

Paolo Speciali

Condividere informazioni in modo sicuro combinando Git e Blockchain

Relatore:

Prof. Luca Grilli

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	7
2.1	Funzioni di hash	7
2.2	VCS e Git	9
2.3	Blockchain ed Ethereum	10
2.4	Node.js	13
2.5	Accumulatori crittografici e Merkle Tree	14
2.6	JSON	15
3	Il Problema e l'Obiettivo	17
3.1	L'obiettivo del nostro sistema	17
3.2	La questione della memorizzazione	18
4	Il Software PineSU	19
4.1	Workflow	19
4.2	Architettura	22
4.3	Moduli in dettaglio	23
4.3.1	PineSU CLI	23
4.3.2	PineSU BEL	24
4.3.3	Excursus sul Merkle Calendar	27
4.3.4	Excursus sullo Storage Group	28
4.3.5	PineSU EC	29
4.3.6	PineSU GC	30
4.3.7	PineSU SM	30

4.4	Funzionalità	32
4.4.1	Creazione di una Storage Unit o Ricalcolo di una Storage Unit pre-esistente	32
4.4.2	Staging di una Storage Unit nello Storage Group	35
4.4.3	Registrazione dello Storage Group nella Blockchain	35
4.4.4	Chiusura di una Storage Unit	37
4.4.5	Esportazione di sottoinsiemi di file da una SU	38
4.4.6	Controllo di integrità di singoli file esportati da altre SU	38
4.4.7	Controllo di integrità su una SU	39
5	Dimostrazioni d'uso per il fine preposto	41
5.1	Prima inizializzazione	41
5.2	Creazione delle Storage Unit	41
5.3	Staging delle Storage Unit	45
5.4	Registrazione su Blockchain	45
5.5	Verifica d'integrità di una Storage Unit	47
5.6	Esportazione di file da una Storage Unit	48
5.7	Chiusura di una Storage Unit	49
5.8	Verifica d'integrità di file esportati	50
6	Conclusioni e Sviluppi futuri	52
7	Bibliografia	54

1. Introduzione

Il passaggio da analogico a digitale, la cosiddetta “digitalizzazione”, è una delle più grandi rivoluzioni del secolo scorso sotto tutti i punti di vista. Un processo di certo lento e pieno di difficoltà che è però in grado di ricompensare ampiamente non solo chi riesce a formalizzarne il procedimento per uno specifico settore, ma anche tutti coloro che seguiranno ad usufruire delle risorse e i beni che hanno subito questa trasformazione. E nei settori non manca di certo quello più importante per le società del ventunesimo secolo, quello lavorativo. E chi sosteneva che la digitalizzazione delle aziende fosse qualcosa su cui ci si potesse permettere di prendere il proprio tempo è stato costretto a ricredersi con quel “pedale di accelerazione” che è stata la pandemia di COVID-19.

Costringendoci a stare tutti a casa una delle conseguenze fondamentali è stata, come ben sappiamo, la spinta involontaria di miliardi di individui verso metodi tecnologici capaci di assicurare continuità alle proprie attività lavorative e didattiche nonché utili a fronteggiare le proprie esigenze personali. Le aziende, sia pubbliche che private, sono state così costrette a reinventare il proprio modo di lavorare, sia nell’organizzazione del lavoro subordinato che nel modo in cui si rapportano con terze parti, che siano clienti, utenti o altre aziende. Tuttavia è stato un processo poco uniforme che ha lasciato molti elementi ancora in forma mista, uno tra tutti la componente burocratica. La burocrazia non è un problema semplice e non si può affrontare con una veloce scannerizzazione in formato PDF o con una produzione diretta in formato elettronico, molti di questi documenti hanno infatti un’importanza notevole all’interno dell’azienda e, in molti casi, anche valenza legale, bisogna perciò essere estremamente sicuri che le informazioni scritte su questi documenti non vengano manomesse o corrotte.

Un grande problema che si affronta di questi tempi e con questi temi estremamente caldi è come sviluppare strumenti informatici che permettano di salvare e trasferire informazioni e documenti in maniera sicura in modo tale da accelerare i tempi della transizione digitale della burocrazia minimizzandone anche i costi, il tutto sempre tenendo a mente che chi ne usufruisce non deve necessariamente avere conoscenze informatiche di alto livello.

Le tecnologie principali per la gestione delle informazioni e della loro autenticità si basano sui due paradigmi di progettazione: centralizzata e distribuita. La prima si basa essenzialmente sulla concentrazione dei dati, e quindi del potere, nelle mani di un'unità centrale, ciò può essere sicuramente vantaggioso dal punto di vista della rapidità dello svolgimento delle operazioni e del consumo di risorse (sia computazionali, sia di archiviazione, sia energetiche) ma se stiamo parlando di sicurezza e fiducia è naturale avere qualche perplessità, non solo stiamo immagazzinando informazioni critiche in un database centralizzato, potenzialmente vulnerabile ad attacchi e perdita di dati, ma stiamo anche fornendo tali informazioni a un'entità di cui sarà necessario avere una totale fiducia per quanto riguarda il corretto mantenimento dei nostri dati.

Complementarmente, il secondo paradigma, ovvero quello distribuito, si basa su una rete di sistemi interconnessi su cui le informazioni vengono replicate e sincronizzate in maniera reciproca, il tutto senza la necessità di un'entità centrale. Questa seconda soluzione è sicuramente più lenta, più costosa e più inefficiente, ma tuttavia più sicura in quanto l'autenticità dell'informazione registrata non è sostenuta da un'unica unità ma da una moltitudine, così facendo è molto più complicato che si manifestino attacchi o manomissioni o, perlomeno, che restino inosservati.

Tra le tecnologie distribuite, **Git** è sicuramente lo standard de facto per la condivisione e il tracciamento delle modifiche di insiemi di file ed è uno strumento essenziale per gli sviluppatori di software sia indipendenti che facenti parte di grandi progetti. Tuttavia, oltre all'essere uno strumento il cui utilizzo è estremamente raro tra chi non scrive codice informatico a causa della sua natura poco orientata verso l'utente medio, soffre anche della mancanza di una funzione per la protezione, registrazione e controllo d'integrità dei propri insiemi di file. Queste lacune esistono perché Git non è un software che si è mai posto obiettivi del genere, tuttavia si può pensare ad un applicativo che, utilizzando proprio Git, si pone questi obiettivi, dopotutto una funzione che permette la verifica di insiemi di file tramite un'architettura distribuita e un'interfaccia più semplice da usare sono due aspetti che potrebbero rendere questa applicazione un potente strumento per

venire in soccorso alla necessità di digitalizzazione della burocrazia. Rimane però il dilemma di come poter memorizzare le informazioni necessarie alla verifica dei documenti utilizzando strutture distribuite, ed è qui che ci viene d'aiuto la giovanissima tecnologia della **blockchain**. La blockchain, tramite i suoi meccanismi di registrazione e consenso, è una soluzione ideale per i problemi di verifica d'integrità nel mondo digitale: la natura plurale e democratica di questa tecnologia ci riporta sempre al paragone iniziale tra le architetture e ci fa arrivare di conseguenza alla medesima conclusione. Per quanto possa essere efficiente una struttura centralizzata essa non ci darà mai la sicurezza che una struttura distribuita con regole ferree come la blockchain può darci, anche a costo di dover spendere un po' di più.

Dato il problema esposto e le tecnologie presentate, in questo documento andremo a presentare il software che implementa una possibile soluzione a questo problema proprio grazie a Git e alla blockchain, **PineSU**.

