

Tesi di laurea?

Paolo Speziali

Condividere informazioni in modo sicuro combinando Git e Blockchain

Relatore:

Prof. Luca Grilli

Tesi di laurea in Ingegneria Informatica

No! ↴

Perugia, Anno Accademico 2020/2021

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



A.D. 1308 —
unipg

DIPARTIMENTO
DI INGEGNERIA

0. Indice

1	Introduzione	4
2	Concetti Preliminari	5
2.1	Funzioni di hash	5
2.2	VCS e Git	6
2.3	Blockchain ed Ethereum	7
2.4	Accumulatori crittografici e Merkle Tree	9
2.5	JSON	10
3	Il Problema e L'Obiettivo	11
3.1	L'obiettivo del nostro sistema	11
3.2	La questione della memorizzazione	12
4	Il Software PineSU	13
4.1	Workflow	13
4.2	Architettura	15
4.3	Moduli in dettaglio	17
4.3.1	PineSU CLI	17
4.3.2	PineSU BEL	18
4.3.3	Excursus sul Merkle Calendar	21
4.3.4	PineSU EC	22
4.3.5	PineSU GC	23
4.3.6	PineSU SM	23
4.4	Funzionalità	24

INDICE

4.4.1	Creazione di una Storage Unit o Ricalcolo di una Storage Unit preesistente	24
4.4.2	Staging di una Storage Unit nello Storage Group	25
4.4.3	Registrazione dello Storage Group nella Blockchain	25
4.4.4	Chiusura di una Storage Unit	25
4.4.5	Esportazione di sottoinsiemi di file da una SU	25
4.4.6	Controllo di integrità di singoli file esportati da altre SU	26
4.4.7	Controllo di integrità su una SU	26
5	Dimostrazioni d'uso per il fine preposto	27
5.1	Prima inizializzazione	27
5.2	Creazione delle Storage Unit	27
5.3	Staging delle Storage Unit	31
5.4	Registrazione su Blockchain	31
5.5	Verifica d'integrità di una Storage Unit	33
5.6	Esportazione di file da una Storage Unit	34
5.7	Chiusura di una Storage Unit	35
5.8	Verifica d'integrità di file esportati	36
6	Conclusioni e Sviluppi Futuri	38
7	Bibliografia	40

L3 + introd - prelim - biff

7 codice

Sbarco problema → esp. tecnologie → Combiniamo → soluz.!

SVILUPP. SOLUZIONI AL FINE DI

1. Introduzione

ACCELERARE LA
DIGITALIZZAZIONE E
RIDURRE I COSTI
(BUROCRATI)

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 Funzioni di hash

Una **funzione di hashing** [1] è un algoritmo che calcola una stringa di lunghezza fissa a partire da una sequenza di dati di lunghezza variabile, come un file, che altro non è che un insieme di blocchi di dati. L'operazione di hashing trasforma questi dati in una stringa di lunghezza fissa, chiamata **hash** o stringa di hash, il cui valore rappresenta la stringa originale. Il valore della stringa può essere considerato il riassunto di tutto il contenuto del file. Un hash è di solito una stringa esadecimale.

Un buon algoritmo di hashing gode di una proprietà chiamata *effetto valanga*, ovvero la stringa di output cambia in maniera significativa o anche interamente anche quando un singolo bit o byte dell'insieme di dati in input viene modificato. ~~Inoltre~~ l'algoritmo deve essere complesso abbastanza da non produrre lo stesso output per due diversi input, se ciò avviene siamo in presenza di una *collisione*. Una possibilità molto bassa di collisione è spesso indice di un algoritmo solido. Lo hashing è un'operazione unidirezionale, dalla stringa di hash non si può risalire al file originale. Uno dei tanti usi dello hashing è quello di confrontare due file per controllare se il loro contenuto è lo stesso, in caso affermativo produrranno, per la stessa funzione, lo stesso hash, possiamo quindi controllare in maniera immediata l'uguaglianza senza dover necessariamente controllare l'intero file parola per parola. Grazie a ciò è anche possibile controllare l'**integrità** di un file, se è stato infatti

identificato

Ogni bit ha 50% di modifiche

in particolare
a quelle critografiche
byte per byte

fare subversion con

5

funzioni critografiche di hash

modificato o manomesso esso produrrà una stringa di hash differente rispetto a quella che dovrebbe produrre.

2.2 VCS e Git

Un **Version Control System** [2] (o anche VCS), in italiano “sistema di controllo di versione”, 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 [3]. 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 [4] è il sistema di controllo di versione distribuito più diffuso al mondo. Esso modella ogni repository come una *sequenza* o *flusso di snapshot* (istantanei) di un piccolo file system. Ogni volta che un utente salva lo stato del suo progetto (tramite l’operazione di *commit*) Git crea uno snapshot di tutti i file e le directory sotto controllo di versione in quel momento e la archivia nel suo database locale, ogni file modificato dall’ultimo commit viene incluso nell’ultimo snapshot, mentre i file che non sono stati modificati non vengono inclusi se non con un collegamento alla loro versione identica nel commit precedente, in modo da evitare alcuna duplicazione non necessaria. Ogni risorsa in una repository è identificata internamente dal suo hash (sezione 2.1) e non dal suo nome, questo permette a Git di individuare efficientemente i cambiamenti nei file. Inoltre, quasi ogni operazione di Git va ad aggiungere informazioni al suo database, anche se si tratta di un’operazione di rimozione, ciò assicura che ogni cambiamento sia reversibile.

Ogni file in una directory assume uno dei questi due stati: *untracked* (non tracciato) o *tracked* (tracciato). Un file è *untracked* se non è stato mai aggiunto ad una repository o se è

stato aggiunto ma poi rimosso dalla lista dei file tracciati (comando rm). Un file *tracked*, ovvero l'esatto opposto di un *untracked*, può assumere a sua volta uno di questi tre stati: *unmodified* (non modificato o *committed*), *modified* (modificato) e *staged*. Un file *tracked* è *unmodified* quando coincide con la sua ultima versione nel database. Se qualsiasi cambiamento avviene, diventa *modified*. Per diventare *staged* è necessario che l'utente utilizzi su di lui il comando add, in questo modo esso viene inserito (o aggiornato se era già presente) nella *staging area* (o *index*) della repository, essa contiene tutti i file tracciati della repository con una flag che indica se sono stati modificati o meno dall'ultimo snapshot. L'operazione di *commit* (comando commit) crea un nuovo snapshot che incorpora tutti i cambiamenti specificati nella staging area e lo immagazzina nel suo database locale. A questo punto la staging area verrà ripulita (*cleaned*). Gli utenti Git possono condividere informazioni e collaborare tra di loro tramite repository remote su server Git che possono essere sincronizzate con le loro repository locali.

