

# Progettazione di una Blockchain

Progetto di SCRS

Bussani Alessandro

Bruno Ivan

Cianetti Marco

Cristini Giordano

Ferrantelli Federico

Ronca Damiano

Simolo Christian

Terribili Lorenzo

Vinciguerra Simone

# Indice

<b>1</b>	<b>Introduzione al problema</b>	<b>3</b>
1.1	Titolare dei diritti . . . . .	3
1.2	Acquisto dei diritti . . . . .	3
1.3	Diritti esclusivi dell'autore . . . . .	4
1.4	La nostra idea . . . . .	4
1.5	Descrizione ad alto livello . . . . .	5
<b>2</b>	<b>Analisi dei requisiti e dell'infrastruttura</b>	<b>7</b>
2.1	Richiami . . . . .	7
2.2	Tecnologie necessarie . . . . .	8
2.3	Modelli ed entità coinvolte . . . . .	10
2.3.1	Diagramma ER Fisico . . . . .	12
2.4	Metodi sincroni ed asincroni . . . . .	12
2.5	Specifiche tecniche . . . . .	13
2.5.1	Specifiche Entry Point . . . . .	13
2.5.2	Specifiche Pool Dispatcher . . . . .	15
2.5.3	Specifiche della applicazione Java e lato client . . . . .	17
2.6	Diagrammi e flussi . . . . .	19
2.6.1	Sequence diagram generali . . . . .	20
2.6.2	Inizializzazione del miner . . . . .	21
2.6.3	manageMine() . . . . .	22
2.6.4	updateMiningService() . . . . .	23
2.6.5	mine() . . . . .	24
2.6.6	updateFilechain() . . . . .	26
2.6.7	verifyBlock() . . . . .	28
<b>3</b>	<b>Implementazione dell'architettura</b>	<b>32</b>
3.1	Strutture dati . . . . .	32
3.1.1	Transaction . . . . .	32
3.1.2	Block . . . . .	32
3.1.3	User . . . . .	32
3.1.4	Citation . . . . .	33
3.1.5	Login . . . . .	33
3.2	Selezione delle possibili transazioni citabili . . . . .	33
3.3	Protocollo di comunicazione . . . . .	37
3.4	Dettagli implementativi . . . . .	38
3.4.1	I package . . . . .	40
3.4.2	Le classi . . . . .	41
3.5	Interfaccia grafica . . . . .	46
<b>4</b>	<b>Criticità e conclusioni</b>	<b>56</b>
4.1	Possibili sviluppi . . . . .	57
4.2	Conclusioni . . . . .	57

# 1 Introduzione al problema

Il problema che andiamo ad affrontare è quello della tutela dei diritti su file, siano essi di testo, multimediali, o di differente natura. Poiché spesso si confonde il diritto d'autore con il copyright, le cui differenze variano di paese in paese, abbiamo deciso di riferirci al diritto d'autore così come viene definito in Italia.

Una persona che crea un'opera *originale* rappresentata da un supporto fisico (necessario, in quanto non si può applicare il diritto d'autore sulle idee) ne possiede automaticamente il *diritto d'autore*. Questa proprietà conferisce al titolare il diritto esclusivo di utilizzo dell'opera in alcuni modi specifici.

Alcuni esempi di opere soggette al diritto d'autore sono:

- Opere audiovisive
- Opere audio
- Opere scritte
- Opere visive
- Software

## 1.1 Titolare dei diritti

Il titolare dei diritti su un'opera è solitamente il suo creatore, che ne sia autore o coautore, ma, nel caso di un'opera realizzata in adempimento di un contratto di lavoro subordinato o su commissione, l'autore diviene titolare dei soli diritti morali dell'opera, mentre i diritti patrimoniali spettano al datore di lavoro.

In quest'ultimo caso, infatti, il diritto morale di essere riconosciuto autore dell'opera è inalienabile, irrinunciabile ed imprescrittibile.

I diritti di utilizzazione economica sono invece trasferibili, ed in virtù dell'art. 25 l.a., durano per tutta la vita dell'autore e per settanta anni dopo la sua morte.

## 1.2 Acquisto dei diritti

Il diritto d'autore si acquisisce originariamente per il solo fatto della creazione dell'opera senza che sia necessario alcun tipo di adempimento amministrativo, sia esso il deposito o la sola registrazione come avviene invece in materia di brevetti e marchi.

Tuttavia, depositare un'opera presso gli uffici competenti presenta l'indubbio vantaggio di fornire all'autore prova certa della paternità e della data di

creazione di un determinato lavoro. Inoltre il deposito alla SIAE è indispensabile in certi casi per potere esercitare i diritti connessi.

### 1.3 Diritti esclusivi dell'autore

Un autore acquista sulla propria opera il diritto esclusivo di riproduzione, esecuzione, diffusione, distribuzione, noleggio, prestito, elaborazione e trasformazione, che può eventualmente cedere ad altri, in tutto o solo in parte, facendosi ricompensare per questo.

In Italia si possono effettuare diversi tipi di deposito dell'opera, a seconda della natura della stessa.

### 1.4 La nostra idea

Quello che intendiamo realizzare è un nuovo modo per esercitare i diritti d'autore su file di qualsiasi tipo.

Il nostro sistema sarà innanzitutto gratuito; l'iscrizione alla SIAE, infatti, ha ovviamente un costo, che varia dai 200 € agli oltre 2000 €.

Utilizzando il nostro sistema, sarà possibile registrare un proprio prodotto in modo gratuito all'interno di un registro distribuito, e saranno gli utenti a stabilire se tale prodotto è originale, o se al contrario già è stato registrato da un altro utente precedentemente; in altre parole, si elimina così la necessità di un'autorità che garantisca per tutti e di cui ci si debba fidare.

Un utente che utilizza un file può, se registrato nel nostro sistema, verificarne l'**integrità** e evitare così di incorrere in file manomessi; per questo motivo abbiamo deciso di utilizzare la Blockchain per implementare il tutto e beneficiare delle conseguenti sicurezze.

Tali considerazioni ci hanno portato all'implementazione del protocollo di Blockchain in quanto coincidente con la nostra idea. Ciò che proponiamo di fare, dunque, è implementare tale registro garantendo la possibilità di espanderne i suoi metodi liberamente in futuro, grazie ad una sua organizzazione precisa, efficace e modulare.

Dovremo dunque affrontare varie questioni come:

- *Similarità tra i file*: il sistema deve negare l'opportunità a un utente di pubblicare un prodotto che è già stato registrato. Ovviamente, deve anche evitare che un utente tenti di pubblicare a suo nome una (inappropriata) versione modificata di un file già registrato. Nel nostro caso, abbiamo

deciso di trascurare questa parte non essendo una competenza della Blockchain e poiché può essere facilmente implementato in un successivo step, tramite uno studio mirato.

- *Framework*: abbiamo scelto di utilizzare alcuni framework invece di scrivere tutto il codice da capo; l'applicazione avrebbe potuto essere più leggera ma questo ci permette di avere una versatilità ed un vantaggio su una implementazione più pulita e leggibile.
- Una possibile estensione futura per il progetto è trovare un metodo per registrare tutte le chiavi degli utenti, in modo da recuperarle tramite gli altri utenti. Tale metodo dovrebbe essere distribuito e definito in modo da non esserci entità che detengono le chiavi pubbliche.

## 1.5 Descrizione ad alto livello

Prima di spiegare cosa sia una Blockchain, descriviamo brevemente come verrà sfruttata nel nostro problema, mantenendo un alto livello di astrazione.

Per comprendere la struttura del progetto, elenchiamo alcune delle principali entità del nostro progetto:

- **TRANSAZIONI** le quali si compongono dell'hash del file e del suo autore; una transazione si verifica ogni volta che il proprietario di un file decide di trasferirlo al nostro database, affinché venga certificato dal sistema
- **BLOCCHI** composti a loro volta da insiemi di transazioni, in attesa di validazione o già validati
- **FILECHAIN** o *catena di blocchi*, ossia una lista che raccoglie i blocchi di transazioni validate dal sistema; questa struttura ci permette di avere una pubblica immagine delle certificazioni effettuate in ogni momento.

### Cos'è la Blockchain?

Una Blockchain è un registro distribuito, che mantiene in modo continuo una lista crescente di dati, i quali faranno riferimento a dati precedenti presenti nella lista stessa.

La Blockchain è composta da blocchi, i quali contengono informazioni relative a transazioni valide recenti. Ciascun blocco contiene un *timestamp* e include l'hash del blocco precedente, collegando i blocchi insieme.

I blocchi così collegati formano una catena, con ogni nuovo blocco che si unisce alla catena come se fosse un suo anello prolungando la catena preesistente. Ad ogni nuova transazione la catena si allunga, rendendone la manomissione praticamente impossibile, attraverso l'utilizzo di protocolli comuni nella crittografia come ad esempio il sistema di firme o l'utilizzo di hash.

In una rete che utilizza questo meccanismo, ciascun nodo ha quanto meno una copia parziale di una Blockchain memorizzata localmente. Ciò significa che la Blockchain non risiede in un singolo server, ma in una rete distribuita dove ciascuno ne verifica il contenuto, non fidandosi degli altri.

Ciò permette di dare *consenso* nel caso in cui si ritiene valido il nuovo blocco, prolungando così la propria catena; questo consenso genera fiducia nella rete, poiché gli altri possono a questo punto verificare a loro volta la correttezza della convalidazione eseguita ed aggiornare così la catena, senza la necessità di passare per una terza entità di fiducia, ma saldando di persona un anello dopo l'altro nel caso in cui si abbiano anelli "validi".