Il resto della tesi sarà strutturata come segue:

- Capitolo 2 - **Concetti preliminari**: Verranno approfondite le tecnologie menzionate nell'introduzione e presentate altre che hanno permesso una corretta ed efficiente implementazione
- Capitolo 3 - **Il Problema e l'Obiettivo**: Verrà formalizzato il problema che andremo ad affrontare e spiegate alcune delle problematiche implementative e come sono state superate
- Capitolo 4 - **Il Software PineSU**: Verranno presentati l'effettiva implementazione, architettura e workflow del software realizzato
- Capitolo 5 - **Dimostrazioni d'uso per il fine preposto**: Esempio pratico del funzionamento del software tramite alcuni esempi che rappresentano situazioni simili a quelle in cui l'utente potrà trovarsi utilizzandolo
- Capitolo 6 - **Conclusioni e Sviluppi futuri**: Si riassumono gli obiettivi raggiunti, alcune criticità e potenziali sviluppi futuri.

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** h è una funzione che permette di associare, a una qualsiasi sequenza m di lunghezza arbitraria in input, una sequenza in output $h(m)$ di lunghezza costante. Questo valore restituito in output è chiamato valore di hash, stringa di hash, o anche semplicemente **hash**, mentre il valore preso in input è detto **preimmagine**. Possiamo pensare a questa funzione come una “macchina per impronte digitali”, per ogni sequenza in input essa riesce a calcolarne una stringa binaria che la identifica univocamente.

Una funzione di hash ha tre caratteristiche fondamentali: innanzitutto è *deterministica*, ciò significa che per lo stesso input essa deve generare sempre lo stesso output, deve poi generare esclusivamente *sequenze in output con una lunghezza fissa*, ciò significa che per qualsiasi input di qualsiasi lunghezza il risultato dovrà avere sempre una lunghezza di b bit decisa a priori, infine deve essere *uniforme*, ovvero i suoi output dovrebbero essere uniformemente distribuiti nel codominio della funzione. Una stringa di hash, essendo una sequenza binaria, può essere rappresentata in molti modi, nell’ambito di questo documento presenteremo i vari hash come stringhe esadecimali.

Mentre una funzione di hash classica è tranquillamente utilizzabile per situazioni in cui non è necessaria una particolare sicurezza nel proteggere le caratteristiche delle

preimmagine, ragion per cui si può decidere, nel progettare, di prestare più attenzione verso la sua rapidità d'esecuzione che altro, quando si ha bisogno che le informazioni in input rimangano nascoste e si necessita di una maggior sicurezza a scapito della velocità si ricorre alle **funzioni crittografiche di hash**.

Una funzione crittografica di hash ha le stesse caratteristiche di una funzione di hash normale ma aggiunge delle proprietà che deve seguire per poter essere considerata *crittograficamente sicura*, i valori della sua lunghezza b sono tipicamente 128, 256 e 512, si va quindi ad ottenere degli output potenzialmente molto più lunghi e che non sembrano adatti alle implementazioni all'interno di semplici strutture dati per cui le classiche funzioni di hash sono designate.

Le proprietà che permettono di definire una funzione crittografica di hash come sicura sono:

1. *Resistenza alla preimmagine*: Dato un hash h deve essere difficile riuscire a trovare un input m tale che $h = h(m)$.
2. *Resistenza alla seconda preimmagine*: Dato un input m_1 deve essere difficile riuscire a trovare un diverso input m_2 tale che $h(m_1) = h(m_2)$.
3. *Resistenza alla collisione*: Dati due messaggi m_1 ed m_2 , deve essere difficile che i due messaggi abbiano lo stesso hash, quindi con $h(m_1) = h(m_2)$.

Queste proprietà ci permettono di arrivare a concludere che una funzione crittografica di hash effettua un'operazione unidirezionale: non è possibile (o perlomeno non dovrebbe esserlo), partendo dal singolo hash, risalire alla preimmagine.

Per riuscire a mantenere tali proprietà la funzione, durante la fase di generazione dell'output, effettua diverse e differenti operazioni sulla preimmagine che fanno sì che anche un solo minuscolo cambiamento all'input generi un *effetto valanga* sull'output, cambiando radicalmente, se non completamente, l'hash generato.

Le funzioni crittografiche di hash vengono utilizzate in moltissime implementazioni nell'ambito della cybersecurity come la verifica di password, la generazione e validazione di firme digitali e la **verifica d'integrità di file**. Quest'ultima assume un'importanza fondamentale anche nel nostro caso: queste funzioni ci permettono di capire se, dati due file, il loro contenuto è identico senza la necessità di effettuare alcun controllo byte per byte in quanto produrranno lo stesso valore di hash, in questo modo possiamo anche

capire se un file, che durante un controllo generava un determinato valore, è stato modificato, ciò perché il valore generato sarà ovviamente differente.

2.2 VCS e Git

Un **Version Control System** [1] (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 [?]. 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 [2] è il sistema di controllo di versione distribuito più diffuso al mondo. Esso modella ogni repository come una *sequenza* o *flusso di snapshot* (istantanee) 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 server remoto, in modo da mantenerli sincronizzati.

2.3 Blockchain ed Ethereum

Non esiste una definizione formalizzata e universalmente condivisa di cosa è la **blockchain**, possiamo però tentare di definirla come una lista in continua crescita di record, chiamati blocchi, collegati utilizzando metodi crittografici: ogni blocco contiene l'hash corrispondente al blocco precedente, un timestamp e i dati riguardanti una transazione. La *chain*, ovvero la catena, è formata grazie alla presenza del riferimento crittografico verso il blocco precedente che si trova all'interno di ogni blocco, ciò rende tale catena di blocchi immutabile: il cambiamento dei dati all'interno di un blocco andrebbe a far mutare il suo riferimento crittografico, invalidando così la catena. La blockchain non è immagazzinata all'interno di una singola unità o insieme di unità

centrali, bensì è distribuita, condivisa e aggiornata tra varie macchine all'interno di una rete.

Abbiamo parlato di come ogni blocco contenga i dati di una transazione, essa non è altro che un'azione compiuta da parte di un account della rete (i.e. Bob) nei confronti di un altro account (i.e. Alice), un esempio classico è l'invio di una quantità di moneta virtuale, a quel punto verrà registrata come transazione che il portafoglio o *wallet* di Bob è in debito di tale quantità nei confronti del portafoglio di Alice. Possiamo quindi concludere che una transazione altro non è che la registrazione di proprietà di un asset, ovvero un elemento, digitale e non, avente valore.

La nascita della blockchain è infatti dovuta alla necessità di avere un "libro mastro" per Bitcoin, un registro che tenesse traccia delle transazioni e che impedisse il fenomeno della *doppia spesa*, ovvero ciò che si verifica quando uno stesso titolo valutario viene speso due o più volte. Mentre questo fenomeno è controllato in un'economia tradizionale dagli istituti finanziari centrali, nell'ambito delle monete digitali distribuite a prendersi in carico di effettuare questo controllo fondamentale è la blockchain con il suo meccanismo di consenso *Proof-of-Work* (meccanismo adottato nella blockchain di Bitcoin, ne esistono altri come il *Proof-of-Stake* che sta venendo adottato da Ethereum).

La *Proof-of-Work* è l'algoritmo di consenso che detta le regole per la conferma di transazioni e la produzione di nuovi blocchi della catena: per poter aggiungere un blocco è infatti necessario risolvere un complesso enigma per cui viene richiesta un'alta potenza computazionale, i *miner* o minatori mettono a disposizione le proprie macchine per poter risolvere tale enigma in cambio di una ricompensa nella moneta virtuale di tale rete. Più una catena è lunga, più lavoro computazionale è stato svolto per produrla e più gli utenti saranno orientati nel riporre la propria fiducia in essa, soprattutto perché un lavoro di manomissione sarebbe a quel punto incredibilmente difficoltoso e costoso.