Le operazioni di *pull*, *push*, *clone* e *fetch* sono tipiche quando si lavora con repository remote. Il comando clone crea una copia esatta di una repository target, incluso il suo database di snapshot. Il comando fetch permette di scaricare le risorse di un progetto remoto che non sono presenti in quello locale, senza però andare a modificare i file già presenti applicando eventuali modifiche. Il comando pull è simile a fetch, eccetto che tenta di eseguire una fusione automatica del file remoto e del file locale applicando a quest'ultimo le modifiche più recenti. Infine, il comando push consente di inviare ogni nuovo commit locale al sever remoto, in modo da mantenerli sincronizzati.

*Non solo gli asset gli elementi fondamentali, parlare della decentralizzazione
2.3 Blockchain ed Ethereum Prendere dalle dispense del*

Una **blockchain** [5] [6] è un registro condiviso e immutabile che facilita il processo di registrazione delle transazioni e di tracciamento degli asset in una rete ~~anonimale e non~~ ^{di blockchain natura} ~~anonimale e non~~ PHD aggiungendo fiducia.

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 in realtà registrata la transazione che si riferisce ad esso, 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 l'effettivo registro distribuito e accessibile a chiunque faccia parte della stessa rete, dove le

dei blocchi
contengono
la
transazione
e la validano

[*dove?*]

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.

→ 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 essere implementata senza problemi 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" per Bitcoin, mantenendosi grazie a persone disposte a mettere a disposizione il proprio hardware in cambio di un compenso nella criptovaluta propria della rete.

→ Nasce da queste tecnologie il progetto **Ethereum** [7] [8], 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 Contract 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

BC più inefficiente, costosa, lenta, ricca
di eritano "node-trusted point", il potere è democratico

pubblicazione di uno Smart Contract sia la registrazione in generale di informazioni sulla blockchain all'interno di reti pubbliche come quella Ethereum ha un costo pecuniario proporzionale alla quantità di dati che si andranno a registrare.

2.4 Accumulatori crittografici e Merkle Tree

I **Merkle Tree** [9] 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. 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 elementi 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 modifica ad un file causerebbe un cambiamento significativo, se non 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.

2.5 JSON

Il JavaScript Object Notation (**JSON**) [10] 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 appropriate 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'alternativa ragionevole ai database per applicazioni che cercano di sviluppare architetture distribuite.

Esempio di JSON

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 sì che questi ultimi possano in ogni momento controllare l'integrità di ogni singolo documento.

*Ti deni
fidare -
del sistema
Centralizzato.*

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 2.3). 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 2.2), ma fornisca anche degli utili strumenti di salvataggio di *hash* (sezione 2.1) 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 2.2) 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 ~~contenga però~~ permetta di ~~intrinsecamente una grande mole di informazione.~~ *un gran numero di controlli di integ.*

espletare La soluzione al dilemma arriva sotto forma di accumulatori crittografici (sezione 2.4). 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

lightweight

La concretizzazione della soluzione al problema esposto è l'applicativo **PineSU** [11]. PineSU si presenta come un software ~~leggero~~ scritto in Javascript e che sfrutta il runtime Node.js. L'applicazione va a considerare gli insiemi di file come delle entità chiamate **Storage Unit** (SU) con cui va ad ~~avere~~ logicamente una repository Git. *inglobare*

Queste SU sono le 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 all'interno gruppi di suoi simili chiamati **Storage Group** (SG), i quali finiranno poi nella grande struttura chiamata **Merkle Calendar** (MC). Infine, la root di questo MC sarà salvata nella blockchain (nell'implementazione attuale, quella Ethereum), 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 root di questa ricostruzione è presente nel registro.

*Contenuto interno
attraverso
metadati*

*migliora
integrazione*

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 sì che il workflow sia molto legato al ciclo di vita di una singola SU, la qualcuna a seconda del suo stato, permetterà di fare alcune operazioni anziché altre.

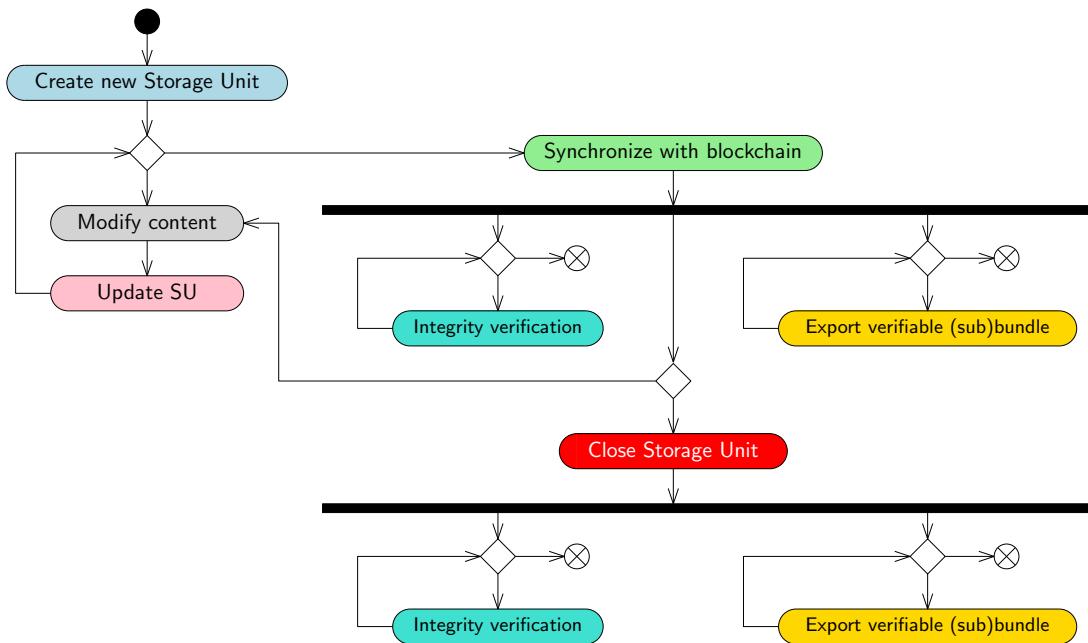


Figura 4.1: Workflow dell'applicativo sotto forma di Activity Diagram.

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:

- i metadati*
- Non qui va più l'ra unità con una nuova regne*
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, e l'esportazione di sottoinsiemi di file dalla SU (*Export verifiable (sub)bundle*), munendoli con i dati necessari per permettere verifiche d'integrità da parte di terzi anche senza disporre dell'intera SU.