## 2 Analisi dei requisiti e dell'infrastruttura

Descriveremo sinteticamente le infrastrutture che sono coinvolte nella creazione della rete su cui far scorrere il nostro protocollo della blockchain modificata.

### 2.1 Richiami

Richiamiamo le seguenti nozioni, che saranno utili alla comprensione dell'analisi effettuata in questo capitolo:

- un *merkle tree* è un albero tale che ogni nodo non foglia viene etichettato con un hash, il cui valore viene calcolato a partire dalle etichette dei nodi figli; la sua radice viene detta *merkle root*. Questo particolare nodo permette verifiche sicure ed efficienti di grandi gruppi di strutture dati. Queste informazioni servono per verificare in un qualsiasi momento la veridicità della lista delle transazioni contenute nel blocco.
- il sistema di *proof-of-work* si basa sulla teoria dei giochi, dove sono presenti agenti egoistici in competizione su di una rete. Si compone di due fasi:

PRIMA FASE viene calcolata la soluzione di un problema di carattere esponenziale, che si ottiene dall'insieme degli attributi del blocco legandoli. Ciò implica che i miner entrano in competizione per rispondere per primi a questo "puzzle" per aggiudicarsi il *reward*. Un miner viene premiato solo se il suo blocco viene verificato ed utilizzato dagli altri miner come ultimo anello per congiungere il blocco che dovranno calcolare.

SECONDA FASE un miner deve poter verificare facilmente la soluzione del "puzzle" fornita in un tempo relativamente breve e decidere quale tra i blocchi verificati utilizzare per proseguire la catena, solitamente scelgono la catena più lunga per poter avere così maggiori possibilità di farsi verificare.

La difficoltà di tale "puzzle" deve essere regolata in modo da poter gestire la frequenza di generazione dei nuovi blocchi. Ciò favorisce la scelta di dire la verità da parte dei miner, in modo da entrare in competizione per aggiudicarsi il premio e mantenere saldo il *consenso* nella rete.

Nella figura 1 (vedi pagina seguente) è possibile vedere come i blocchi vengano collegati tra loro e collegati a partire dall'hash del precedente, in modo da risultare così immodificabili: il minimo cambiamento di bit al suo interno causerebbe la modifica a cascata di tutti gli hash nella catena. Si può quindi affermare che ogni valore di hash contiene le informazioni di tutti gli hash precedenti relativi alla transazione.

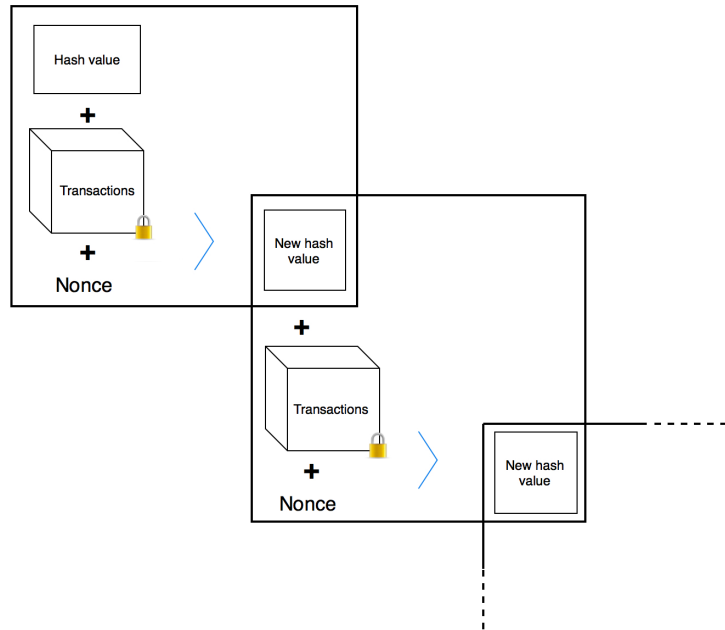


Figura 1: Struttura generale del sistema.

## 2.2 Tecnologie necessarie

Tra le strutture necessarie coinvolte nel progetto troviamo:

- *Rete peer-to-peer (P2P)* grazie alla quale possiamo sfruttare al meglio l'essenza del protocollo della blockchain, permettendo così a tutti i client di rimanere sullo stesso livello ed ottenendo così una vera e propria rete distribuita
- *VPN* vogliamo utilizzare una rete privata sulla quale far girare il nostro traffico in maniera protetta tramite il meccanismo di cifratura ed autenticazione/validazione *certificato INUIT Tor Vergata*; in questo modo dovremmo rendere più difficile la possibilità di attacchi di diversa natura, tra i quali man-in-the-middle, syn-flooding, sniffing, ecc. Inoltre, questo ci permette di risolvere problemi di indirizzamento, evitando così indirizzi pubblici fissi per i client
- *Entry Point* server tramite il quale possiamo accedere alla rete P2P in modo da conoscere gli altri client e farsi inserire nella lista dei client attivi. Anche al suo interno, verrà implementato il meccanismo di validazione/autenticazione *certificato INUIT Tor Vergata*
- *Pool Dispatcher* server che raccoglierà le richieste (o *transazioni*) inviate dai client connessi alla rete; questa entità si occuperà quindi di fare una



prima scrematura (eliminando le eventuali transazioni inconsistenti) e successivamente si occuperà di smistarle nella P2P ai vari miner, in modo da garantirne il corretto flusso di funzionamento

- *Applicazione web:*

- *Sito (front-end):* permette di creare un utente, effettuare l'upload dei file nella Filechain in modo da poter essere validati e successivamente certificati e visualizzare statistiche ed informazioni relativi alla rete, il tutto attraverso un'interfaccia accattivante ed user-friendly
- *Server (back-end):* permette di mantenere la persistenza dei dati e della Filechain e gestisce i dati stessi; inoltre, esegue tutte le operazioni che vengono richieste all'utente attraverso il server di interfaccia front-end, come ad esempio l'avvio dell'algoritmo che si occupa di minare un blocco

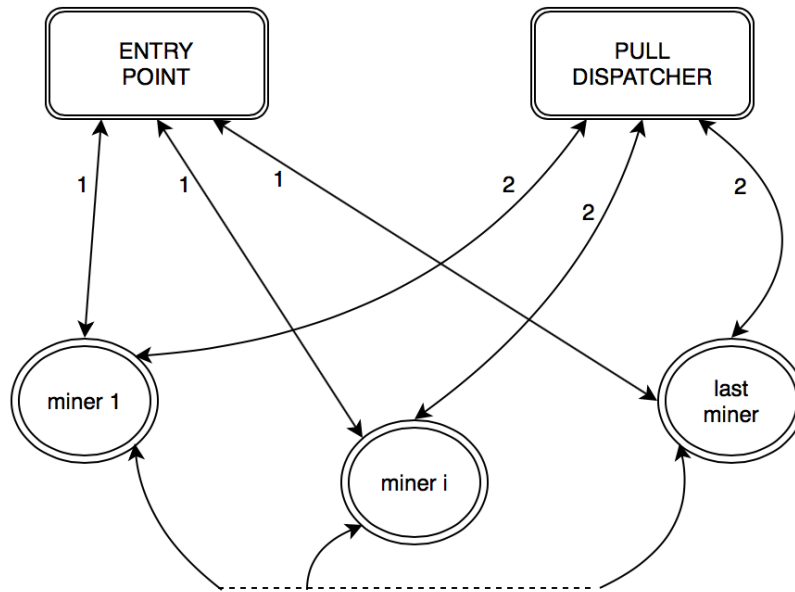


Figura 2: Struttura generale del sistema.

Dalla figura 2 è possibile vedere come interagiscono tra loro le varie entità che entrano in gioco; più in dettaglio, gli archi etichettati con 1 individuano le richieste di connessione alla rete da parte dei miner, mentre gli archi con etichetta 2 individuano le richieste di transazioni da parte dei miner al sistema. Il PD manterrà aggiornati tutti i miner e ne garantirà l'interoperabilità. Infine, come è possibile vedere nella parte tratteggiata in basso, i miner sono tra loro

connessi, e possono tutti comunicare direttamente tra loro risultando, nel totale, un grafo completo.