Vediamo quindi come, nonostante le limitazioni e gli sprechi di risorse che un sistema distribuito come la blockchain ha per sua natura, si ha il grande vantaggio di un registro condiviso in cui possiamo fidarci della parola di numerosissime entità che ne usufruiscono anziché di un'unica entità centrale.

Mentre reti come quella Bitcoin a poco si prestano oltre al gestire l'omonima moneta virtuale, troviamo altre reti in cui si è pensato di aggiungere molte più funzionalità. Una di queste è **Ethereum**, una rete che trascende il concetto di semplice rete per criptovaluta (in questo caso **Ether** 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.

Uno **Smart Contract** è essenzialmente un programma (una collezione di codice che descrive funzioni e un insieme di dati) che risiede ad uno specifico indirizzo della blockchain (quindi all'interno di uno o più blocchi), gli utilizzatori possono comunicare con loro ed utilizzare le loro funzioni richiamando queste ultime tramite delle transazioni. Con Ethereum è quindi possibile andare a creare vere e proprie applicazioni decentralizzate (unendo Smart Contract con un'interfaccia utente), chiamate **dapp**, 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.

Grazie alla nascita delle **dapp** si può azzardare ad andare a definire una nuova tipologia di web, diverso dal Web2 che conosciamo oggi ricco di contenuti generati dagli utenti ma su piattaforme centralizzate nella mani di poche aziende. Questo nuovo web, chiamato **Web3** ha lo scopo di creare una nuova esperienza di navigazione che permetta a chiunque sia in grado di connettersi alla rete Ethereum di partecipare al servizio e dove controllo centralizzato, censura e blocco dei pagamenti sono ormai concetti superati che non appartengono al grande network decentralizzato, purtroppo tutto ciò a scapito di costi elevati, spreco di risorse e di una limitata scalabilità, non è detto però che l'evoluzione di queste tecnologie non permetta di superare questi problemi.

Un grande scoglio per gli sviluppatori di **dapp** è sicuramente la presenza delle **gas fee**, ovvero i dazi sul gas, l'unità di misura del lavoro computazionale che la EVM compie per svolgere determinate operazioni. Anche il semplice invio di denaro da un indirizzo a un altro o addirittura anche solo di un messaggio, poiché implica la creazione di un blocco nella blockchain, comporta il dovuto pagamento di una tassa. Nel caso di un messaggio registrato in una transazione, più il messaggio sarà lungo, più la tassa da pagare sarà alta. L'esistenza delle fee è praticamente obbligata per come la blockchain è strutturata: il costringere al pagamento ogni qual volta si vuol creare un nuovo blocco rende la struttura incredibilmente meno vulnerabile allo spamming di nuovi blocchi da parte di cicli infiniti malevoli o accidentali. Ciò implica che ogni volta che una **dapp** vorrà andare a registrare un dato sulla blockchain qualcuno dovrà pagare affinché ciò accada.

2.4 Node.js

Node.js è un ambiente di run-time, ovvero che supporta l'esecuzione di software nonostante non faccia parte del sistema operativo, open source e cross platform per l'esecuzione di codice in linguaggio Javascript.

Javascript è un linguaggio di programmazione ad alto livello, spesso con compilazione *Just In Time* e a multi-paradigma: supporta infatti diversi stili di programmazione tra cui la programmazione a eventi, a oggetti, funzionale e persino imperativa. Inizialmente limitato alla sola esecuzione lato client su browser, la sua crescita in popolarità, complice la sua versatilità, facilità di apprendimento e la sua sintassi molto simile a C e Java, ha portato allo sviluppo anche di ambienti esterni su cui poter eseguire codice lato server e standalone, come il sopracitato Node.js, il quale si basa infatti sull'interprete Javascript V8, sviluppato da Google per il suo Chrome e dalle prestazioni elevate.

Node.js rappresenta a tutti gli effetti non solo un motore per Javascript server-side, ma un'implementazione del paradigma *Javascript everywhere*, riuscendo ad unire tutte le componenti di una qualsiasi applicazione, web e non, sotto un unico linguaggio.

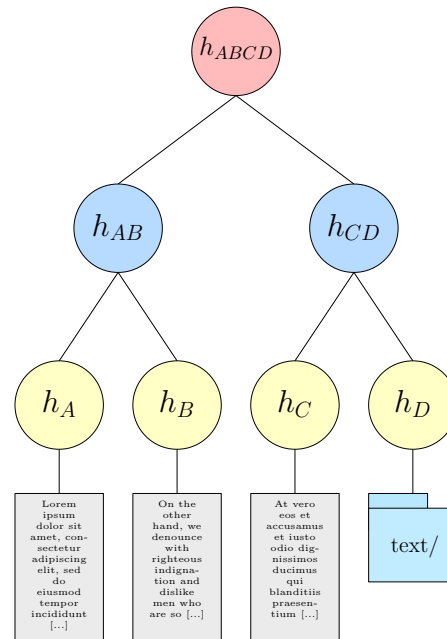
Le applicazioni Node.js vengono eseguite in un singolo processo, senza la creazione di nuovi thread per ogni richiesta, quando deve essere eseguita un'operazione di I/O, come una lettura o scrittura nella rete, un accesso a un database o a un file system, anziché bloccare il thread e sprecare cicli della CPU facendola attendere, Node.js interrompe il processo e riprende le sue funzioni non appena l'operazione I/O termina, questo permette a un server con ambiente Node.js di gestire efficientemente moltissime connessioni in concorrenza senza che il programmatore debba preoccuparsi di gestire alcuna concorrenza tra thread differenti.

Uno dei punti più affascinanti ed eccitanti di Node.js, e di molti altri linguaggi con un supporto così attivo da parte della community, è sicuramente il semplice e veloce accesso alle migliaia di librerie che gli utenti pubblicano ogni giorno e installabili tramite il package manager **npm** in maniera pressoché immediata.

2.5 Accumulatori crittografici e Merkle Tree

I **Merkle Tree** [3] 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 i cui hash sono le foglie dell'albero, 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 riusciremo ad individuare i file modificati come quelli per cui non è possibile ricostruire il cammino verso la radice lasciandola inalterata.

2.6 JSON

Il JavaScript Object Notation (**JSON**) [4] 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'alternativa ragionevole ai database per applicazioni che cercano di sviluppare architetture distribuite. Osserviamo e analizziamo un esempio di oggetto JSON:

```
1 { "nome": "Mario",
2   "cognome": "Rossi",
3   "eta": 27,
4   "indirizzo": {
5     "indirizzoStradale": "Via Fasulla 123",
6     "citta": "Perugia"
7   },
8   "numeriTelefono": [
9     {
10      "tipo": "casa",
11      "numero": "212 555-1234"
12    },
13    {
14      "tipo": "ufficio",
15      "numero": "646 555-4567"
16    } ],
17   "figli": [],
18   "coniuge": null }
```

Possiamo osservare come ogni attributo e il suo valore vengano rappresentati come

$$“key” = value$$

dove il valore è circondato da due virgolette in caso sia una stringa (riga 1) o da nulla in caso sia un tipo primitivo (riga 3). Un valore può inoltre essere anche un altro oggetto (righe 4-7), in questo caso vediamo come sia incluso tra parentesi graffe e segua poi al suo interno la sintassi che abbiamo appena osservato, o un array di valori semplici o oggetti (quest'ultimo alle righe 8-16) andando ad inserire i valori tra parentesi quadre. È possibile anche rappresentare array vuoti (riga 17) con delle parentesi quadre senza contenuto e valori nulli (riga 18) con la parola chiave null. I valori vengono separati tra loro con una semplice virgola.

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 rigoroso 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.