Possiamo osservare il workflow appena descritto anche in Fig. 4.2, dove il diagramma degli stati di una singola Storage Unit riflette le operazioni appena descritte. Appena creata possiamo infatti riferirci ad essa come una Storage Unit **Updated Unsynchronized Open**, ovvero aperta e non sincronizzata alla blockchain, tuttavia aggiornata in quanto gli hash sono stati calcolati sulla situazione corrente dei file contenuti. Sincronizzandola diventa una SU **Updated Synchronized Open**, su cui possiamo eseguire operazioni di verifica ed esportazioni senza cambiare però il suo stato. Modificando un file esternamente, in entrambi gli stati che abbiamo visto, si entra in uno stato da cui si può uscire solamente eseguendo un ricalcolo: ovvero quello di SU **Not-updated Unsynchronized Open**. Una chiusura unita con una registrazione su blockchain fanno diventare la SU **Updated Synchronized Closed**, uno stato in cui la modifica non è permessa.

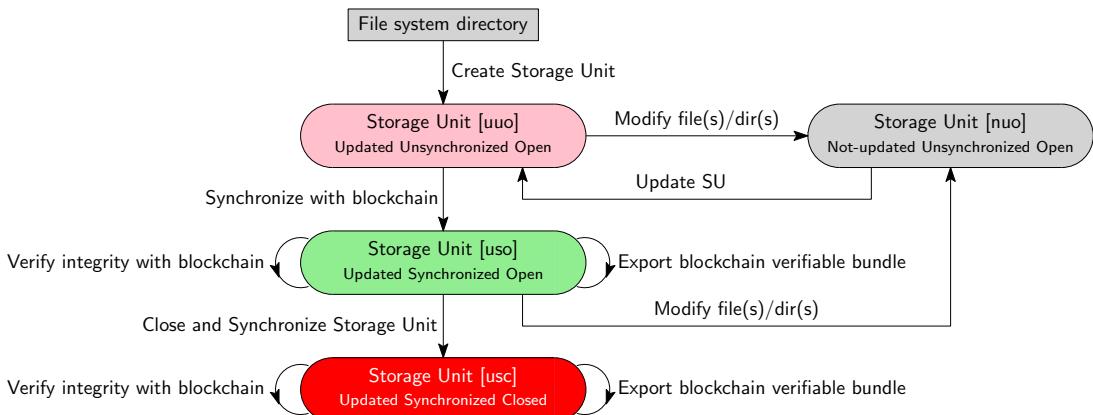


Figura 4.2: Ciclo di vita di una Storage Unit sotto forma di State Diagram.

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, analogamente a come avviene con Git, di una macchina avente Node.js installato¹, tuttavia con un’interazione più guidata rispetto al noto VCS. 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 PineSU². 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, in aggiunta, si occupa della gestione 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.
- *PineSU EC (Ethereum Connector)* Si interfaccia con il modulo web3.js [16] [17].
- *PineSU GC (Git Connector)* Si interfaccia con il modulo simple-git [18].
- *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.

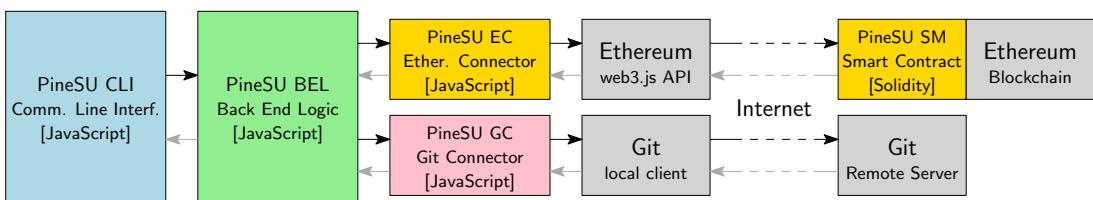


Figura 4.3: Rappresentazione dell’architettura ad alto livello di PineSU. Le frecce nere sono messaggi scatenati dalle entità sorgente corrispondenti mentre le frecce grigie sono risposte passive dell’entità interrogata.

¹L’interfaccia si basa sul modulo npm Inquirer.js [12]

²Il componente si basa su alcuni moduli npm per operazioni di supporto [13] [14] [15]

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 dell'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*.

```
MINGW64:/d/Progetti/Tesi
? Welcome to PineSU, choose the operation to perform (Use arrow keys)
> Create new SU / Recalculate open SU
Stage Storage Unit for Synchronization
Close current SU
Register Staged SUS in the blockchain network
Check SU integrity
Export files from current SU
Check files integrity
(Move up and down to reveal more choices)
```

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 e scrittura di file JSON e della lettura dei file di cui andrà calcolata la stringa di hash corrispondente. 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 alla 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** (ovvero preparata per la registrazione collettiva, ne parlremo meglio nella sottosezione 4.4.2), 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 più importante è il **Merkle Calendar**, formato da due sottoalberi in cui vengono accolte come foglie rispettivamente le root delle istanze di OSG e di CSG. Le

*zuhui
meglio*

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.

Per ogni BSP, sappiamo che c'è stata la registrazione su blockchain della blockchain della root dell'MC in quel momento. 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.

