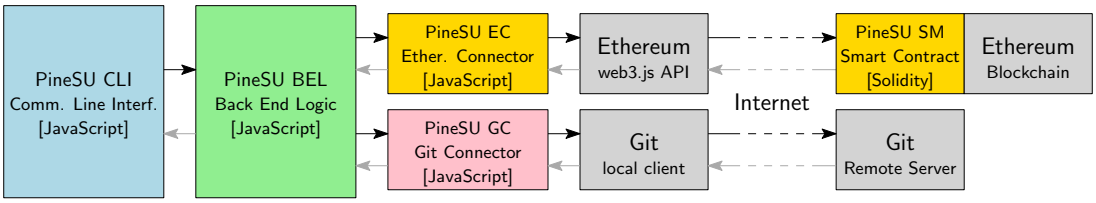

Figura 4.3: Rappresentazione dell'architettura ad alto livello di PineSU. Le frecce nere sono messaggi scatenati dalle entità sorgente corrispondenti mentre le frecce grigie sono risposte passive dell'entità interrogata.

4.3 Moduli in dettaglio

4.3.1 PineSU CLI

Il modulo si occupa di creare l'effettiva interfaccia utente con cui è possibile interagire, le domande vengono create dal modulo apposito "inquirer", dove sono definite insieme alle risposte possibili e ai controlli di consistenza delle risposte date dall'utente.

Dopo un setup una tantum in cui all'utente vengono chieste informazioni come gli indirizzi dei suoi due wallet, a seconda delle scelte dall'utente, la prima di cui sarà quella dell'effettiva operazione da eseguire (Fig. 4.4), il modulo va a delineare il workflow preciso che descrive il ciclo vitale di una SU, il tutto richiamando all'occorrenza le librerie del modulo *PineSU BEL*.

[2] `https://github.com/ChainSafe/web3.js`.
[3] `https://github.com/steveukx/git-js`.

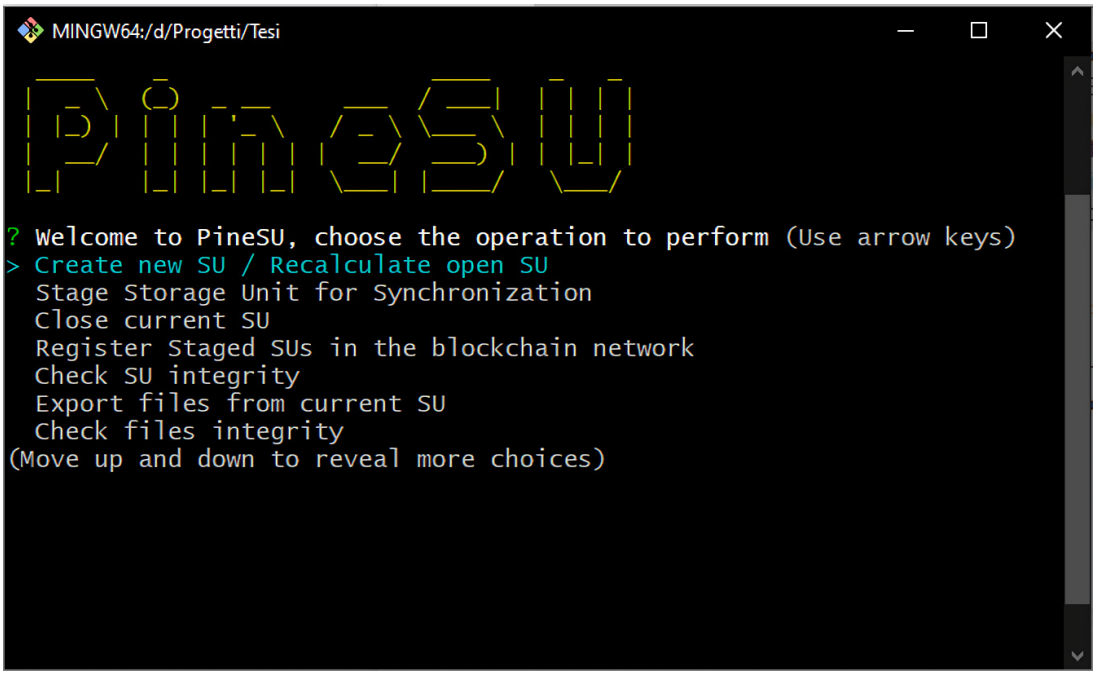

Figura 4.4: Menù principale dell'applicativo con la scelta dell'operazione da effettuare.