La possibilità di appoggiarsi ad un sistema centralizzato è a questo punto quella che offrirebbe una verifica meno sicura: non solo sarebbe necessario avere fiducia dell'entità che mantiene i propri dati, ma usando dei database centralizzati 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 permetta però di espletare controlli su un gran numero di documenti.

La soluzione al dilemma arriva sotto forma di accumulatori crittografici (sezione 2.5). 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** [5].

PineSU si presenta come un software *lightweight* 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 inglobare logicamente una repository Git, costruendo, tramite la definizione di alcuni metadati, una struttura intorno ad essa.

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 nella blockchain, inserendola all'interno gruppi di suoi simili chiamati **Storage Group** (SG). Questi 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.

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 quale a seconda del suo stato permetterà di fare alcune operazioni anziché altre.

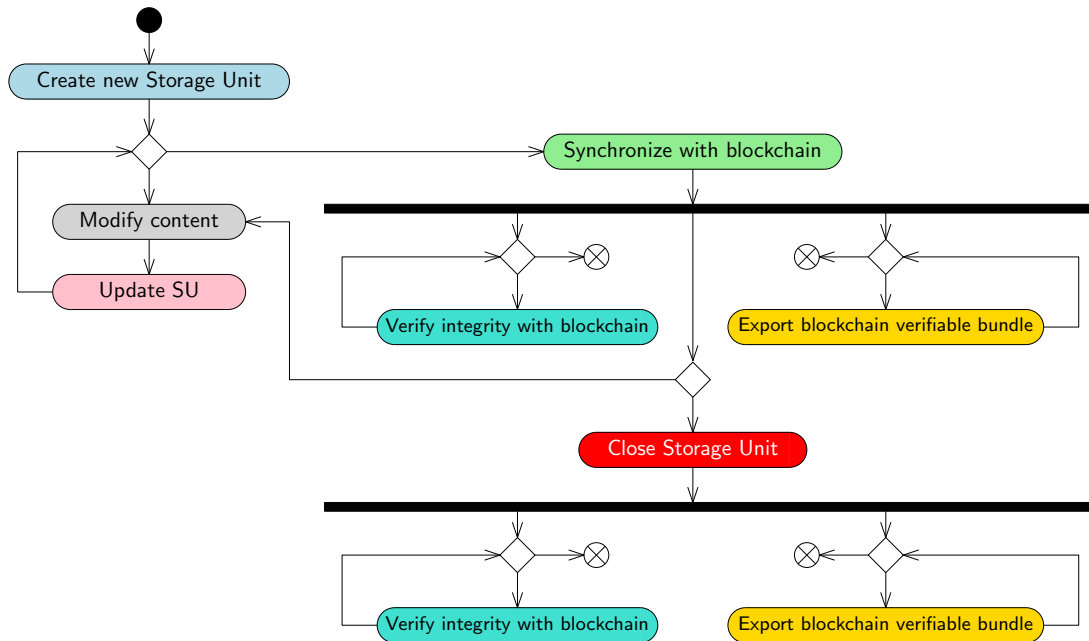


Figura 4.1: Workflow dell'applicativo sotto forma di Activity Diagram.

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 i metadati in modo da rendere già possibili registrazioni e verifiche. Dopo questa operazione potrebbero accadere i due 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 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 (*Verify integrity with blockchain*), sia

offline che online, e l'esportazione di sottoinsiemi di file dalla SU (*Export blockchain verifiable bundle*), munendoli con i dati necessari per permettere verifiche d'integrità da parte di terzi anche senza disporre dell'intera SU.

Una terza operazione da effettuare una sola volta in seguito a una registrazione è la chiusura (*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, a cui dovrebbe seguire una registrazione su blockchain, potranno essere eseguite in pratica le stesse operazioni eccetto, ovviamente, quella del ricalcolo;

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.

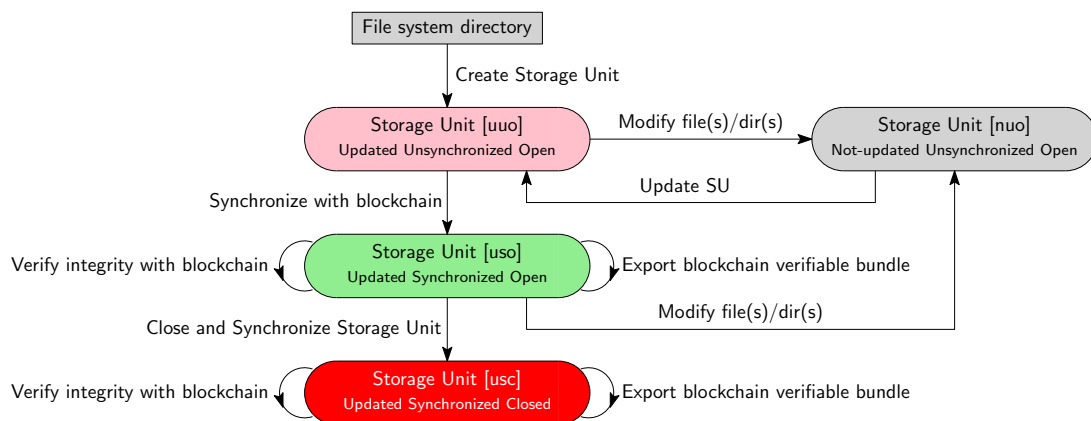


Figura 4.2: Ciclo di vita di una Storage Unit sotto forma di State Diagram.

4.2 Architettura

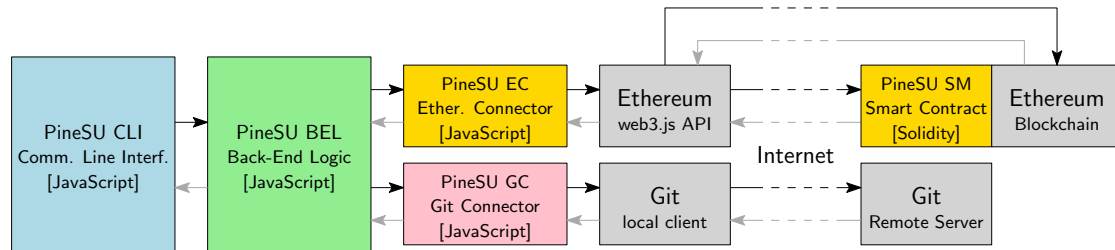


Figura 4.3: Rappresentazione dell'architettura ad alto livello di PineSU. Le frecce nere sono messaggi scatenati dalle entità sorgente corrispondenti mentre le frecce grigie sono risposte passive dell'entità interrogata.

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 Storage 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.

¹L'interfaccia si basa sul modulo npm Inquirer.js [6]

²Il componente si basa su alcuni moduli npm per operazioni di supporto [7] [8] [9]

- *PineSU EC (Ethereum Connector)* Si interfaccia con il modulo web3.js [10] [11].
- *PineSU GC (Git Connector)* Si interfaccia con il modulo simple-git [12].
- *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.

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*.