## 2.3 Modelli ed entità coinvolte

I modelli che vengono utilizzati per le varie richieste in formato JSON, ed i loro relativi attributi, sono:

- **Transazione**, intesa come il singolo invio specifico di un file a nome dell'utente proprietario; l'operazione di verifica delle transazioni garantisce l'integrità della catena e permette così di certificare i file stessi. Si compone dei seguenti attributi:
  - *Hash del file*: identifica il file univocamente permettendo così di evitare lo storage del file stesso per intero all'interno della blockchain, alleggerendone così la dimensione; viene calcolato tramite SHA256 del file inviato
  - *Nome del file*: scelto dall'utente proprietario del file
  - *Public key user*: relativa all'utente proprietario
  - *Lista "cita"*: di transazioni citate nel file, contiene le coppie che permettono l'identificazione delle citazioni, ossia la chiave hash sia di chi cita sia di chi viene citato, permettendo così il collegamento logico tra le transazioni stesse, qualora una transazione necessiti di alcuni riferimenti relativi a transazioni precedentemente eseguite sulla Filechain
- **Utente**, ovvero il proprietario del file che intende richiederne una certificazione di autenticità. Si parlerà di *user* qualora s'intenda l'utente proprietario che invia il file al sistema, e di *miner* se s'intende invece l'utente impegnato nella validazione dei blocchi. La sua chiave privata viene impiegata esclusivamente nella firma digitale dei blocchi e non è quindi presente tra i suoi attributi. Si compone dei seguenti attributi:
  - *Public key user*: contiene la chiave pubblica dell'utente
  - *Hash public key*: contiene la chiave pubblica dell'utente calcolata tramite SHA256  
NB: la chiave privata dell'utente non verrà mai inviata nella rete; dunque, nel modello del database dell'EP, non sarà presente.
  - *Nome*: contiene il nome dell'utente
  - *Cognome*: contiene il cognome dell'utente
  - *Username*: contiene l'username scelto dall'utente
  - *E-mail*: contiene l'indirizzo e-mail dell'utente

- **Blocco**, inteso come un quantitativo di transazioni raccolte per la loro validazione. Convalidare un blocco significa verificarne le transazioni contenute (come visto alla voce Transazioni) ed aggiungerlo in coda alla catena. Nella fase di testing eseguita, ogni blocco è stato formato da 2 transazioni, ma è possibile aumentarne la quantità. Si compone dei seguenti attributi:

- *Proof-of-work*: è l'hash del blocco identificato univocamente
- *Merkle-root*: ovvero la lista delle transazioni codificata attraverso i *merkle-tree*
- *Public key miner*: contiene la chiave pubblica dell'utente
- *Nonce*: ovvero un valore intero inizialmente randomico e poi incrementato ad ogni round
- *Chain level*: è il livello di profondità della catena nel quale giace il blocco considerato (vedi disegno sotto)
- *Lista di transazioni*: è una lista di quelle transazioni che sono validate nel blocco
- *Tempo di creazione*: valore che identifica la data attuale della creazione del blocco (facendo risultare così ogni blocco univoco)
- *Hash previous block*: è il valore hash del blocco precedente
- *Firma digitale*: ottenuta tramite chiave privata dell'utente

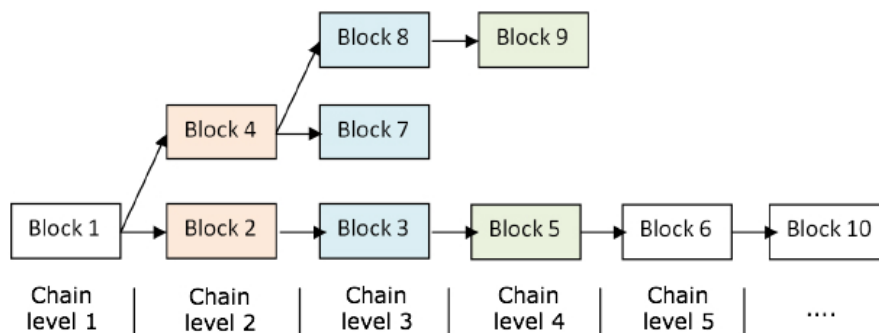


Figura 3: Chain level nella blockchain in caso di *fork* della catena.

- **Login**, tramite il quale è possibile gestire l'autenticazione degli user (che possono così diventare miner) assegnando a ciascuno di essi un ruolo. Si compone dei seguenti attributi:
- *Public key user*: contiene la chiave pubblica dell'utente
  - *Hash public key*: contiene la chiave pubblica dell'utente calcolata tramite SHA256

- *Role*: identifica il ruolo dello user
- *Username*: relativo allo user
- *Password*: relativa allo user

### 2.3.1 Diagramma ER Fisico

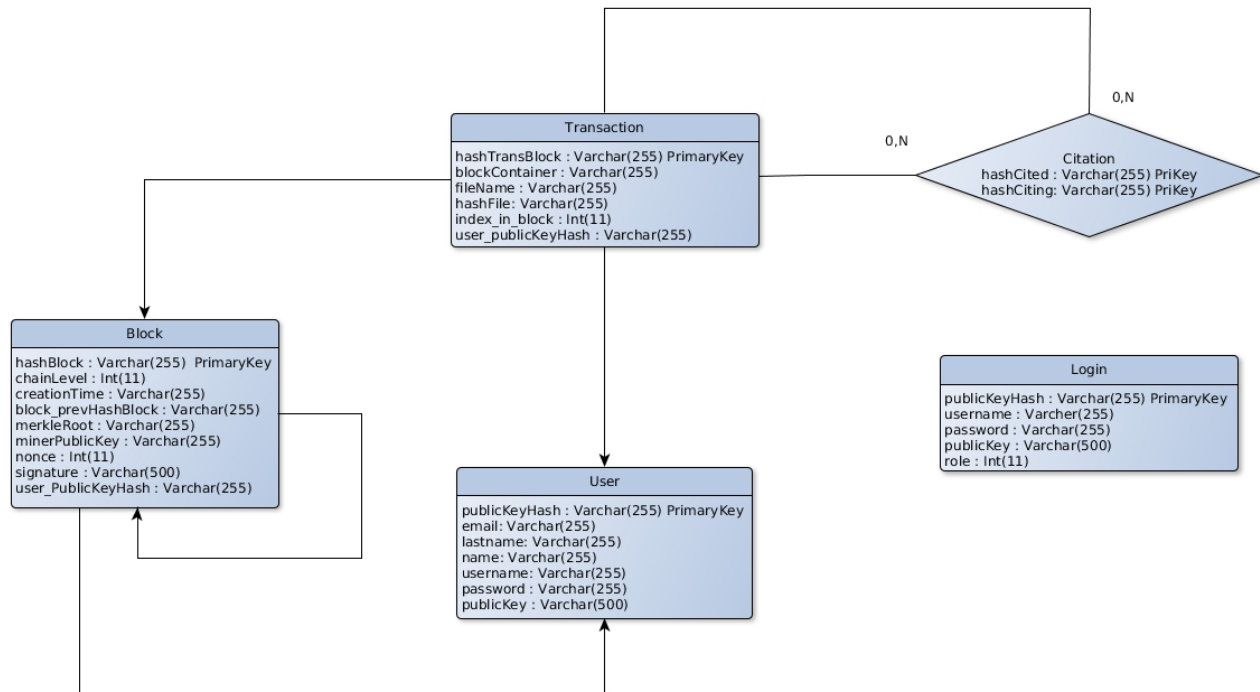


Figura 4: Diagramma Entity-Relationship della Filechain.

## 2.4 Metodi sincroni ed asincroni

Nel corso del progetto si parlerà di metodi sincroni ed asincroni: la differenza tra i due riguarda strettamente il ciclo di vita del miner. I metodi sincroni sono bloccanti e reagiscono agli eventi, mentre i metodi asincroni creano un thread che rimane in ascolto degli eventi.

Relativamente all'utilizzo che ne si fa nel progetto, la fase di sincronizzazione riguarda strettamente la comunicazione con l'Entry Point nel caso in cui si voglia ottenere una lista dei miner presenti nella rete o tra miner stessi nel caso in cui si vogliano ottenere da uno di essi alcuni blocchi necessari all'aggiornamento della catena del miner che ne fa richiesta.

## 2.5 Specifiche tecniche

Si osservi generalmente che tutti i parametri delle richieste effettuate sono in formato JSON, esclusi i casi dove specificato diversamente. Se viene chiesto un parametro chiamato *nome*, allora sarà inviato un JSON così composto:

```
1 {  
2   "nome": "scrs_2016"  
3 }
```

Nelle richieste dove è necessaria una risposta di ACK o simili allora la risposta attesa, nel campo dati del pacchetto HTTP, sarà un JSON così composto:

```
1 {  
2   "response": "ok"  
3 }
```

### 2.5.1 Specifiche Entry Point

L'Entry Point (EP) è l'entità che gestisce la connessione degli utenti alla rete, permettendo il login/logout e la gestione degli IP degli utenti connessi. Dovrà inoltre tenere traccia degli utenti registrati, per poter permettere ai miner di verificare l'identità degli altri miner.

Di seguito vengono elencate le richieste dell'Entry Point.

#### Connessione di un utente

**Richiesta:** POST user\_connect

**Parametri:** user\_ip: Stringa rappresentante l'IP e la porta dell'utente che si sta connettendo. (es. "user\_ip": "192.168.0.1:8080")

Alla ricezione di tale richiesta, l'EP dovrà effettuare le seguenti operazioni:

- Rispondere all'utente con la lista degli IP degli utenti connessi alla rete.
- Aggiornare la propria lista di IP di utenti connessi.
- Inviare una richiesta a tutti gli utenti connessi, contenente l'IP e la porta del nuovo utente. In questo modo, gli altri utenti potranno aggiornare la propria lista di utenti.

La richiesta da inviare ai miner corrisponde esattamente a quella appena ricevuta, ovvero:

POST user\_connect (user\_ip)

## Disconnessione di un utente

**Richiesta:** `POST user_disconnect`

**Parametri:** `user_ip`: Stringa rappresentante l'IP e la porta dell'utente che si sta disconnettendo. (es. `"user_ip": "192.168.0.1:8080"`)

Alla ricezione di tale richiesta, l'EP dovrà effettuare le seguenti operazioni:

- Rispondere all'utente con un messaggio di ACK.
- Aggiornare la propria lista di IP di utenti connessi.
- Inviare una richiesta a tutti gli utenti connessi, contenente l'IP e la porta dell'utente disconnesso. In questo modo, gli altri utenti potranno aggiornare la propria lista di utenti.

La richiesta da inviare ai miner corrisponde esattamente a quella ricevuta, ovvero:

```
POST user_disconnect (user_ip)
```

La disconnessione di un utente può avvenire anche dopo un lungo periodo di inattività, ovvero dopo mancati arrivi di richieste di `user_keep_alive`, descritta in seguito.