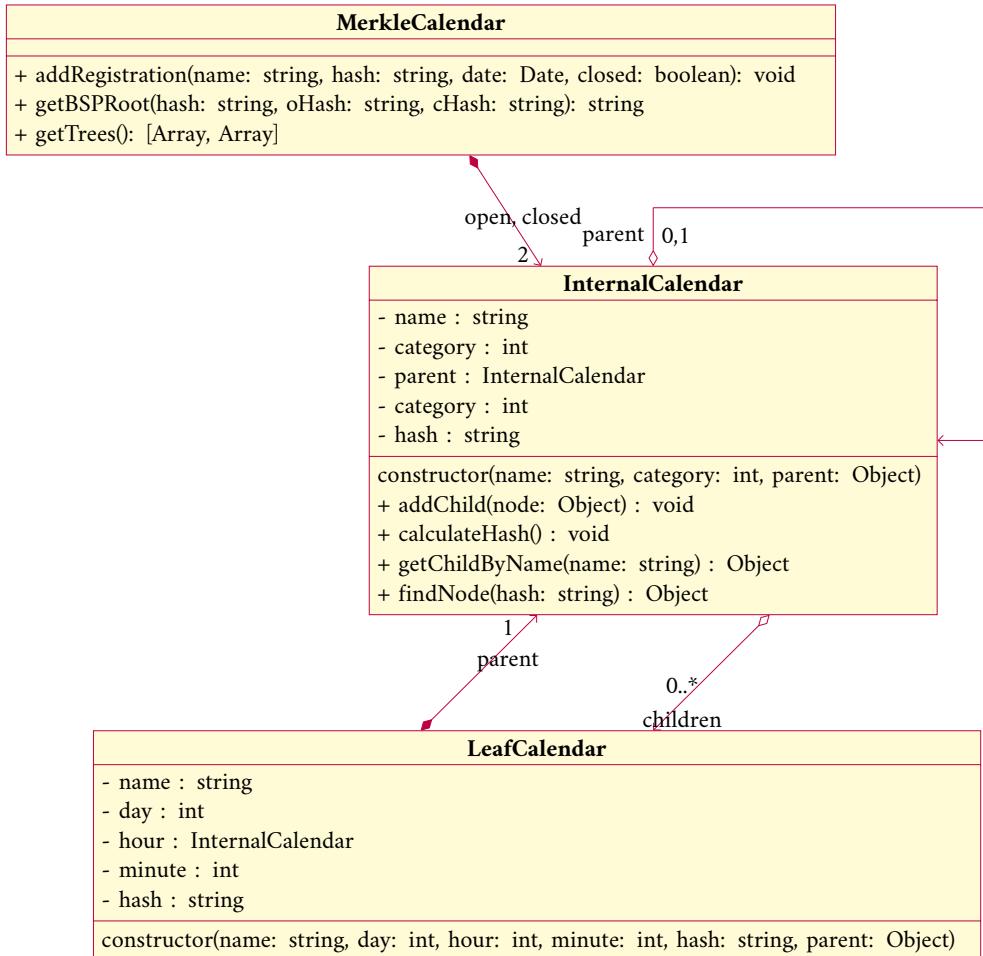


Figura 4.5: Rappresentazione UML delle classi descritte

4.3.3 Excursus sul Merkle Calendar

In Fig. 4.6 possiamo osservare un possibile Merkle 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.

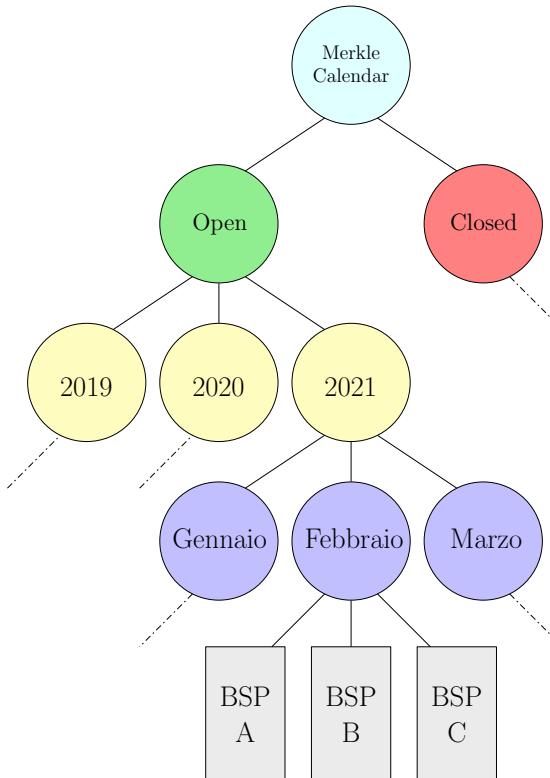


Figura 4.6: Rappresentazione grafica di un Merkle Calendar.

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).

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.

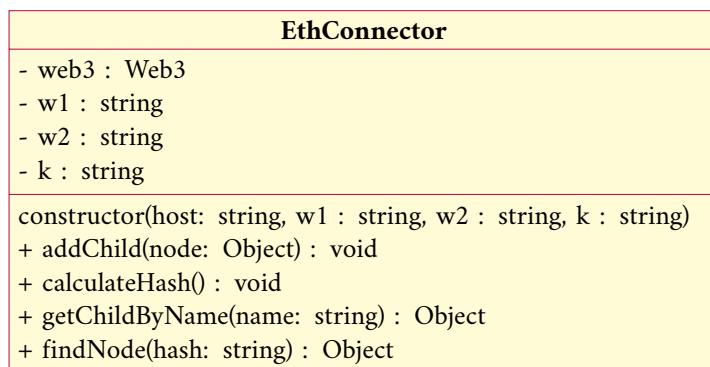


Figura 4.7: Rappresentazione UML di EthConnector

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.

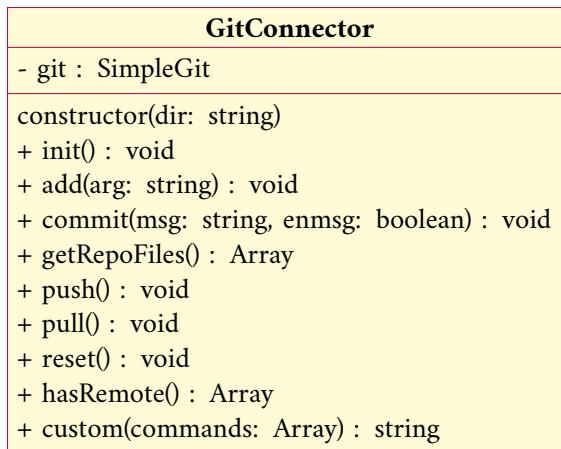


Figura 4.8: Rappresentazione UML di GitConnector

4.3.6 PineSU SM

Lo Smart Comtract (sezione 2.3) 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. PineSU SM non è tutt’ora implementato ed è destinato a sviluppi futuri (Capitolo 6).

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 (sezione 2.2). 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 alla 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 non sia già avvenuto.
- *Strong*: Si controlla la blockchain, tramite PineSU SM (sottosezione 4.3.6), per verificare che non sia già 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 vecchio 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, l'altra verrà prima registrata aperta, poi chiusa e registrata nuovamente, in questo modo riusciremo anche ad osservare i cambiamenti sull'intero Merkle Calendar.

5.1 Prima inizializzazione

Alla prima apertura di PineSU sulla macchina il processo ci andrà a chiedere quattro valori: due indirizzi di Wallet Ethereum, la chiave privata del primo dei wallet e, opzionalmente, la repository Git remota con cui sincronizzare il MerkleCalendar

5.2 Creazione delle Storage Unit

Partiamo definendo il contenuto delle nostre due directory, la prima, **sample**, ha questa struttura:

- sample/graphCreator.js
- sample/first/astar.js
- sample/first/graph.js

CAPITOLO 5. DIMOSTRAZIONI D'USO PER IL FINE PREPOSTO

- sample/second/priorityQueue.js
- sample/second/third/main.js
- sample/second/third/vertex.js

Dove first e second sono due subdirectory di sample e third è una subdirectory di second. I file contenuti sono dei file plain text salvati in formato JavaScript.