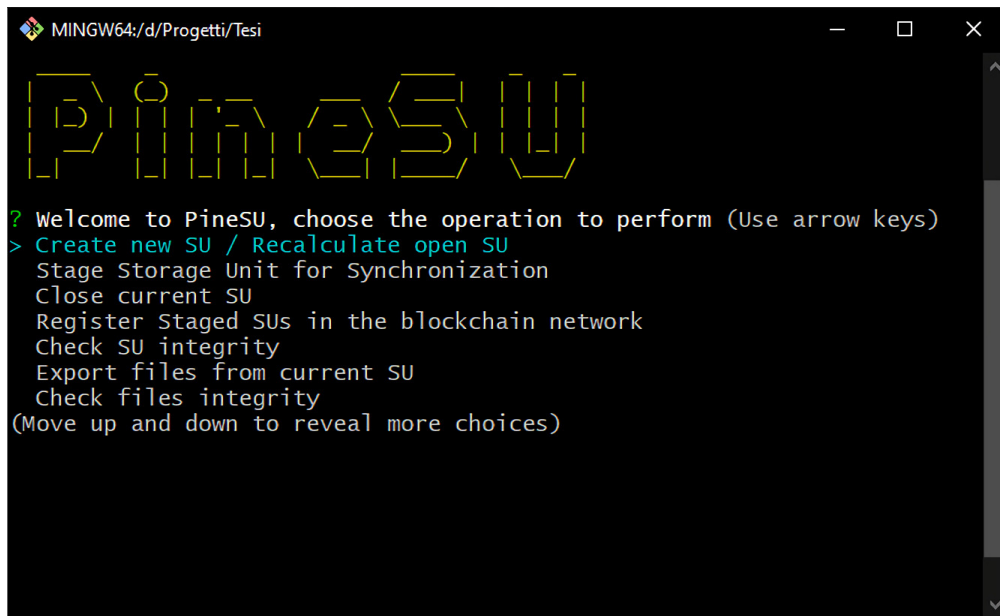


Figura 4.4: Menù principale dell'applicativo con la scelta dell'operazione da effettuare.

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 parleremo meglio nella sottosezione 4.3.4 e 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 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.

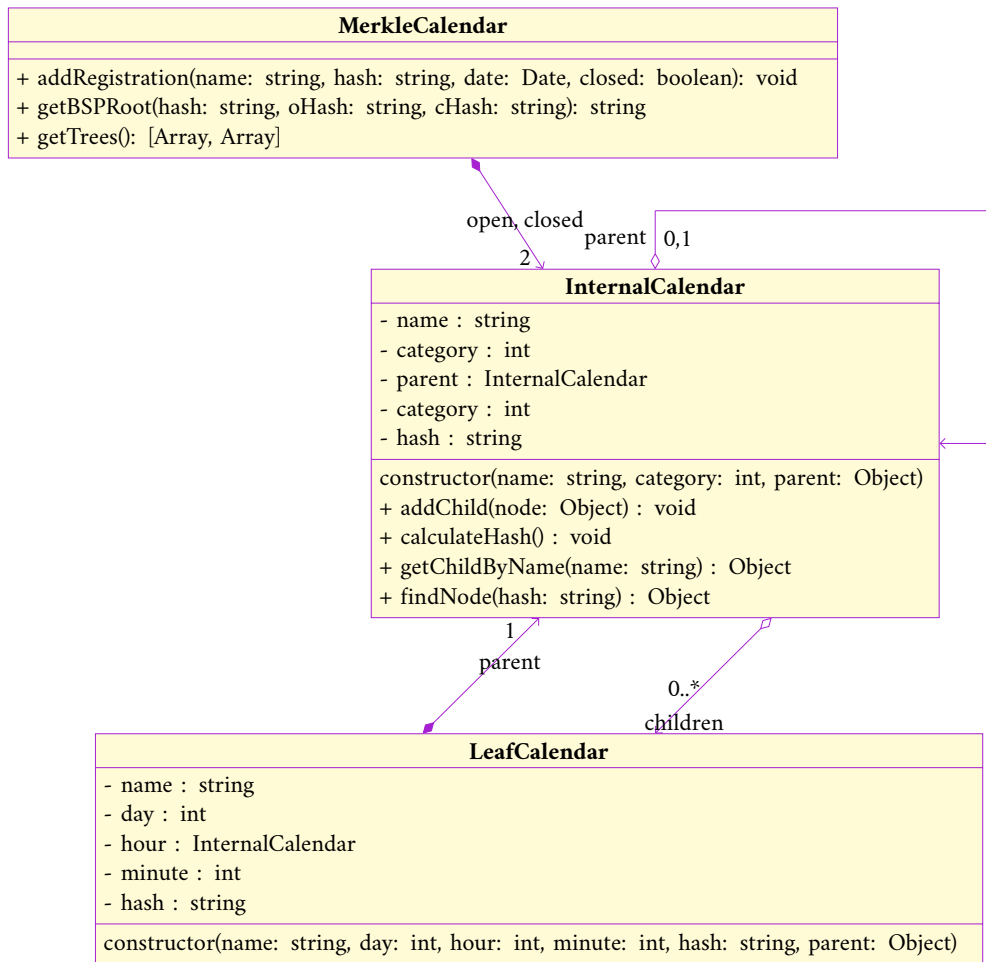


Figura 4.5: Rappresentazione UML delle classi descritte

4.3.3 Excursus sul Merkle Calendar

In Fig. 4.6 possiamo osservare un possibile 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.

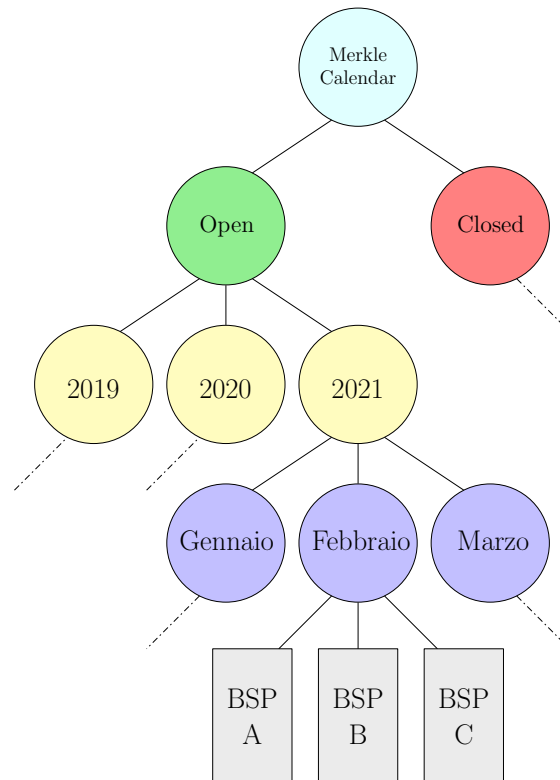


Figura 4.6: Rappresentazione grafica di un Merkle Calendar.

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).

4.3.4 Excursus sullo Storage Group

È possibile, alla vista della struttura del Merkle Calendar, aver pensato se fosse realmente necessario inserire l'ulteriore struttura dello Storage Group (Fig. 4.7), se non si potesse, anziché andare ad inserire la Merkle Root dello SG nel BSP, inserire direttamente l'hash di una singola Storage Unit, in modo da rendere ogni BSP proprio di una sola entità.

Come si può notare infatti nella schematizzazione del Workflow e dell'Architettura (sezione 4.1) non è presente una fase di Staging, in cui la Storage Unit viene inserita all'interno dello Storage Group, ciò perché effettivamente quest'ultima non è una struttura necessaria ai fini di una definizione formale dell'applicativo. Quel che è necessario però è che gli utilizzatori di questo spendano meno soldi possibile, cosa improbabile se la registrazione su blockchain avviene di ogni singola Storage Unit. Si tratta invece di una mossa saggia andare ad introdurre una piccola struttura che permetta, con una singola registrazione dal costo identico, di andare a registrare un numero indefinito di Storage Unit, quando ci riferiremo quindi all'operazione “*Synchronize with blockchain*” per questa particolare implementazioni di PineSU, comprenderemo anche uno staging a priori di tale SU.