In tal caso, l'EP seguirà le stesse operazioni appena descritte per la richiesta di `user_disconnect`.

## Registrazione di un utente

**Richiesta:** `POST user_register`

**Parametri:** `user`: Oggetto User, che rappresenta l'utente appena registratosi.

Alla ricezione di tale richiesta, l'EP dovrà effettuare le seguenti operazioni:

- Inserire nel proprio database il nuovo utente.

## Verifica di un utente

**Richiesta:** `POST user_verify`

**Parametri:** `user_public_key`: Stringa rappresentante lo SHA256 della chiave pubblica dell'utente.

Alla ricezione di tale richiesta, l'EP dovrà effettuare le seguenti operazioni:

- Controllare nel proprio database se esiste un utente con una chiave pubblica il cui SHA256 corrisponda a quello passato come parametro.

- Restituire l'utente in formato JSON, qualora fosse trovato, o un JSON così composto:

```

1 {
2     "response": "user_not_valid"
3 }
```

### Keep alive di un utente

**Richiesta:** POST `user_keep_alive`

**Parametri:** `user_ip`: Stringa rappresentante l'IP e la porta dell'utente.

Alla ricezione di tale richiesta, l'EP dovrà effettuare le seguenti operazioni:

- Aggiornare il timestamp relativo all'IP e alla porta passati come parametro.
- Rispondere con un ACK.

### 2.5.2 Specifiche Pool Dispatcher

Il Pool Dispatcher (PD) avrà il compito di ricevere nuovi file da convalidare e manterrà una Block Chain. I file da convalidare verranno mantenuti in una coda (FIFO) e verranno inviati ai miner, qualora richiesti.

Il PD dovrà immagazzinare anche una lista di complessità che sono state utilizzate dai miner. Sarà una semplice lista di coppie del tipo (*complexity, timestamp*), dove la complessità è un intero e il timestamp è il tempo, espresso in millisecondi, in cui è divenuta attiva quella complessità.

Di seguito vengono elencate le richieste del Pool Dispatcher.

#### Richiesta di complessità

**Richiesta:** POST `get_complexity`

**Parametri:** `date`: Stringa rappresentante il timestamp a cui deve fare riferimento la complessità.

Alla ricezione di tale richiesta, il PD eseguirà le seguenti operazioni:

- Controllare quale complessità sia stata valida al tempo del timestamp passato come parametro. Dovrà essere dunque l'ultima complessità con timestamp minore o uguale a quello passato come parametro.
- Restituire la complessità sotto forma di JSON:

```

1 {
2     "complexity": 4
3 }

```

Qualora non fosse trovata alcuna complessità relativa a quel timestamp, verrà restituito -1 come valore del campo *complexity*.

È prevista una seconda implementazione sincrona, chiamata `get_blockcomplexity`, in grado di fornire la complessità al tempo della creazione di un nuovo blocco.

### Invio di una transazione

**Richiesta:** POST `sendtransaction`

**Parametri:** `transaction`: Oggetto di tipo `Transaction` rappresentante la transazione da convalidare.

Alla ricezione di tale richiesta, il PD eseguirà le seguenti operazioni:

- Aggiungere la transazione ricevuta nella coda delle transazioni da validare.
- Inviare un messaggio di ACK all'utente.

### Richiesta transazioni

**Richiesta:** GET `get_transactions`

**Parametri:** void

Alla ricezione di tale richiesta, il PD eseguirà le seguenti operazioni:

- Invierà all'utente un numero N (10 ad esempio) di transazioni in attesa di validazione.

Il PD però dovrà avere anche implementata la gestione della Block Chain. I miner al momento del calcolo di un nuovo blocco lo invieranno in broadcast alla rete, PD compreso.

Dunque, l'indirizzo del PD dovrebbe essere compreso in quelli nella lista dell'EP, in modo da trattarlo come un normale utente della rete per quanto riguarda la gestione della Block Chain.

In linea di massima, il PD sarà in ascolto su una seconda porta (dedicata ai miner), a cui arrivano messaggi contenenti i nuovi blocchi appena calcolati.



All'arrivo di questi messaggi, il PD deve controllare la validità del blocco e aggiungerlo alla propria Filechain.

Ricordiamo che il PD mantiene tutta la Filechain, ma ad ogni nuovo blocco ricevuto, il PD controllerà la propria catena fino ad un'ora prima dell'ora corrente e, poiché i blocchi in quella porzione della catena sono considerati stabili, potrà eliminare dalla coda delle transazioni da validare tutte le transazioni contenute nei blocchi della catena stabile.

### 2.5.3 Specifiche della applicazione Java e lato client

Verrà descritta a seguire la struttura dell'applicazione Java, cercando di mostrare ciò che vede l'utente e descrivendone le azioni in background.

Per quanto riguarda il codice, all'interno della cartella `STATIC` sono presenti due ulteriori cartelle che distinguono le due fasi in cui può trovarsi il sistema:

- `DEVELOPMENT` è la fase di sviluppo, nella quale il sistema viene testato e migliorato. Questa è la fase che è stata implementata effettivamente
- `PRODUCTION`, invece, è la fase successiva di produzione, alla quale si accede dopo aver effettuato la minificazione dei file, ossia dopo aver compresso i file al fine di aumentare le prestazioni del sistema, in termini di tempi di caricamento delle pagine web

Nella cartella `SRC/MAIN/RESOURCES` giace tutta la parte grafica relativa all'applicazione.

Il sistema ha una struttura dinamica, in quanto ha la facoltà di cambiare dinamicamente il proprio comportamento con la semplice abilitazione di una proprietà, presente nel file **`ui.properties`** all'interno della cartella `SRC/MAIN/RESOURCES/CONFIGURATIONS`.

Tale file è fondamentale, in quanto contiene la proprietà **`ui.environment`**, la quale appunto setta l'ambiente di lavoro del sistema. A seguire, nel file sono presenti le risorse necessarie al corretto funzionamento di ciascun ambiente, tra le quali si segnalano **`ui.development.prefix`** e **`ui.development.suffix`**.

Il lato client dell'applicazione è configurato nel file **`MvcConfig.java`**, all'interno della cartella `SRC/MAIN/JAVA/CS/SCRS/CONFIG`. In particolare, vengono impostati i percorsi per le risorse statiche ed il componente che si occupa del reperimento delle viste HTML.

A seconda dell'ambiente utilizzato, specificato nel file **`ui.properties`**, vengono aggiunte le risorse relative, presenti anche loro nello stesso file. Inoltre,

vengono importati i *prefissi* ed i *suffissi* dei file da **ui.properties**, ossia rispettivamente il percorso e l'estensione dei file.

Tutte le classi di configurazione estendono la classe astratta **AUiConfig.java** ed implementano l'interfaccia **IUiConfig.java**, entrambe presenti all'interno della cartella SRC/MAIN/JAVA/CS/SCRS/CONFIG/UI. Di conseguenza, le classi di configurazione possiedono tutte i metodi **getPrefix()**, **getSuffix()**, **getResources()** e **getConfigFromEnvironment()**.

Questo è il fulcro del lato client del sistema ed è il motivo per il quale è possibile cambiare il comportamento dell'intero sistema impostando una sola variabile, poiché con questi metodi verranno di volta in volta importate le risorse relative all'ambiente attuale.

Segue ora una breve descrizione delle classi principali dell'applicazione, contenute all'interno della cartella SRC/MAIN/RESOURCES/STATIC/DEVELOPMENT/APP/SCRIPTS.

La classe **app.js** si occupa di avviare e configurare l'applicazione.

All'interno della cartella CONTROLLERS, troviamo:

- **citation.dialog.js** contiene gli script necessari a selezionare e deselectare la citazione di altre transazioni all'interno della finestra di dialogo
- **miner.dialog.js** apre una finestra di dialogo per permettere all'utente di selezionare l'IP con cui avviare il mining
- **miner.js** contiene le funzioni necessarie ad avviare il mining, come la scelta dell'IP da parte dell'utente e la configurazione dei bottoni dell'interfaccia grafica durante il mining
- **wallet.transactions.js** mostra le transazioni presenti all'interno del wallet dell'utente
- **wallet.transactions.post.js** si occupa dell'invio di una nuova transazione da parte dell'utente
- **welcome.signin.js** è una classe non implementata, poiché la funzione di registrazione non è presente in questa versione del progetto
- **welcome.signup.js** si occupa del login di un utente

La classe DIRECTIVES/UPLOADDIRECTIVE.JS è l'unica presente in questa cartella, e si occupa di caricare un file da locale, bypassando quindi il relativo campo HTML.

All'interno della cartella FILTERS, troviamo i seguenti filtri:

- **TransactionsFilter.js** filtra le transazioni per hash delle transazioni citate
- **ExactMatch/ExactMatchFilter.js** filtra le transazioni per nome
- **fabState/fabStateFilter.js** filtra le transazioni a seconda del loro stato

All'interno della cartella `MODULES` sono presenti le classi che implementano i vari moduli dell'applicazione, nello specifico troviamo:

- **fil3chain.js** elenca le dipendenze di un modulo appena creato
- **navbar.fil3chain** è una cartella che include diverse classi, tutte relative alla barra di navigazione
- **sha256.fil3chain** contiene due classi, le quali si occupano della cifratura dei file
- **sidenav.fil3chain** è una cartella che include diverse classi, tutte relative alla barra di navigazione laterale
- **speedDial.fil3chain** si occupa dello speed dial, ossia del bottone in basso a destra
- **user.fil3chain** contiene alcune classi, relative alla creazione, aggiornamento, eliminazione e validazione degli user

All'interno della cartella `SERVICES`, troviamo:

- **MinerService.js** include le funzioni utilizzate per il mining, come l'avvio e l'arresto del mining
- **TransactionService.js** include i servizi utilizzati per l'invio e la ricezione di transazioni

Nel prossimo capitolo, sarà possibile vedere un'anteprima dell'interfaccia grafica con un esempio di navigazione nella applicazione web.