4.3.2 PineSU BEL

Questo modulo è il nucleo centrale del software, si occupa dell'effettiva comunicazione con i connettori per Git e per la blockchain, del gestire il File System andando a leggere e scrivere i file all'interno delle Storage Unit, di assegnare stringhe crittografiche ai singoli file e di creare e gestire le strutture di accumulazione crittografica.

Per la prima delle operazioni sopra citate troviamo due librerie, rispettivamente **GitLogic** e **EthLogic**, le quali si occupano essenzialmente di creare oggetti delle rispettive classi di connettori, reperire tramite altre librerie le informazioni necessarie, chiamare le funzioni dei connettori con gli input dovuti e gestire gli output in maniera coerente con ciò che necessita *PineSU CLI*.

Andiamo ora a vedere le due classi che si occupano di creare e gestire le strutture di memorizzazione che il programma utilizza: **Files** e **TreeList**. Files si occupa della lettura di file JSON, la lettura dei file di cui andrà letta la stringa hash corrispondente e della scrittura dei file JSON. Questi file JSON corrispondono ai descrittori delle SU (riguardanti la creazione e la registrazione su blockchain), alle informazioni sull'utente utilizzatore e alle strutture degli accumulatori crittografici.

TreeList si occupa del reperimento delle stringhe di hash e del calcolo dei Merkle Tree binari. Essi sono fondamentali per varie operazioni che vanno dal calcolo degli hash delle directory al quello della creazione dei tre accumulatori crittografici che andremo ora a descrivere.
Il primo è il semplice SU Merkle Tree, necessario per il calcolo dell'hash corrispondente alla Storage Unit, che altro non è che un Merkle Tree binario in cui ogni foglia corrisponde a un file o una directory contenuta nella SU. Questo MT verrà utilizzato anche nella fase di esportazione per generare le proof dei file esportati che serviranno per un eventuale controllo d'integrità singolo.
Il secondo è lo Storage Group, di cui ne esistono due, un **Open** (OSG) e un **Closed** (CSG). Si tratta essenzialmente di due MT binari in cui ogni foglia corrisponde all'hash di una Storage Unit **Staged**, la differenza tra i due alberi è nel loro contenuto, uno contiene le SU Open, l'altro le SU Closed, differenza già discussa nella sezione 4.1. Le root di questi alberi verranno poi salvate all'interno della prossima struttura come foglie.
Il terzo e il più importante è il **Merkle Calendar**, formato da due sottoalberi in cui vengono accolte come foglie rispettivamente le istanze di OSG e di CSG. Le radici di questi due sottoalberi hanno come figli dei nodi corrispondenti agli anni, i quali a loro volta hanno come figli dei nodi corrispondenti ai mesi, i figli dei mesi saranno infine le foglie corrispondenti a ciò che chiamiamo **Blockchain Synchronization Point** (BSP) in quanto nodi contenenti un timestamp e la root dello SG corrispondente.
Possiamo vedere, anche grazie a Fig. 4.5, come questa struttura sia implementata tramite tre classi:

- **MerkleCalendar**: contiene i riferimenti alle due radici dei sottoalberi e mette a disposizione varie funzioni per la ricerca di hash e reperimento di determinati valori della root in un certo BSP.
- **InternalCalendar**: corrisponde a un nodo interno dell'albero, quando si aggiungono figli si può scegliere di effettuare il ricalcolo del loro hash in modo da non doverlo calcolare successivamente (semplifica l'operazione di reperimento dell'hash di un certo BSP).
- **LeafCalendar**: corrisponde a una foglia dell'albero.