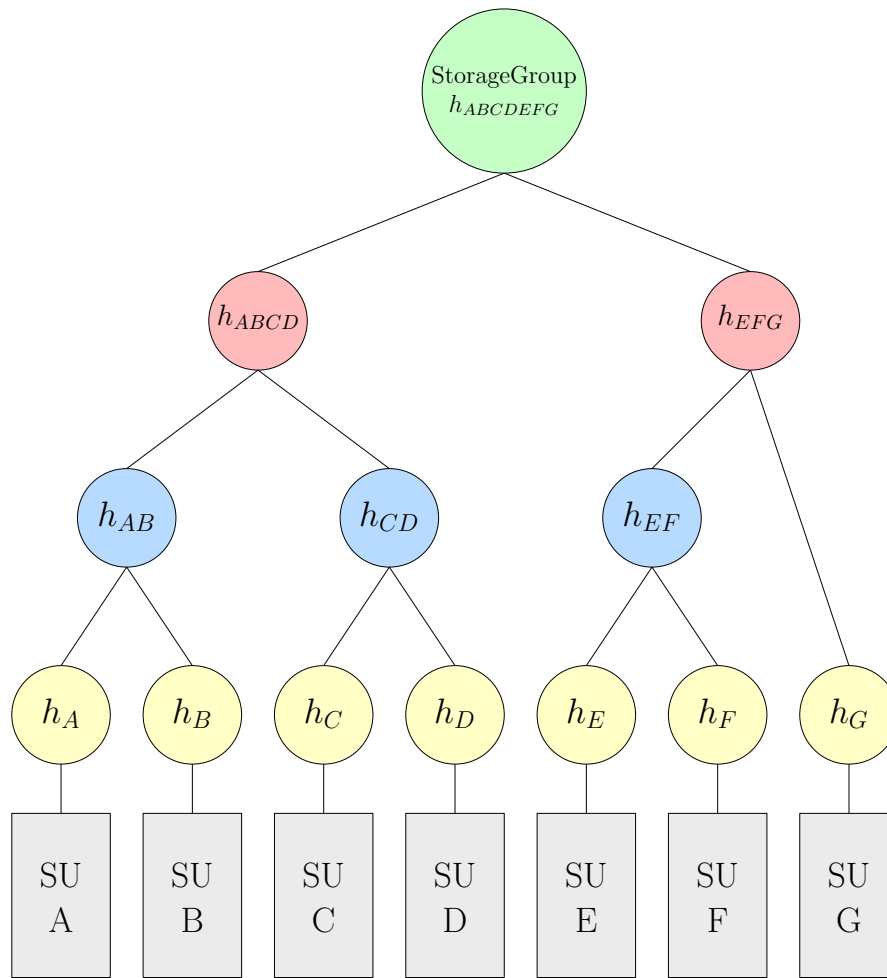


Figura 4.7: Rappresentazione grafica di uno Storage Group, essendo un semplice Merkle Tree si può usare come rappresentazione del calcolo degli hash in una Storage Unit prendendo, come foglie, i singoli file e directory.

4.3.5 PineSU EC

Come visualizzabile in Fig. 4.8 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.

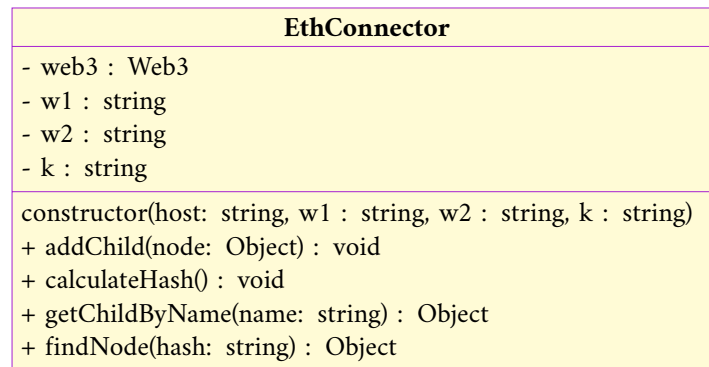


Figura 4.8: Rappresentazione UML di EthConnector

4.3.6 PineSU GC

Il connettore per Git, la cui rappresentazione UML è visualizzabile in Fig. 4.9, 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.

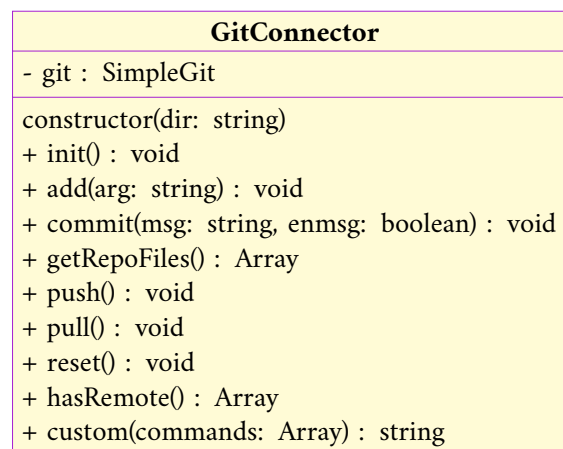


Figura 4.9: Rappresentazione UML di GitConnector

4.3.7 PineSU SM

Lo Smart Contract (??) 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 mostrando anche il codice della parte PineSU CLI associato.

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.

```
1  const create = async () => {
2    // Controllo d'esistenza di repository Git con creazione in caso
      contrario
3    if (files.fileExists('.git')) {
4      console.log(chalk.green('Already a Git repository!'));
5    } else {
6      const setup = await inquirer.gitSetup();
7      if(setup.gitinit == "Yes"){
8        gitLogic.init();
9      }else{
10       console.log('PineSU requires a git repository to work on');
11       process.exit();
12     }
13   }
14
15   // Creazione del .gitignore
16   const inqignore = await inquirer.gitAdd();
17   if(inqignore.gitignore == "Yes"){
18     await files.createGitignore();
19     await gitLogic.addFileSU('.gitignore');
20   }
21
22   // Aggiunta dei file della directory a un commit "fantoccio"
23   // e calcolo dei loro hash
24   const spinnerAdd =
25     ora('Adding files to the Storage Unit...').start();
26   await gitLogic.addAllSU();
27
28   var filelist = await gitLogic.commitSU("").then( async () => {
29     return await gitLogic.calculateSU()
30   });
31
32   if(filelist[0] == "null"){
33     gitLogic.resetCommit();
34     return;
35   }
```

```
36 // Calcolo del Merkle Tree della SU
37 var merkleroot = gitLogic.calculateTree(filelist);
38 spinnerAdd.succeed("All files added");
39
40 // Creazione del file .pinesu.json
41 // contenente i metadati della SU
42 await inquirer.askSUDetails(files.getCurrentDirectoryBase(),
43   remote).then((details) => {
44   var hash = merkleroot.toString('utf8');
45   Object.assign(details, {owner: w1});
46   Object.assign(details, {hash: hash});
47   Object.assign(details, {filelist: filelist});
48   Object.assign(details, {closed: false});
49   files.saveJSON(details);
50   console.log(chalk.green("The Storage Unit has been created!"))
51 });
52
53 // Reset del commit "fantoccio" e creazione
54 // di un commit vero e proprio includendo
55 // anche il file descrittore JSON
56 gitLogic.resetCommit();
57 await gitLogic.addAllSU();
58
59 const commit = await inquirer.gitCommit();
60 await gitLogic.commitSU(commit.message);
61 };
```

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*.

```
1  const stage = async () => {
2    // Lettura dei metadati della SU,
3    // se non e' gia' presente viene
4    // inserita nello SG
5    var pinesu = files.readPineSUFile(".pinesu.json");
6    const found = sg.some(el => el.hash == pinesu.hash);
7    if(!found){
8      sg.push({
9        name: pinesu.name,
10       hash: pinesu.hash,
11       path: files.getCurrentDirectoryABS(),
12       closed: pinesu.closed
13     });
14     files.saveSG(sg);
15   }
16 };
```