## 2.6 Diagrammi e flussi

Saranno mostrati a seguire diagrammi e flussi di esecuzione relativi al funzionamento base del sistema. Queste illustrazioni vogliono semplificare la comprensione della logica che risiede dietro all'implementazione della *filechain*.

Per fare ciò, vengono forniti i sequence diagram del flusso principale d'esecuzione e dei metodi più rilevanti più volte richiamati all'interno dei diagrammi

stessi.

Con precisione, una prima sequenza di diagrammi riguarderà le operazioni "elementari" del sistema, che mettono in relazione le entità tra loro. Successivamente, tali metodi verranno richiamati nei sequence diagram che mostrano l'intero ciclo di vita dell'applicazione, dal momento in cui la si avvia fino all'istante nel quale si decide di fermare il task di mining.

Al fine di chiarire la logica dei diagrammi, viene specificato fin da subito che:

- **Ref** è un riferimento ai diagrammi "elementari" precedenti i flussi
- La **GUI**, interfaccia grafica del sistema, fa riferimento ai file contenuti in `RESOURCES/STATIC` dove risiedono le pagine HTML mostrate nello strato applicativo dello stack ISO/OSI; il coinvolgimento di Spring in ciò permette sia l'accesso da *development* che da *production* a seconda di quali privilegi ha l'utente
- La dicitura **Thread** indica che i metodi vengono richiamati in maniera *asincrona* ed agiscono in background in attesa di eventi esterni, come ad esempio l'arrivo di nuovi blocchi

E, inoltre, si ricordi che:

- Il metodo **manageMine()** è utile alla gestione delle operazioni di mining permesse all'utente
- Il metodo **updateMiningService()** serve alla creazione di un nuovo blocco
- Il metodo **mine()** esegue l'operazione di mining vera e propria, ovvero la *proof-of-work* valida
- Il metodo **updateFilechain()** aggiorna la blockchain fino al blocco corrispondente, confrontando i chain level e inviando richieste a tutti i miner presenti sulla rete; il miner contattato viene rimosso dalla lista IP nel caso in cui non restituisca alcun blocco, altrimenti inizia la ricezione del blocco (o della quantità di blocchi) mancante che deve essere seguita da un task di verifica. La prima chiamata alla update presente avviene immediatamente dopo l'ingresso del miner nella rete P2P

### 2.6.1 Sequence diagram generali

Segue una breve illustrazione dei flussi di esecuzione che fanno riferimento ad azioni elementari incontrate più volte nei successivi sequence diagram. Per richiamare una di queste azioni, verrà usata nel diagramma una **Ref**.

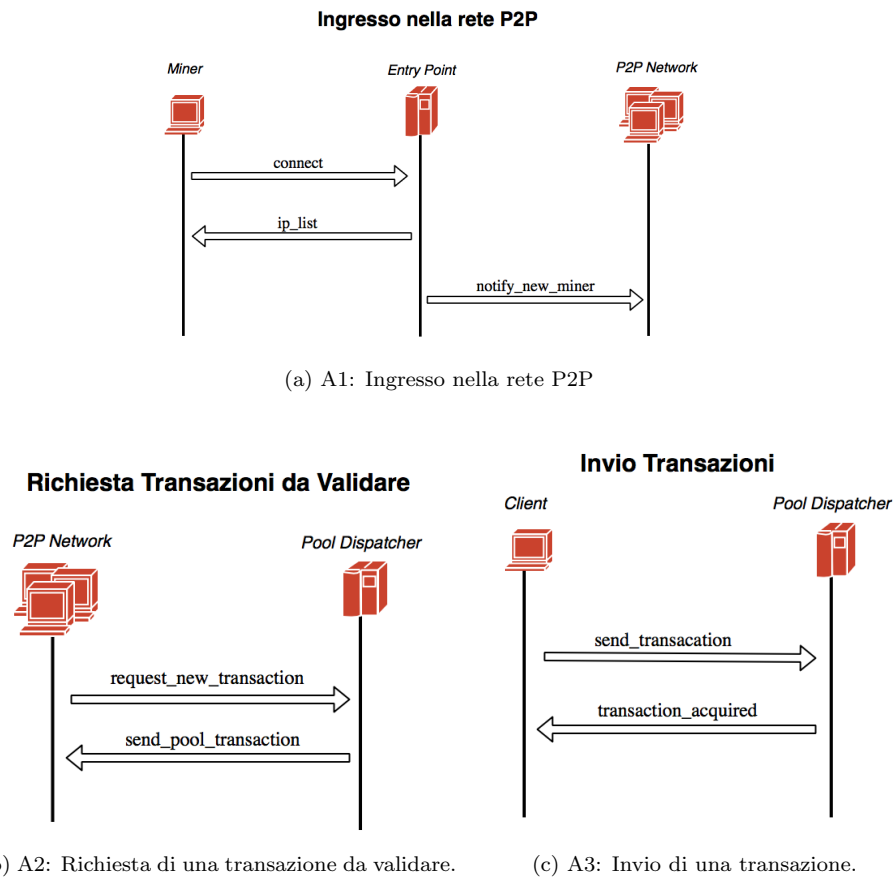


Figura 5: Sequence diagram generali (1)

## 2.6.2 Inizializzazione del miner

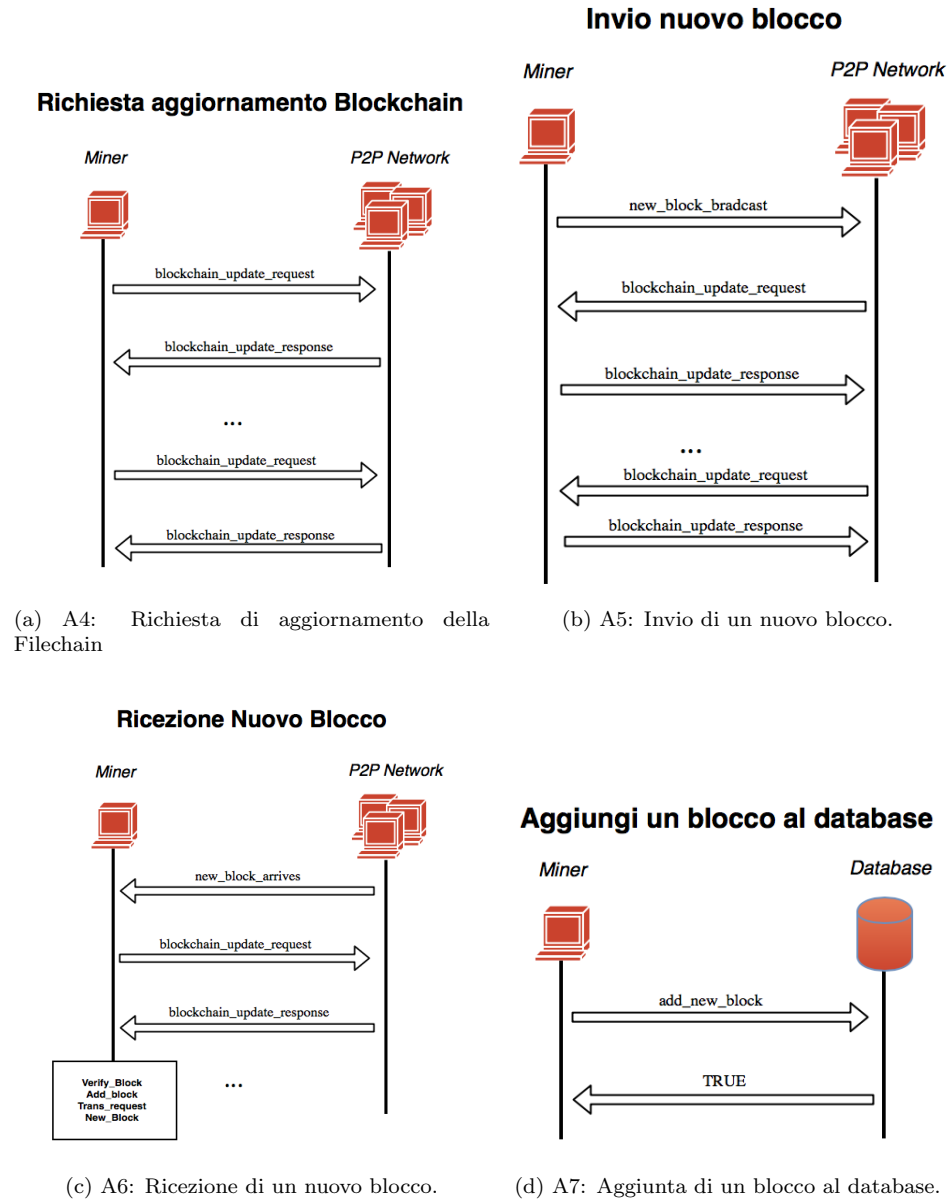


Figura 6: Sequence diagram generali (2).