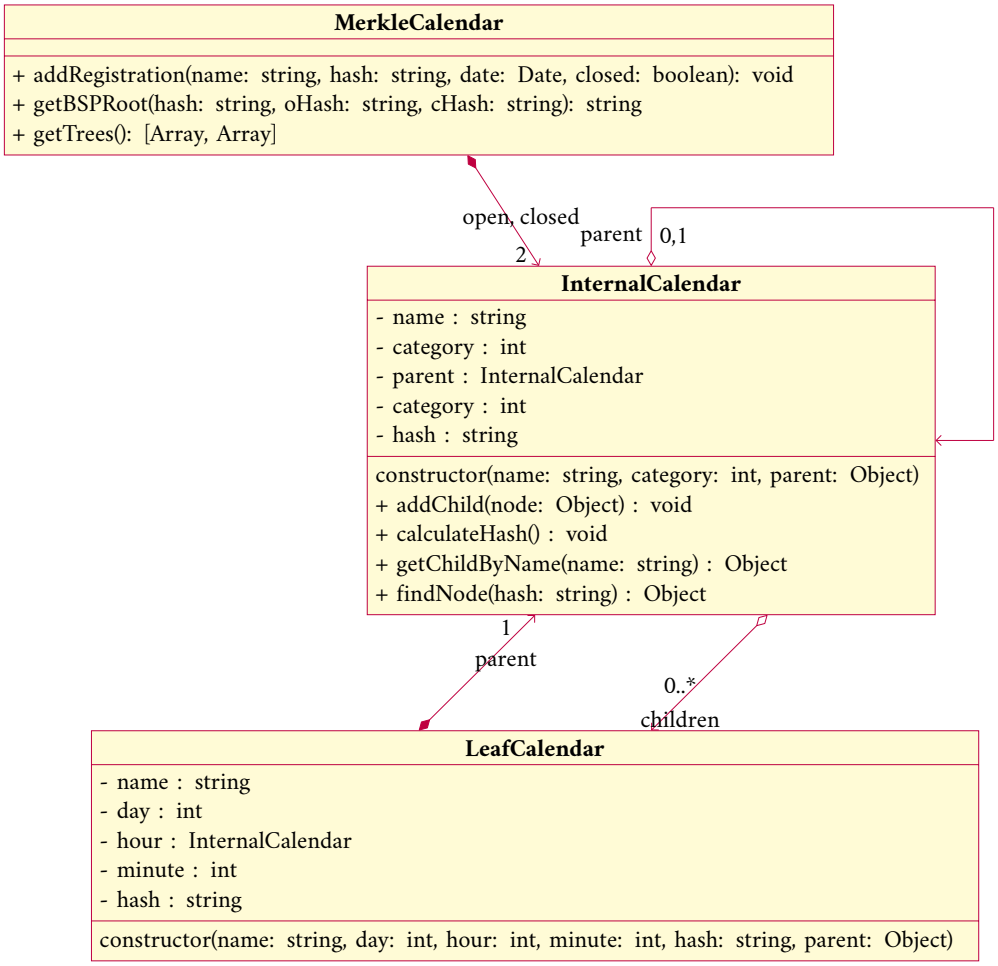

Figura 4.5: Rappresentazione UML delle classi descritte

4.3.3 Excursus sul Merkle Calendar

In Fig. 4.6 possiamo osservare un possibile Markle Calendar. Nell'esempio è stato esplorato il mese di Febbraio 2021 e lì troviamo tre BSP: esse corrispondono a tre registrazioni della root dell'intero MC su blockchain, ovviamente sempre diversa dato che l'hash root si calcola con gli hash delle sue foglie.

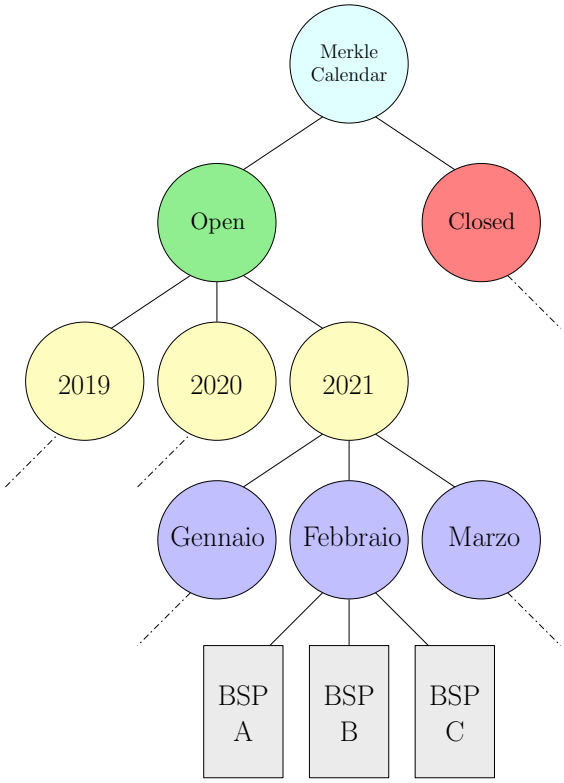

Figura 4.6: Rappresentazione grafica di un Merkle Calendar.

Ogni volta che un nuovo BSP viene inserito per la prima volta si avvia la procedura di ricalcolo dell'hash di ognuno dei suoi antenati in modo da non dover effettuare il calcolo dell'hash di gran parte dell'albero nella fase di reperimento.