La seconda directory, **secondSample**, ha questa struttura:

- secondSample/esonero1/Immagine.png
- secondSample/esonero1/preesonero.pdf
- secondSample/esonero1/preesonero.tex
- secondSample/esonero2/preesonero2.pdf
- secondSample/esonero2/preesonero2.tex
- secondSample/esonero3/preesonero3.pdf
- secondSample/esonero3/preesonero3.tex
- secondSample/esonero3.smith-chart.png

Dove esonero1, esonero2 ed esonero3 sono tre subdirectory di secondSample. In questa directory troviamo anche la presenza di file di diversa natura.

Per trasformare le directory in Storage Unit posizioniamoci con il terminale all'interno di ognuna, avviamo lo script di avvio di PineSU e selezioniamo la prima opzione (*Create / Recalculate SU*), ovviamente il procedimento andrà ripetuto due volte. Il processo ci chiederà se inizializzare una repository Git e se escludere alcuni file, procederà poi al calcolo dei vari hash.

CAPITOLO 5. DIMOSTRAZIONI D'USO PER IL FINE PREPOSTO

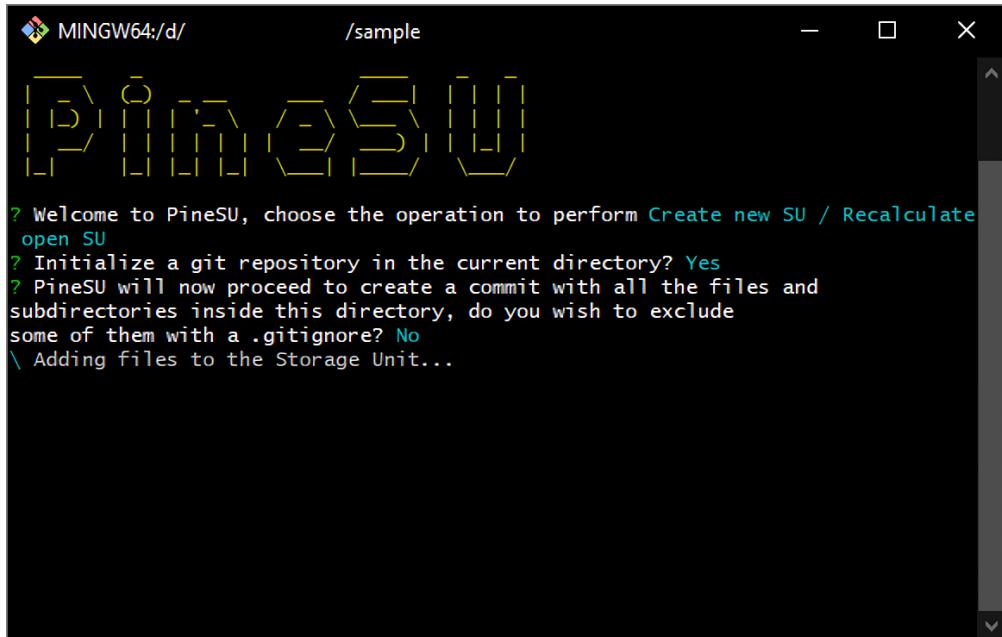


Figura 5.1: L'interfaccia di PineSU durante il calcolo.

Una volta finito di calcolare ci verranno fatte domande sulla natura della Storage Unit come il suo nome, la repository remota a cui si sincronizza, la sua descrizione, ecc... Finite le domande, l'applicativo ci riporterà al menù principale.

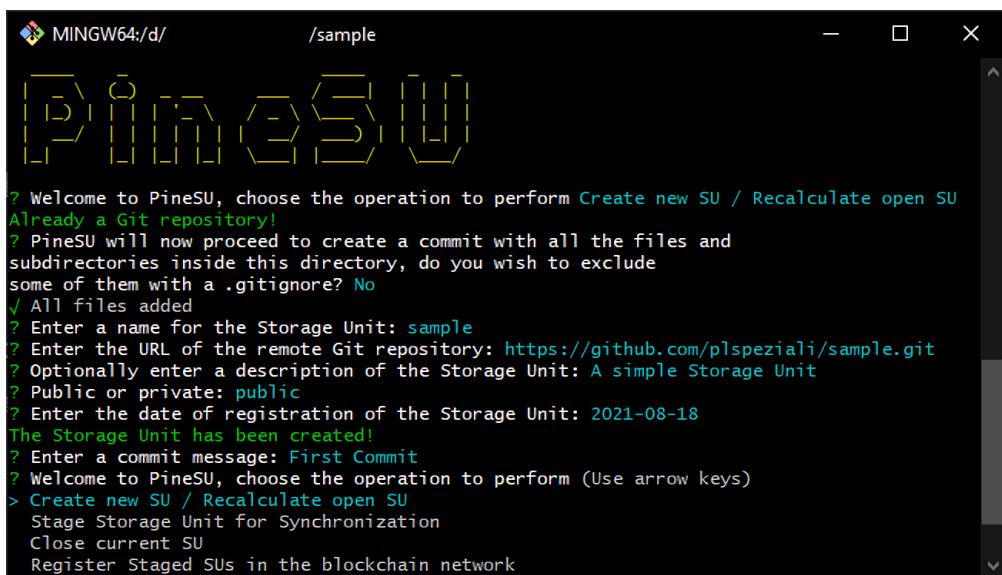


Figura 5.2: L'interfaccia di PineSU appena terminata la fase di creazione.

Al termine di entrambe le elaborazioni sulle due directory avremo alcune nuove aggiunte al loro interno: una cartella .git (eventualmente, anche un file .gitignore) e un file .pinesu.json, quest'ultimo è il descrittore JSON in cui sono salvate le informazioni che abbiamo inserito, gli hash calcolati e lo stato di chiusura. Ecco, ad esempio, il file JSON di sample:

```

1 {"name": "sample",
2 "remote": "https://github.com/plspeziali/sample",
3 "description": "A simple Storage Unit",
4 "visibility": "public",
5 "date": "2021-08-18",
6 "owner": "0xCF23544bFC002905532bD86bF647754A84732966",
7 "hash": "3837ec6fe66032ba593d227ee800a079c61a5853",
8 "filelist": [
9     "first/astar.js:
10        e09deffb3654301a9a8d20acc5a7091cda7039b6",
11     "first/graph.js:53982
12        e4feeaf1445434864b409014706e31da1cc",
13     "graphCreator.js:
14        c3b4338c33d6ea5a30b31c16defc7661a4ae767b",
15     "second/priorityQueue.js:1
16        cb66abaaafd6a7125ab7dac1d7e0fb1860da574",
17     "second/third/main.js:1
18        b8dad338691dead8edc66e7c01b8db6d834e3d8",
19     "second/third/vertex.js:
20        aa3fa9242ceec7062c7d84764e4068711e53c4e3",
21     "first:75d936d52208d14c2cd571e0c595bc29e7d0e3a0",
22     "second:75d936d52208d14c2cd571e0c595bc29e7d0e3a0",
23     "second/third:2
24        ec86afc483f3685893831dfa04b66620be690d2",
25     "second:d6e51cdb8a84cb3ba2c9cfdc2773a96b2401a59"
26 ],
27 "closed": false }
```