### 2.6.3 manageMine()

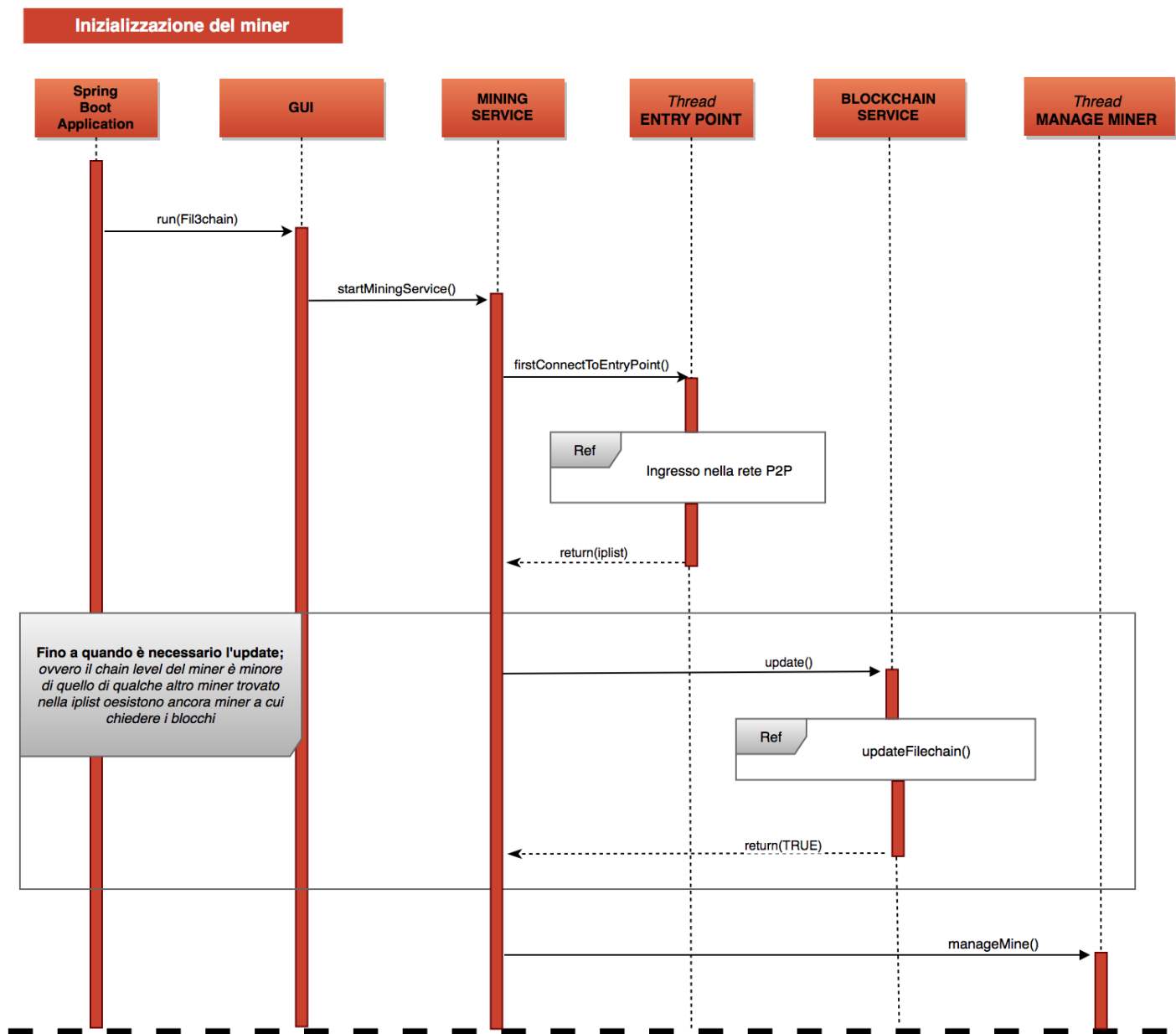


Figura 7: Flusso di bootstrap del miner.

#### 2.6.4 updateMiningService()

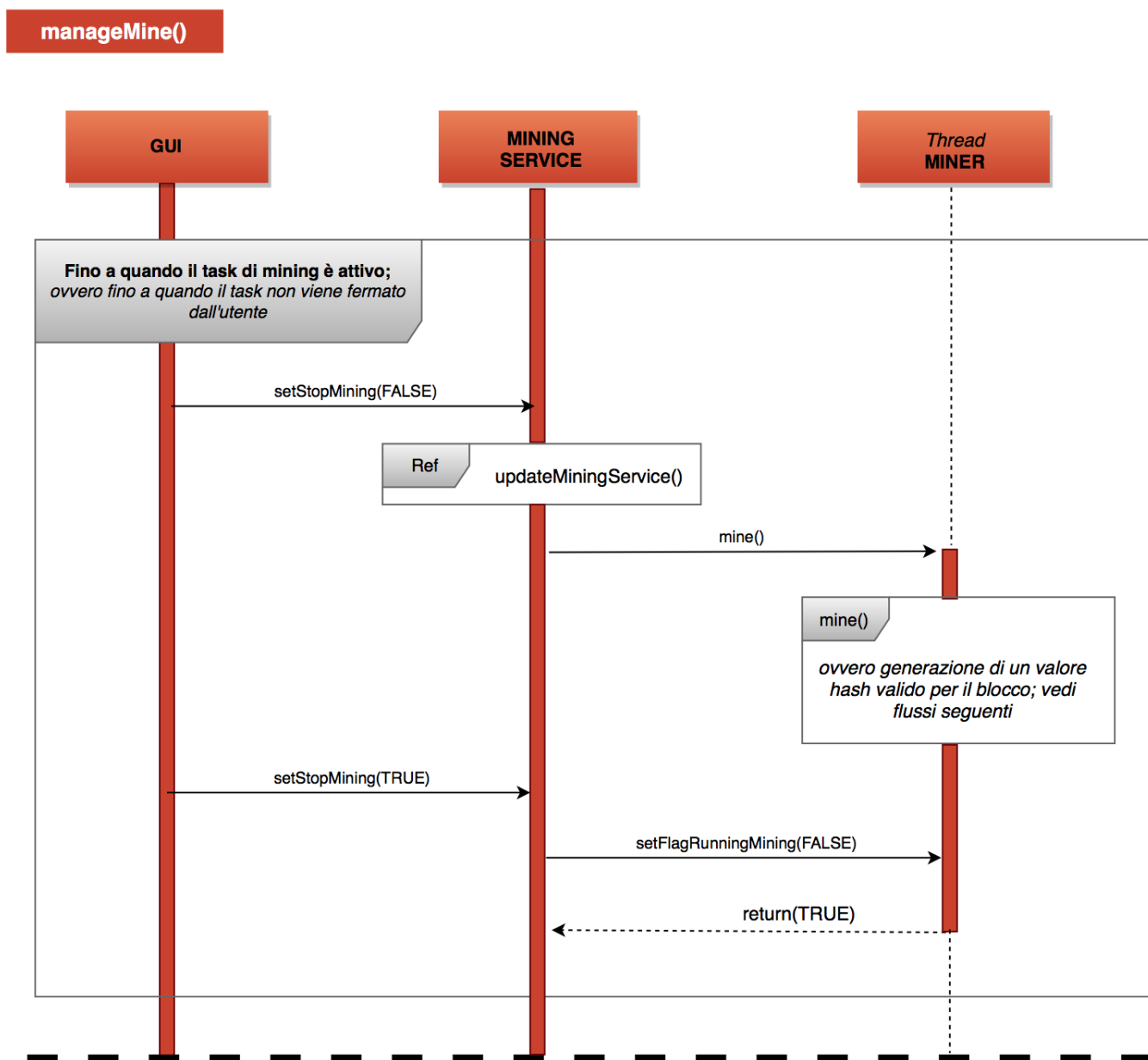


Figura 8: Flusso del metodo `manageMine()`.

### 2.6.5 `mine()`

Il metodo `mine()` viene chiamato in maniera asincrona immediatamente dopo l'esecuzione di `manageMine()`; ne mostriamo di seguito i due possibili flussi (a