Nel momento in cui ognuna di queste tre BSP è stata inserita nell'albero essa era la foglia più giovane, ergo se volessimo calcolare la root del MC in una data BSP ci basterebbe non considerare tutto ciò che è stato inserito successivamente, andando ad escludere una partizione di estrema destra dal sottoalbero radicato in Open (in questo caso).

Ipotizziamo di voler recuperare la MC root nel momento in cui è stata inserita BSP B:

1. Calcoliamo l'hash di Febbraio tramite l'hash di BSP a e di BSP B;

2. Calcoliamo l'hash del 2021 tramite l'hash di Gennaio e il "nuovo" hash di Febbraio;

3. Calcoliamo l'hash di Open tramite gli hash del 2019, del 2020 e il "nuovo" hash del 2021;

4. Calcoliamo la root del MC tramite l'hash di Closed e il "nuovo" hash di Open, l'hash di Closed nell'istante dell'inserimento di BSP B viene salvato in un descrittore JSON presente in ogni SU registrata con BSP B.

Il calcolo di hash tramite altri hash è effettuato tramite dei piccoli Merkle Tree binari "usa e getta", alla stessa maniera di come calcoliamo l'hash root di uno Storage Group (che andrà a finire nei BSP foglia dei MC).

4.3.4 PineSU EC

Come visualizzabile in Fig. 4.7 il connettore per la blockchain, in questo caso specifico quello della rete Ethereum, è composto da una semplice classe che, con i suoi attributi che corrispondono ad un oggetto del modulo web3.js, i due indirizzi dei wallet e la chiave privata del primo, svolge le operazioni di effettuare una transazione e verificarne una precedente.

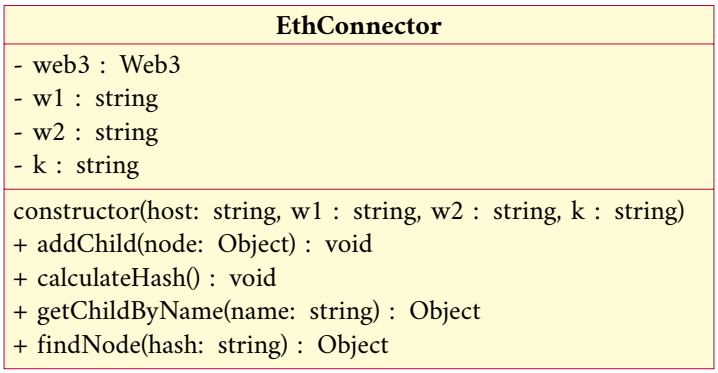

Figura 4.7: Rappresentazione UML di EthConnector

4.3.5 PineSU GC

Il connettore per Git, la cui rappresentazione UML è visualizzabile in Fig. 4.8, è una semplice classe che lavora con un attributo della classe proveniente dal modulo simple-git, essa prende in input una directory su cui lavorare ed è poi in grado, in base alle chiamate delle funzioni del connettore, di operare su di essa tramite il client Git installato sulla macchina.

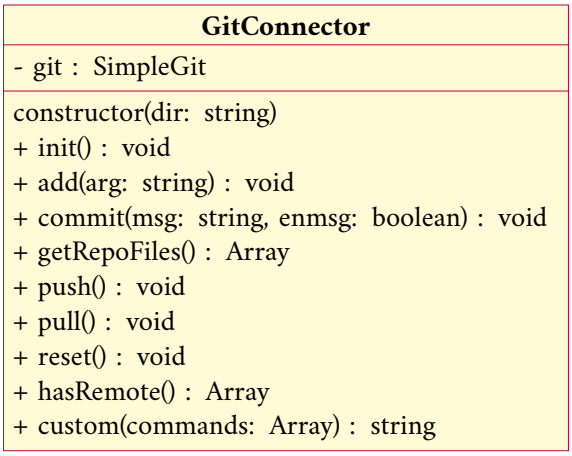

Figura 4.8: Rappresentazione UML di GitConnector

4.3.6 PineSU SM

Lo Smart Comtract (sezione 2.2) di PineSU è necessario per poter effettuare registrazioni "forti", ovvero andare a salvare su blockchain che una determinata SU è stata chiusa in modo da evitare che delle eventuali manomissioni cerchino di chiuderla nuovamente (ricordiamo che una SU chiusa implica un'impossibilità di modifica e ricalcolo). Questo modulo è presente solamente in caso si vadano ad utilizzare blockchain di criptovalute che supportano la presenza di Smart Contract, nel nostro caso, con Ethereum, siamo a posto. Questo modulo non è tutt'ora implementato ed è destinato a sviluppi futuri **??**.