fallo meglio

5.3 Staging delle Storage Unit

A questo punto vogliamo registrare le nostre Storage Unit aperte nella blockchain, occorre però prima inserirle negli Storage Group (in questo caso solo in Open Storage Group). Per effettuare questa operazione occorre solo selezionare l'opzione apposita (*Stage SU for Synchronization*) in entrambe le directory, nella nostra cartella d'installazione verrà aggiornato il file `merkles/storageGroup.json` con le informazioni delle nostre SU.

```
1 [ {
2     "name": "sample",
3     "hash": "3837ec6fe66032ba593d227ee800a079c61a5853",
4     "path": "D:/sample",
5     "closed": false
6 },
7 {
8     "name": "secondSample",
9     "hash": "7a61ed5e43cd436fb1f88895625a8193fdb9b3be",
10    "path": "D:/secondSample",
11    "closed": false
12 } ]
```

Questa è la lista delle foglie con cui calcoleremo l'effettivo Merkle Tree di OSG.

5.4 Registrazione su Blockchain

Per la registrazione non è necessario posizionarsi in una delle directory delle SU in quanto quelle da registrare sono già nello Storage Group, ci rimane solo da selezionare l'opzione apposita (*Register Staged SUs*) per eseguire con la registrazione, a questo punto lo Storage Group verrà svuotato e verrà popolato, almeno nel sottoalbero Open, il Merkle Calendar con una foglia in più e, all'occorrenza, dei nodi interni extra (in caso non fossero state registrate delle SU in questo mese e anno).

Ovviamente di conseguenza la nuova Merkle Root del Merkle Calendar verrà registrata su blockchain Ethereum inserendola in un messaggio scambiato tra i due wallet forniti dall'utente, ciò non è visualizzabile dall'utente a meno che non vada nelle directory delle SU registrate in cui potrà trovare un nuovo file `.registration.json`, esso conterrà

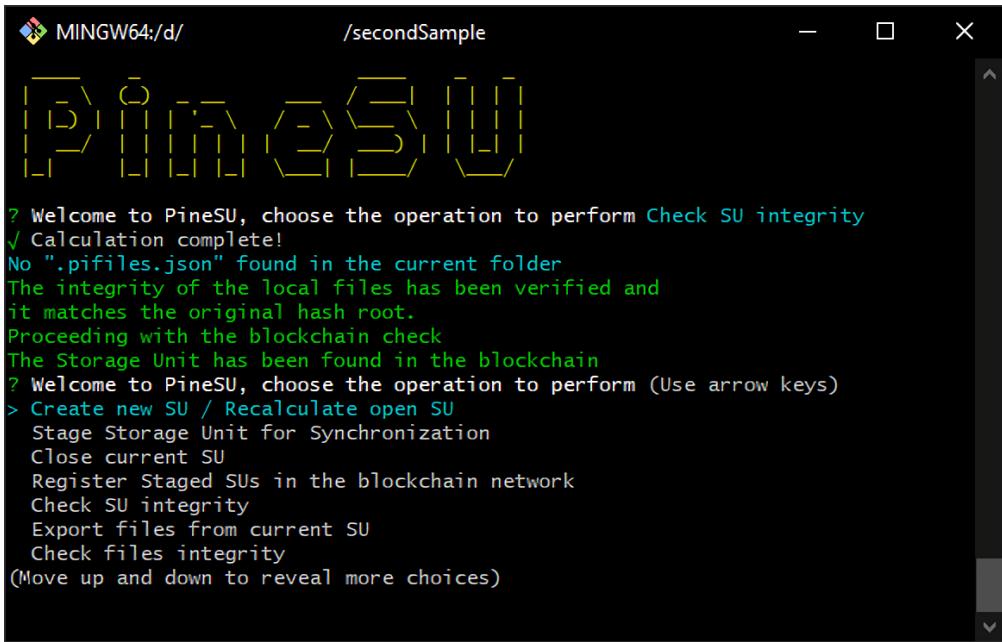
CAPITOLO 5. DIMOSTRAZIONI D'USO PER IL FINE PREPOSTO

informazioni come la root dello Storage Group in cui è stato registrato, la proof per ricostruire tale root, l'hash della transazione per recuperarla dalla catena e gli hash dei sottoalberi Open e Closed del MerkleCalendar nel momento in cui è stato registrato.

```
1 { "path": "D:/Progetti/Tirocinio/sample",
2 "root": "e67006f15ecd3fa2719d148be68d3a3242e1be8b",
3 "proof": [ {
4     "right": "7a61ed5e43cd436fb1f88895625a8193fdb9b3be"
5 } ],
6 "transactionHash": "0xc56303032[...]2dc87d2e",
7 "oHash": "e67006f15ecd3fa2719d148be68d3a3242e1be8b",
8 "cHash": null }
```

5.5 Verifica d'integrità di una Storage Unit

A questo punto ci troveremo ad avere Storage Unit USO (Updated Synchronized Open), ciò significa che ha senso sia effettura verifiche d'integrità avvalendoci della blockchain, sia esportare sottoinsiemi di file da essa in modo che anche chi reperisce tali file sia in grado di verificare la loro integrità e la loro registrazione su blockchain. La verifica può essere fatta selezionando l'opzione *Check SU integrity* mentre ci si trova nella directory della SU da verificare, il successo o l'insuccesso ci verranno comunicati su schermo.



```
MINGW64:d/          /secondSample      -  □  X
[...]
? Welcome to PineSU, choose the operation to perform Check SU integrity
✓ Calculation complete!
No ".pifiles.json" found in the current folder
The integrity of the local files has been verified and
it matches the original hash root.
Proceeding with the blockchain check
The Storage Unit has been found in the blockchain
? Welcome to PineSU, choose the operation to perform (Use arrow keys)
> Create new SU / Recalculate open SU
Stage Storage Unit for Synchronization
Close current SU
Register Staged SUs in the blockchain network
Check SU integrity
Export files from current SU
Check files integrity
(Move up and down to reveal more choices)
```

Figura 5.3: L'interfaccia di PineSU appena terminata la fase di verifica con successo.