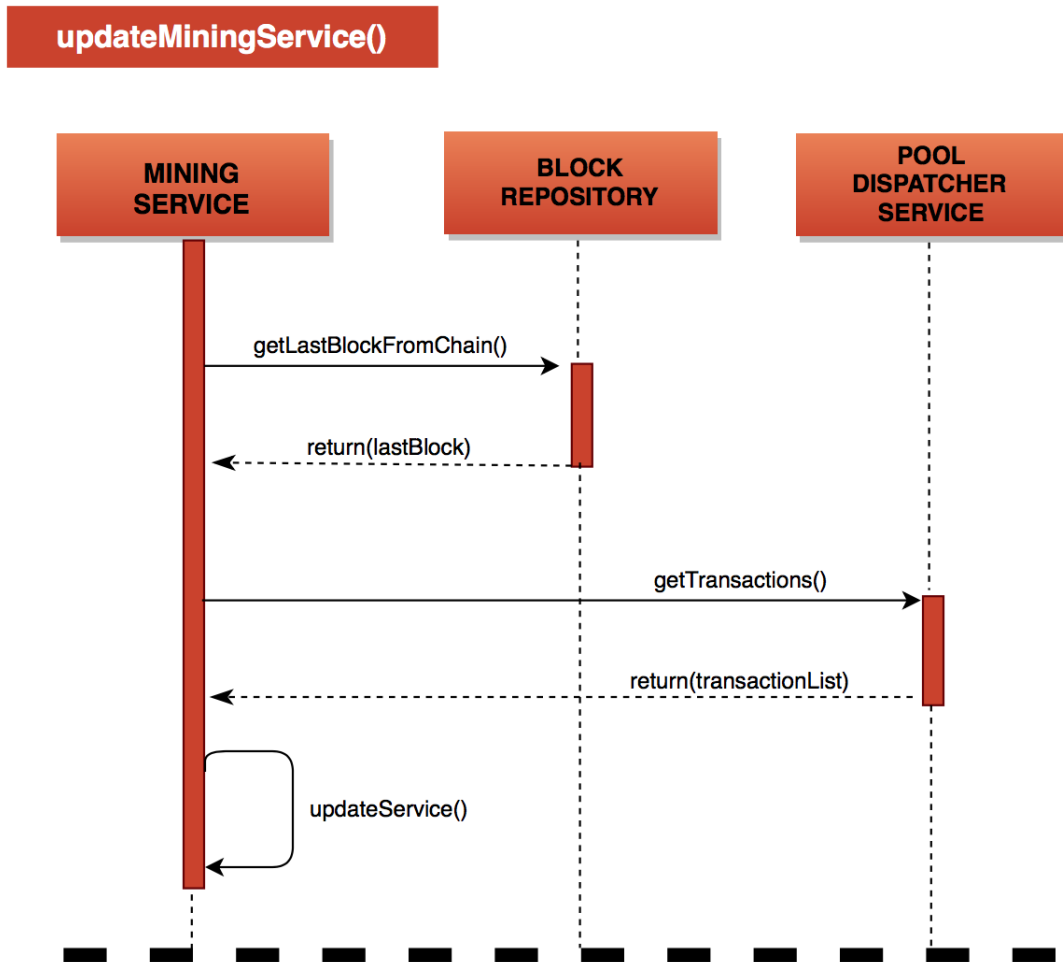


Figura 9: Flusso del metodo `updateMiningService()`.

seconda degli eventi che intervengono durante la sua esecuzione):

**mine(): caso in cui ho calcolato un nuovo blocco da verificare (ed isDone==TRUE)**

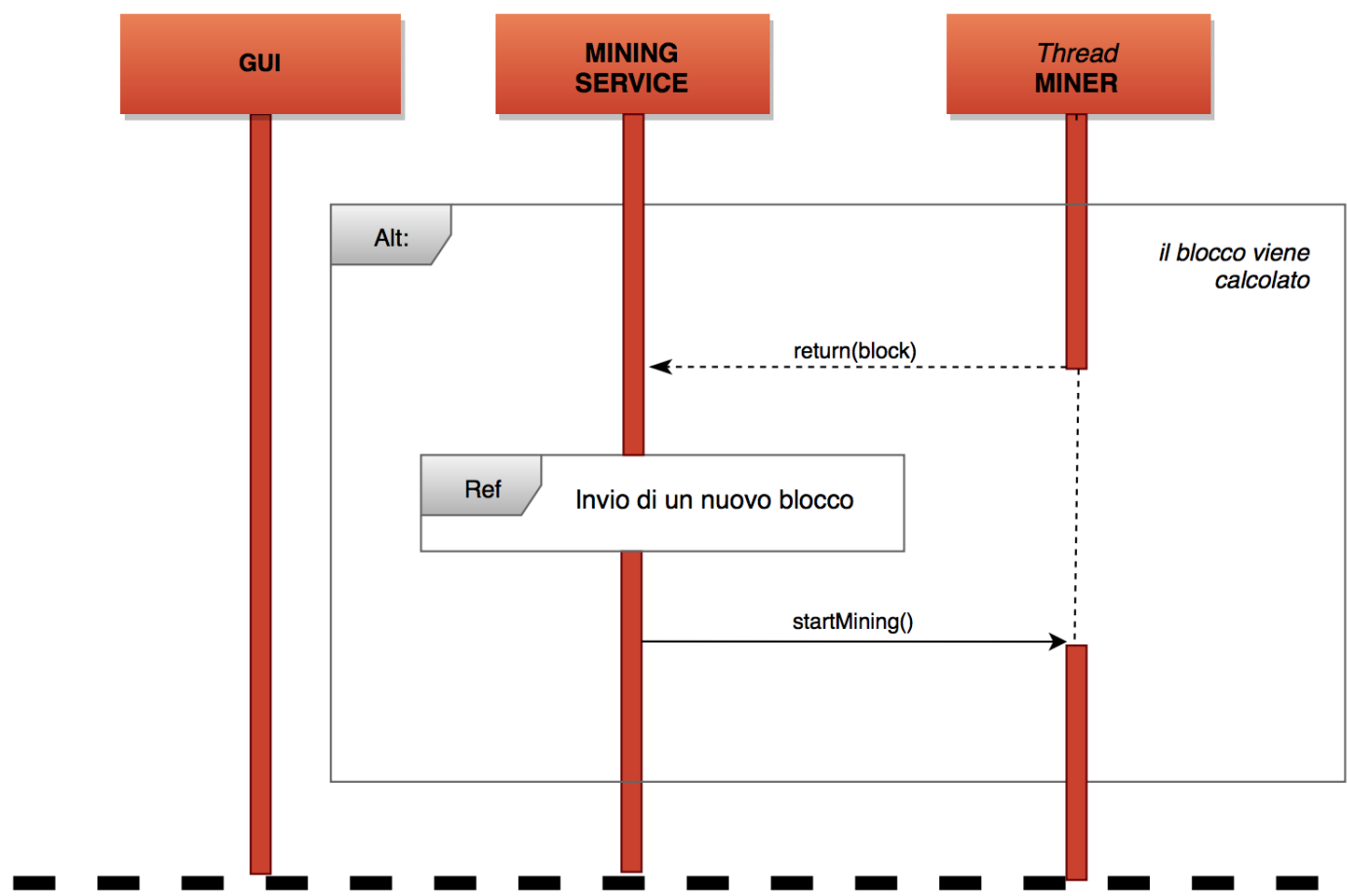


Figura 10: Flusso del metodo `mine()` nel caso in cui viene calcolato un nuovo blocco in attesa di validazione. Nel codice ciò equivale al verificarsi di `isDone==TRUE`.

### 2.6.6 `updateFilechain()`

**mine(): caso in cui arriva un nuovo blocco verificato (flagNewBlock==TRUE)**

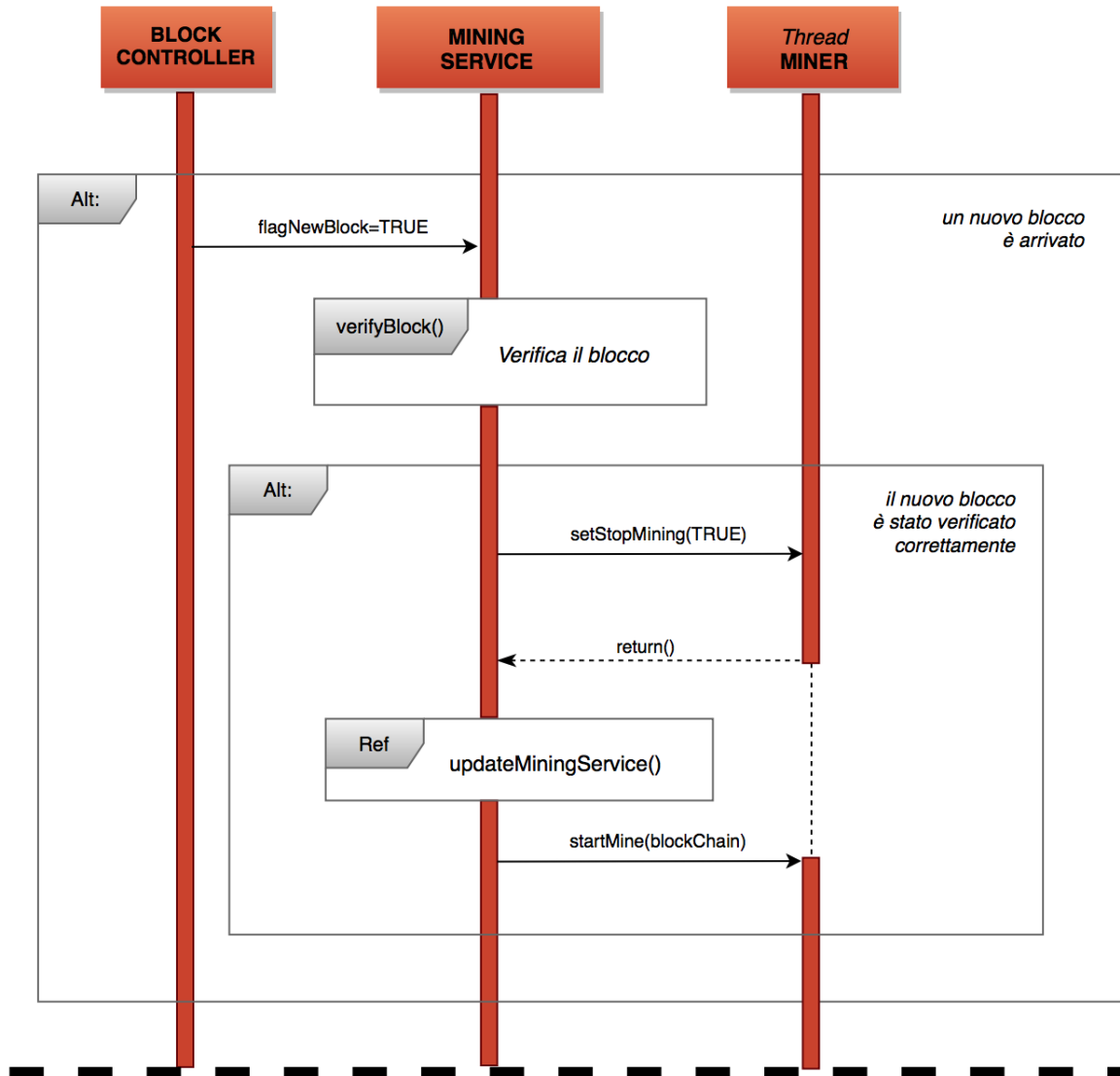


Figura 11: Flusso del metodo mine() nel caso in cui arriva un nuovo blocco calcolato e già verificato da qualche altro miner. Il miner deve pertanto verificarlo a sua volta ed aggiungerlo alla propria catena. Nel codice ciò equivale al verificarsi di *flagNewBlock==TRUE*.