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.

```
1  const register = async () => {
2      // Vengono calcolate le MR dei due Storage Group
3      [document, openRoot, closedRoot] = files.createSGTrees(sg);
4
5      // Si aggiungono massimo due nuovi BSP al Merkle Calendar
6      if(openRoot !== null){
7          ethLogic.addToTree(openRoot, mc, false);
8      }
9      if(closedRoot !== null){
10         ethLogic.addToTree(closedRoot, mc, true);
11     }
12     if(openRoot !== null || closedRoot !== null){
13         // Si richiama il connettore per la rete Ethereum
14         // per registrare la root del Merkle Calendar
15         // nella blockchain
16         var [oHash, cHash, transactionHash] =
17             await ethLogic.registerMC(mc);
18
19         // Si creano i file di metadati ".registration.json"
20         // da inserire in ogni directory delle SU facente parti
21         // degli Storage Group appena inseriti
22         for(var el of document){
23             el.oHash = oHash;
24             el.cHash = cHash;
25             el.tranHash = transactionHash;
26             files.createRegistration(el);
27             // Si fa un Git Commit in ognuna di queste SU
28             await gitLogic.makeRegistrationCommit(el.path);
29         }
30         gitLogic.changeDir('..');
31         files.flushSG();
32         files.saveTree(mc);
33     } else {
34         console.log(chalk.red("There are no SUs staged!"));
35     }
36 };
```

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 4.3.7), per verificare che non sia già presente la entry corrispondente alla chiusura di quella SU.

```
1  const close = async () => {
2    // Chiusura "weak", vengono controllati i commit,
3    // se e' presente un commit di chiusura viene
4    // annullata l'operazione, altrimenti si chiude la SU
5    let res = await gitLogic.checkCommitMessages();
6    if(res){
7      var pinesu = files.closePineSUFile('.pinesu.json');
8      if(pinesu == null){
9        console.log(chalk.red("Failure"))
10     } else {
11       console.log(chalk.green("Success"));
12       await gitLogic.addFileSU(".pinesu.json");
13       await gitLogic.commitSU("The Storage Unit
14                               is now closed");
15     }
16   } else {
17     console.log(chalk.red("This Storage Unit has
18                           already been closed in the past"));
19   }
20 };
```

L'implementazione *strong* appare identica eccetto per la chiamata di un metodo differente alla riga 5.

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.

```
1  const distribute = async () => {
2    // Si fanno scegliere all'utente i file
3    // da inserire nell'archivio
4    var filelist = await files.distributeSU();
5
6    if(filelist[0] != "null"){
7      // Si crea il file ZIP con i file
8      // e un descrittore ".pifiles.json"
9      var filesJSON = gitLogic.createFilesJSON(filelist);
10     files.createZIP(filelist, filesJSON);
11     console.log(chalk.green("ZIP file successfully created"))
12   } else {
13     console.log(chalk.red("An error occurred"))
14   }
15 };
```

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.

```
1  const checkFilesBlockchain = async () => {
2    // Si controlla la presenza nella directory
3    // di un file chiamato ".pifiles.json"
4    var pifile = files.readPifile();
```

```
5     if(pifile[0] == "null"){
6         console.log(chalk.cyan('No ".pifiles.json" found'));
7         return;
8     }
9     // Per ogni file elencato in ".pifiles.json"
10    // si effettuano controlli locali
11    for(var el of pifile){
12        var hash = gitLogic.fileHashSync(el.path)
13        if(gitLogic.validateProof(el.proof, hash, el.root)){
14            console.log(chalk.green("Success"));
15            var res = files.checkRegistration(el.root)
16            if(res[0]){
17                // Se la verifica locale e' andata a buon fine
18                // si esegue una verifica su blockchain
19                console.log(chalk.green("Success"));
20                if(await ethLogic.verifyHash(mc, res[1].root,
21                    res[1].oHash, res[1].cHash, res[1].tranHash)){
22                    console.log(chalk.green("Success"));
23                } else {
24                    console.log(chalk.red("Success"));
25                }
26            } else {
27                console.log(chalk.red("Failure"));
28            }
29        } else {
30            console.log(chalk.red("Failure"));
31        }
32    }
33    };
```

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.


```
1  const check = async () => {
2      // Si calcolano gli hash della SU in esame
3      const spinnerCalc = ora('Calculating the SU hash...').start();
4      await gitLogic.calculateSU().then( async (filelist) => {
5          // Si leggono i suoi metadati e si calcola
6          // la MR della SU attuale
7          var merkleroot = gitLogic.calculateTree(filelist);
8          var pinesu = files.readPineSUFile(".pinesu.json");
9          spinnerCalc.succeed("Calculation complete!");
10         // Si controlla l'integrita' di file descritti
11         // da un eventuale ".pifiles.json"
12         checkFiles();
13         // Si controlla che la MR appena calcolata corrisponda
14         // con quella precedentemente registrata
15         if(pinesu.hash == merkleroot){
16             // Si verifica che lo stato della SU combaci
17             // con l'ultimo stato registrato in blockchain
18             // (da quanto dicono i metadati)
19             var res = files.checkRegistration(merkleroot)
20             if(res[0]){
21                 console.log(chalk.green("Success"));
22                 // Si procede all'effettivo controllo su blockchain
23                 if(await ethLogic.verifyHash(mc, res[1].root,
24                     res[1].oHash, res[1].cHash, res[1].tranHash)){
25                     console.log(chalk.green("Success"));
26                 } else {
27                     console.log(chalk.red("Failure"));
28                 }
29             } else {
30                 console.log(chalk.red("Failure"));
31             }
32             } else {
33                 console.log(chalk.red("Failure"));
34             }
35         });
36     };
```

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

- `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/lmmagine.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.



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.

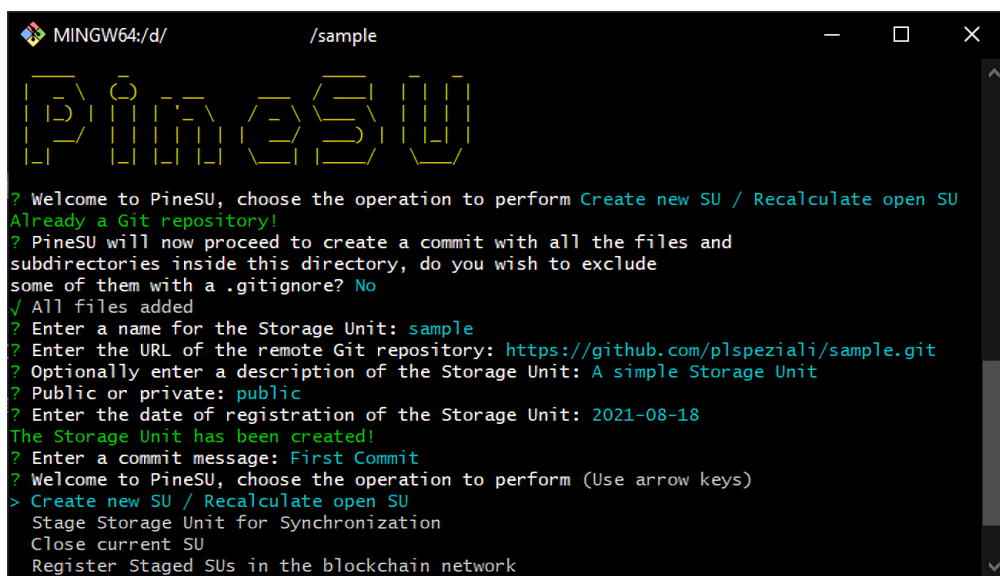


Figura 5.2: L'interfaccia di PineSU appena terminata la fase di creazione.