5.6 Esportazione di file da una Storage Unit

Per quanto riguarda l'esportazione di file, andremo ad eseguirla su secondSample. Essa viene eseguita selezionando i file da esportare con l'apposita interfaccia di selezione multipla

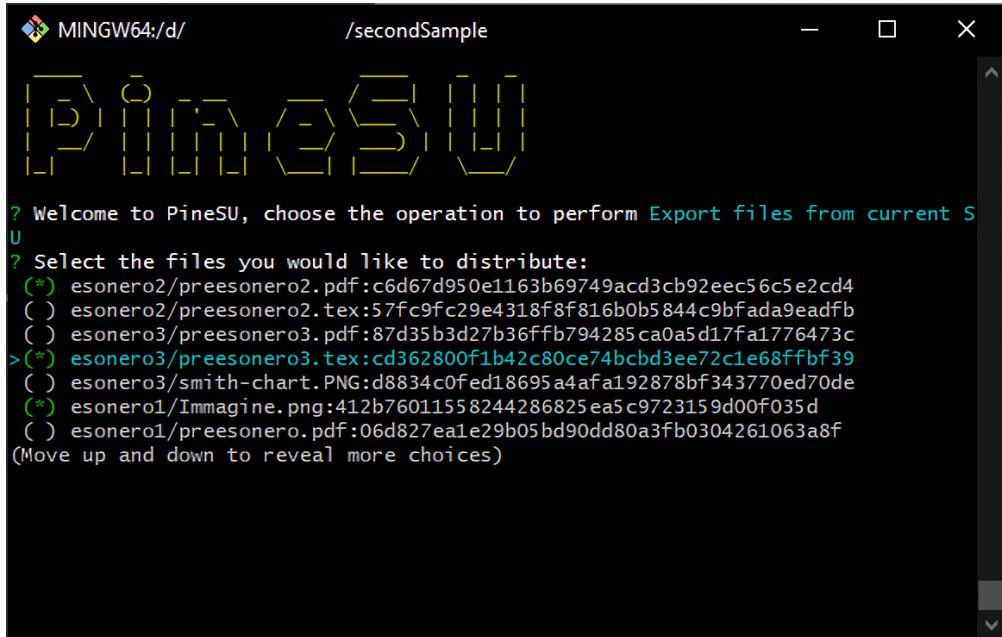


Figura 5.4: L'interfaccia di PineSU durante un'esportazione.

In questo caso siamo andati a selezionare `esonero2/preesonero2.pdf`, `esonero3/preesonero3.tex` e `esonero1/Immagine.png`, essi verranno inseriti, assieme a un file descrittore `.pifiles.json`, che contiene per ognuno le proof per ricostruire l'hash della loro SU a partire dai loro hash, e il file `.registration.json`, in un file ZIP che verrà salvato nella cartella superiore a quella in cui si trovano.

5.7 Chiusura di una Storage Unit

Abbiamo già parlato delle implicazioni della chiusura di una Storage Unit, ora descriveremo come effettivamente può essere realizzata. Di fatto basta eseguire PineSU nella directory della SU e selezionare l'opzione *Close current SU*, con questa opzione verrà aggiunto un commit con un messaggio particolare, infatti se proveremo a richiudere la stessa Storage Unit, anche se il file `.pinesu.json` venisse eliminato, tale commit non ci permetterà di richiuderla nuovamente.

Procediamo con l'esempio chiudendo `secondSample` e registrandola nuovamente su blockchain: in questo caso una foglia verrà aggiunta sia a Closed Storage Group che, successivamente, al sottoalbero Closed del Merkle Calendar. Dopo queste operazioni infatti il Merkle Calendar sarà questo:

```
1 "open": [ {
2     "name": 2021,
3     "hash": "e67006f15ecd3fa2719d148be68d3a3242e1be8b",
4     "children": [ {
5         "name": 7,
6         "hash": "e67006f15ecd3fa2719d148be68d3a3242e1be8b",
7         "children": [ {
8             "name": "SU of Wed Aug 18 2021 16:00:20",
9             "year": 2021,
10            "month": 7,
11            "day": 3,
12            "hour": 16,
13            "minute": 0,
14            "hash": "e67006f15ecd3fa2719d148be68d3a3242e1be8b"
15        } ]
16    } ],
17 } ],
```

```
18 "closed" : [ {
19     "name": 2021 ,
20     "hash": "7a61ed5e43cd436fb1f88895625a8193fdb9b3be" ,
21     "children": [ {
22         "name": 7 ,
23         "hash": "7a61ed5e43cd436fb1f88895625a8193fdb9b3be" ,
24         "children": [ {
25             "name": "SU of Thu Aug 19 2021 12:16:36" ,
26             "year": 2021 ,
27             "month": 7 ,
28             "day": 4 ,
29             "hour": 12 ,
30             "minute": 16 ,
31             "hash": "7a61ed5e43cd436fb1f88895625a8193fdb9b3be"
32         } ]
33     } ]
34 } ]
```

Come possiamo vedere entrambi i sottoalberi hanno due nodi interni (anno e mese) e la foglia corrispondente al BSP, quello in Open è stato registrato alla sezione 5.4, mentre quello in Closed è il BSP generato dalla registrazione della SU appena chiusa.

5.8 Verifica d'integrità di file esportati

Avevamo esportato, nella sezione 5.6, un sottoinsieme dei file di secondSample, una Storage Unit che nel frattempo è mutata chiudendosi, cambiando root e finendo in un altro sottoalbero di Merkle Calendar, tuttavia l'esportazione era avvenuta quando la SU era stata registrata già in blockchain come aperta. Dimostriamo come, nonostante tutto, la verifica su blockchain avvenga senza problemi, ci basterà infatti posizionarci in una directory arbitraria contenente i file estratti dal file ZIP e selezionare *Check files integrity* per farsi che il processo svolga i dovuti controlli e ci comunichi a schermo il successo o l'insuccesso.