4.4 Funzionalità

L'elenco delle funzionalità è il seguente.

1. Creazione di una Storage Unit o Ricalcolo di una Storage Unit pre-esistente
2. Staging di una Storage Unit nello Storage Group
3. Registrazione dello Storage Group nella Blockchain
4. Chiusura di una Storage Unit
5. Esportazione di sottoinsiemi di file da una SU
6. Controllo di integrità di singoli file esportati da altre SU
7. Controllo di integrità su una SU

Alle operazioni che implicano modifiche alla struttura o allo stato della SU seguirà sempre un Git commit. Andremo ora ad analizzarle una per volta.

4.4.1 Creazione di una Storage Unit o Ricalcolo di una Storage Unit pre-esistente

La creazione di una Storage Unit è in realtà un'operazione che comprende sia la trasformazione in una Git Repository della directory, sia il calcolo degli hash che serviranno poi per registrare la nostra SU nella blockchain. Le informazioni della nostra SU sono conservate nel file descrittore **.pinesu.json** nella root della directory, la presenza di questo file indica al programma che la directory è già una SU. Anche nel caso in cui .pinesu.json sia già presente nella directory questa operazione può essere eseguita e provvederà al ricalcolo degli hash e la creazione di un nuovo descrittore, questo ovviamente solo se la SU è aperta. Dei file possono essere esclusi dalla Storage Unit con l'ausilio di un semplice file *.gitignore*, la cui creazione viene anch'essa gestita, in maniera opzionale, dal software.

4.4.2 Staging di una Storage Unit nello Storage Group

Lo hash principale della Storage Unit viene inserito in uno dei due Storage Group (a seconda che sia aperta o chiusa), questa operazione è stata nominata *staging* in quanto è concettualmente simile all'operazione omonima di Git se consideriamo la registrazione nella Blockchain analoga ad un *commit*.

4.4.3 Registrazione dello Storage Group nella Blockchain

Le root dei due Storage Group vengono inserite nel Merkle Calendar, si effettua una transazione tra due wallet contenente la nuova root del Merkle Calendar. Gli SG vengono poi svuotati e le proof per essere dinamicamente ricostruiti vengono salvate, insieme alle informazioni per reperire la transazione, in un descrittore JSON nelle directory delle varie Storage Unit appena registrate.

4.4.4 Chiusura di una Storage Unit

La Storage Unit viene chiusa ma ad una condizione che varia a seconda dei casi:

- *Weak*: Si controllano i commit della repository per controllare che un commit di chiusura sia già avvenuto.
- *Strong*: Si controlla la blockchain, tramite PineSU SM (sottosezione 4.3.6), per verificare che sia presente la entry corrispondente alla chiusura di quella SU.

4.4.5 Esportazione di sottoinsiemi di file da una SU

In questa fase all'utente viene data la possibilità di scegliere di esportare alcuni file dalla SU, viene creato un file **.pifiles.json** in cui si salva, per ogni file esportato, il suo percorso originale, il suo hash, l'hash della root e le *proof* per calcolare la root dato l'hash del file (ciò servirà nell'operazione di verifica d'integrità). Infine i file, seguendo la struttura in cui comparivano nella SU originale, e .pifiles.json vengono compressi in un file ZIP e salvati nella cartella precedente a quella in cui si sta operando.

4.4.6 Controllo di integrità di singoli file esportati da altre SU

Viene analizzata la directory in modo da trovare dei file descrittori **.pifiles.json**, da quelli e dai file in essi elencati si fa un controllo d'integrità che va anche ad effettuare lo stesso controllo su Blockchain che si effettua nell'ultima operazione.

4.4.7 Controllo di integrità su una SU

Si legge la root dello Storage Group della SU selezionata e si cerca tale root nel Merkle Calendar, una volta trovato si è in grado di ricalcolare la root del Merkle Calendar nel momento in cui tale Storage Group è stato registrato, da qui si controlla se la transazione salvata contiene anch'essa la root del Merkle Calendar come messaggio.

5. Dimostrazioni d'uso per il fine preposto

In questo capitolo andremo a testare l'applicativo mostrando cosa accade nel file system ogni volta che PineSU esegue una funzione. Prenderemo in esame una situazione in cui creeremo due SU, una verrà lasciata aperta, ò'altra verrà prima registrata aperta, poi chiusa e registrata nuovamente, in questo modo riusciremo anche ad osservare i cambiamenti sull'intero Merkle Calendar.

6. Conclusioni e Sviluppi Futuri

7. Bibliografia

[1] Git-scm.com. Git, Accessed: July, 2021.