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/third:2
23     ec86afc483f3685893831dfe04b66620be690d2",
24     "second:d6e51cdbe8a84cb3ba2c9cfdc2773a96b2401a59"
25 ],
26 "closed": false }
```

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).

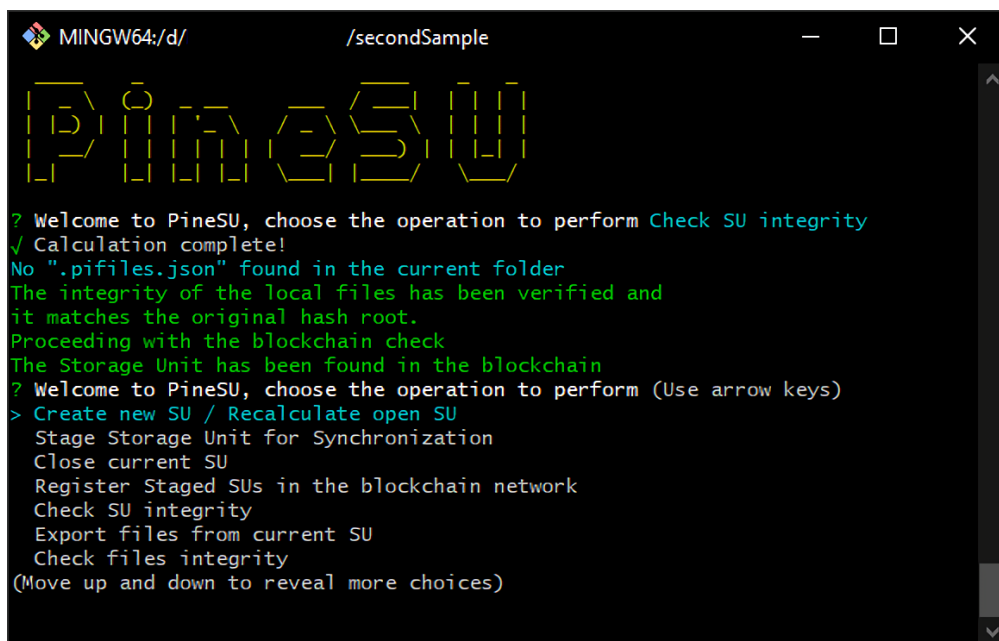
Ovviamente di conseguenza la nuova Merkle Root del Merkle Calendar verrà registrata su blockchain Ethereum inserendola in un messaggio scambiato tra i due wallet forniti dall'utente, ciò non è visualizzabile dall'utente a meno che non vada nelle directory

delle SU registrate in cui potrà trovare un nuovo file `.registration.json`, esso conterrà 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 effettuare 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

PineSU

? 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

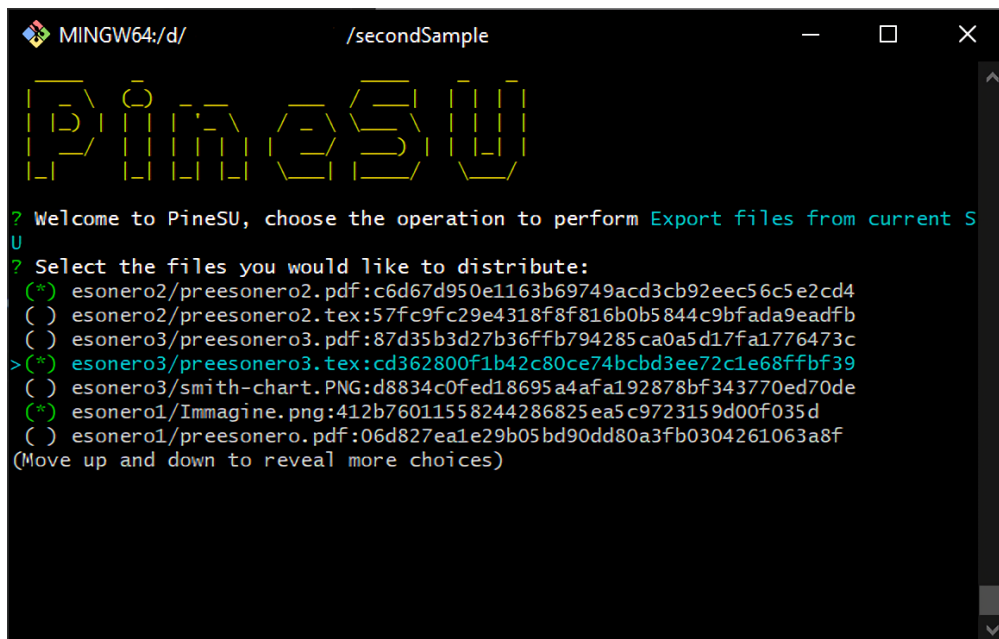


Figura 5.4: L'interfaccia di PineSU durante un'esportazione.

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 questa 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.



```

MINGW64:/d/pinesuExport

PineSU

? 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, grazie all'implementazione del software descritto, sia possibile andare a soddisfare tutti gli obiettivi che ci eravamo preposti: l'applicativo riesce infatti a garantire un'esperienza user-friendly andando ad accompagnare l'utente nella creazione, gestione, registrazione permanente e verifica di insiemi di documenti, il tutto sfruttando in maniera efficace ed efficiente Git e la blockchain di Ethereum, due tecnologie che, data la comune natura distribuita e le loro funzionalità, sono perfette per questo ruolo.

PineSU può infatti vantare, grazie all'implementazione di strutture dati salvate su descrittori JSON e metadati e grazie al collegamento remoto esclusivo per la blockchain e per eventuali repository Git remote scelte dall'utente, una struttura totalmente decentralizzata e resistente ad attacchi informatici e manomissioni di dati, finché continuerà ad esistere la rete nella cui blockchain si sono registrate le Storage Unit, quest'ultime potranno essere verificate. 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 implementativi 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 imprese.

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 indentificandoli 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 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, andando ad integrare il salvataggio e la verifica sulle blockchain pubbliche più gettonate.

7. Bibliografia

- [1] Wikipedia, “Controllo versione,” Accessed: August, 2021. [https://it.wikipedia.org/wiki/Controllo_versione].
- [2] Git-scm.com, “Git,” Accessed: August, 2021. [<https://git-scm.com/>].
- [3] H. Qureshi, “Merkle Trees,” 2019. [<https://nakamoto.com/merkle-trees/>].
- [4] JSON.org, “JSON.org,” Accessed: August, 2021. [<https://www.json.org/json-it.html>].
- [5] P. Speziali, “GitHub - plspeziali/PineSU : PineSU,” Accessed: August, 2021. [<https://github.com/plspeziali/PineSU>].
- [6] S. Boudrias, “GitHub - SBoudrias/Inquirer.js,” Accessed: August, 2021. [<https://github.com/SBoudrias/Inquirer.js/>].
- [7] The Brain, “GitHub - cthackers / adm-zip : ADM-ZIP for NodeJS,” Accessed: August, 2021. [<https://github.com/cthackers/adm-zip>].
- [8] Chalk, “GitHub - chalk / chalk : Chalk,” Accessed: August, 2021. [<https://github.com/chalk/chalk>].
- [9] Tierion, “GitHub - Tierion / merkle-tools : merkle-tools,” Accessed: August, 2021. [<https://github.com/Tierion/merkle-tools>].
- [10] ChainSafe Systems, “GitHub - ChainSafe/web3.js: Ethereum JavaScript API,” Accessed: August, 2021. [<https://github.com/ChainSafe/web3.js>].

- [11] ChainSafe Systems, “web3.js - Ethereum JavaScript API,” Accessed: August, 2021. [<https://web3js.readthedocs.io/en/v1.4.0/>].
- [12] Steve King, “GitHub - steveukx / git-js: Simple Git,” Accessed: August, 2021. [<https://github.com/steveukx/git-js>].