CAPITOLO 5. DIMOSTRAZIONI D'USO PER IL FINE PREPOSTO

The screenshot shows a terminal window titled "MINGW64:/d/pinesuExport". The window contains the following text output from the PineSU application:

```
? Welcome to PineSU, choose the operation to perform Check files integrity
The integrity of the file esonero1/Immagine.png
has been verified and it matches the original hash root
The file esonero1/Immagine.png was verified
in being once part of a Storage Unit.
Proceeding with the blockchain check
The Storage Unit of the file esonero1/Immagine.png
has been found in the blockchain
The integrity of the file esonero2/preesonero2.pdf
has been verified and it matches the original hash root
The file esonero2/preesonero2.pdf was verified
in being once part of a Storage Unit.
Proceeding with the blockchain check
The Storage Unit of the file esonero2/preesonero2.pdf
has been found in the blockchain
The integrity of the file esonero3/preesonero3.tex
has been verified and it matches the original hash root
The file esonero3/preesonero3.tex was verified
in being once part of a Storage Unit.
Proceeding with the blockchain check
The Storage Unit of the file esonero3/preesonero3.tex
has been found in the blockchain
? Welcome to PineSU, choose the operation to perform (Use arrow keys)
> Create new SU / Recalculate open SU
  Stage Storage Unit for Synchronization
  Close current SU
  Register Staged SUs in the blockchain network
  Check SU integrity
  Export files from current SU
  Check files integrity
(Move up and down to reveal more choices)|
```

Figura 5.5: L'interfaccia di PineSU durante la verifica dei file esportati da secondSample.

Vediamo in Fig. 5.5 come, per ogni file, viene eseguito un controllo sia locale, per controllare che non sia stato manomesso, sia remoto, ovvero viene controllato che la SU di cui lui sostiene di far parte sia stata registrata in blockchain.

6. Conclusioni e Sviluppi Futuri

Abbiamo potuto osservare come, con il software descritto, si è risolto il problema della mancanza di una funzione di verifica d'integrità dei file su Git, andando a sfruttare le tecnologie del Web 3.0 come la Blockchain Ethereum, sempre con un occhio di riguardo al portafoglio dell'utilizzatore, servendosi di strutture dati peculiari progettate per permettere un salvataggio economico e una verifica veloce ed efficiente.

Il progetto ha già subito diverse revisioni e riscritture e si trova ora in uno stato ben definito e molto fedele alla descrizione fornita (tralasciando alcuni dettagli poco importanti), tuttavia, nonostante sia un programma decisamente completo sotto il punto di vista delle funzionalità, almeno per ciò che avevamo progettato di realizzare, sono presenti alcuni aspetti su cui si potrebbe ancora lavorare per rendere l'esperienza d'uso molto più adatta ad affrontare le esigenze lavorative di ogni giorno all'interno di grandi e piccole aziende.

In primis potrebbe essere migliorato il calcolo della Merkle Root corrispondente ad una singola Storage Unit, infatti per ora la lista di file e directory viene semplicemente ordinata in ordine alfabetico e da quella viene calcolato un Merkle Tree binario. Invece, ispirandosi al modo con cui Git traccia le modifiche dei propri file identificandoli con il loro hash (come spiegato nella sezione 2.2), si potrebbe identificare ogni file con l'hash corrispondente e tracciare le modifiche ogni volta che si effettua un ricalcolo della Storage Unit, evitando quindi di ricalcolare hash di file che sono rimasti identici, usando possibilmente le funzionalità dei Git commit.

Una seconda aggiunta è sicuramente la creazione del PineSU Smart Contract per la gestione di registrazioni “forti” su blockchain EVENTUALMENTE POTREI FARLO.

CAPITOLO 6. CONCLUSIONI E SVILUPPI FUTURI

Una seconda aggiunta è l'implementazione di connettori per blockchain, in modo tale da poter registrare le proprie SU su blockchain pubbliche differenti o, potenzialmente, anche su blockchain private.

Una terza aggiunta, decisamente più ambiziosa, sarebbe la creazione di una piattaforma per il salvataggio remoto di Storage Unit, un equivalente a ciò che servizi come GitHub, BitBucket e GitLab sono per Git. In questo modo si potrebbe venire incontro anche alle aziende in cui non è richiesta una grande competenza informatica da parte dei dipendenti (nonostante l'attuale implementazione di PineSU non si lasci intimidire sotto il punto di vista dell'essere user-friendly).

7. Bibliografia

Mettere fonti autorevoli.

- [1] C. Chung, “Introduction to Hashing and its uses,” Accessed: August, 2021. [<https://www.2brightsparks.com/resources/articles/introduction-to-hashing-and-its-uses.html>].
- [2] Wikipedia, “Controllo versione,” Accessed: August, 2021. [https://it.wikipedia.org/wiki/Controllo_versione].
- [3] M. Lubański, “Centralized vs Distributed Version Control Systems,” 2019. [<https://bit.ly/3sw1gh6>].
- [4] Git-scm.com, “Git,” Accessed: August, 2021. [<https://git-scm.com/>].
- [5] IBM, “What is Blockchain?,” Accessed: August, 2021. [<https://www.ibm.com/it-it/topics/what-is-blockchain>].
- [6] Wikipedia, “Blockchain,” Accessed: August, 2021. [<https://it.wikipedia.org/wiki/Blockchain>].
- [7] Ethereum.org, “Ethereum.org” Accessed: August, 2021. [<https://ethereum.org>].
- [8] Ethereum.org, “Ethereum Documentation,” Accessed: August, 2021. [<https://ethereum.org/en/developers/docs/>].
- [9] H. Qureshi, “Merkle Trees,” 2019. [<https://nakamoto.com/merkle-trees/>].
- [10] JSON.org, “JSON.org,” Accessed: August, 2021. [<https://www.json.org/json-it.html>].

CAPITOLO 7. BIBLIOGRAFIA

- [11] P. Speziali, “GitHub - plspeziali/PineSU : PineSU,” Accessed: August, 2021. [<https://github.com/plspeziali/PineSU>].
- [12] S. Boudrias, “GitHub - SBoudrias/Inquirer.js,” Accessed: August, 2021. [<https://github.com/SBoudrias/Inquirer.js/>].
- [13] The Brain, “GitHub - cthackers / adm-zip : ADM-ZIP for NodeJS,” Accessed: August, 2021. [<https://github.com/cthackers/adm-zip>].
- [14] Chalk, “GitHub - chalk / chalk : Chalk,” Accessed: August, 2021. [<https://github.com/chalk/chalk>].
- [15] Tierion, “GitHub - Tierion / merkle-tools : merkle-tools,” Accessed: August, 2021. [<https://github.com/Tierion/merkle-tools>].
- [16] ChainSafe Systems, “GitHub - ChainSafe/web3.js: Ethereum JavaScript API,” Accessed: August, 2021. [<https://github.com/ChainSafe/web3.js>].
- [17] ChainSafe Systems, “web3.js - Ethereum JavaScript API,” Accessed: August, 2021. [<https://web3js.readthedocs.io/en/v1.4.0/>].
- [18] Steve King, “GitHub - steveukx / git-js : Simple Git,” Accessed: August, 2021. [<https://github.com/steveukx/git-js>].