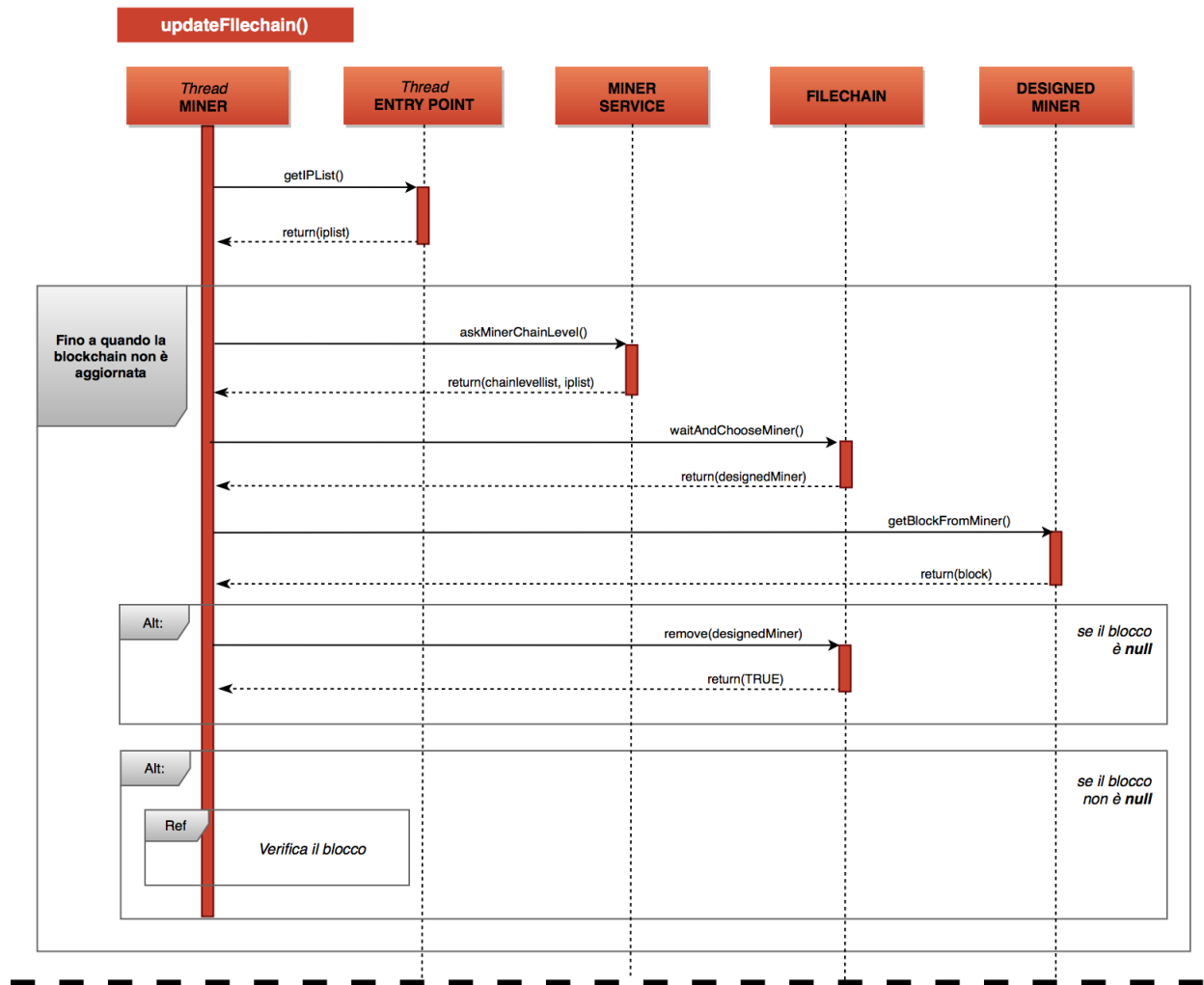


Figura 12: Flusso del metodo `updateFilechain()` chiamato ogni volta che è necessario l'aggiornamento della propria blockchain tramite confronto del *chain level*.

### 2.6.7 `verifyBlock()`

Il metodo **`verifyBlock()`** effettua la verifica del blocco; scegliamo di mostrarne la logica attraverso un sequence diagram delle chiamate ai metodi e di descriverne testualmente l'esecuzione per favorirne la leggibilità e la comprensione.

La verifica di un blocco coinvolge 3 metodi, i cui primi 2 vengono chiamati

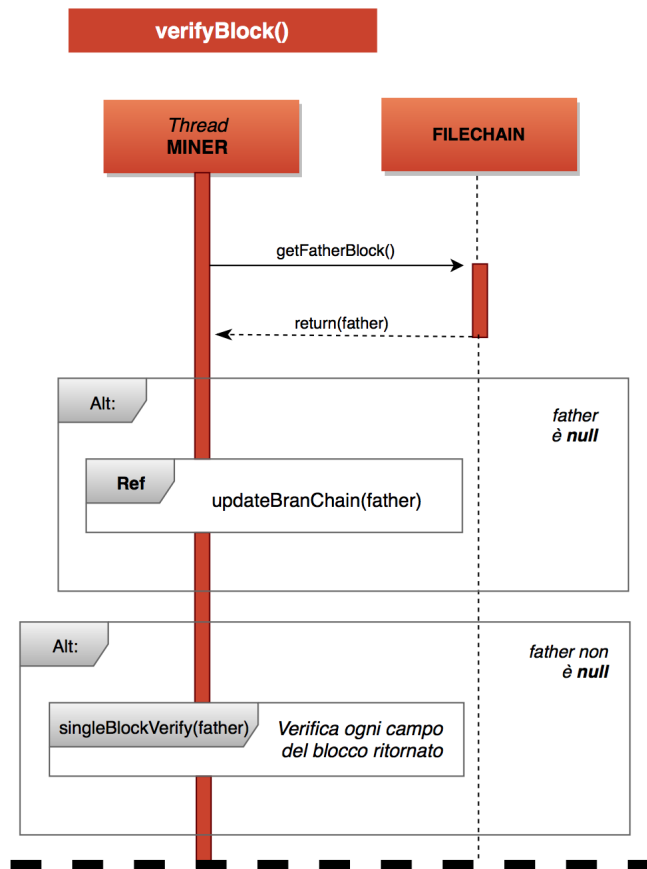


Figura 13: Flusso del metodo `verifyBlock()` chiamato ogni volta che si vuole verificare un blocco ricevuto o creato.

in mutua ricorsione fino all'allineamento del miner alla catena più lunga nella rete, facendo partire solo successivamente, tramite il terzo metodo, l'operazione di verifica vera e propria dei blocchi; più in dettaglio:

1. In **`verifyBlock()`**, il blocco passato come parametro viene analizzato a partire dal *father* del blocco nella catena, trovato tramite il suo valore di hash; se la ricerca non va a buon fine (ed il blocco padre ritorna un valore **`null`**) viene richiamato **`updateBranChain()`**. Alternativamente, se il blocco esiste viene ricercato tramite hash e passato come parametro per la verifica del blocco singolo al metodo **`singleBlockVerify`**
2. In **`updateBranChain()`** il miner, percorrendo a ritroso la catena, effettua un update così come lo abbiamo descritto in **`updateFilechain()`** e, guardando al *father*, decide se richiamare mutuamente la verifica del nuo-

vo blocco ricevuto durante la fase di update (chiamata a **verifyBlock()**), o, alternativamente, se il blocco ritornato è **null**, nessun miner nella rete può fornire il blocco voluto e quindi si ferma

In altre parole, se il confronto tra l'hash in possesso del miner ed il campo *hash previous block* del blocco da verificare non va a buon fine, bisogna dedurre che il blocco chiesto al designed miner durante l'operazione di *update* potrebbe essere relativo ad una catena più lunga di quella in possesso.

In questo caso, il miner procede a ritroso, chiedendo al designed miner anche i blocchi precedenti, fino a trovare una corrispondenza con l'hash posseduto. A questo punto, il miner verifica la quantità di blocchi necessari a raggiungere la chain level più lunga sulla rete, creando un *branch* (e quindi una diramazione) all'interno della catena. Come è usuale per il protocollo blockchain, vincerà il branch relativo alla catena più lunga.

Il terzo metodo **singleBlockVerify()** (il cui flusso non è stato illustrato) svolge le seguenti operazioni in ordine:

- Verifica della firma digitale, per mezzo di un'operazione di decifratura tramite chiave pubblica del miner
- Verifica della proof-of-work, eseguita nel seguente ordine:
  1. Considero l'hash del blocco e il nonce
  2. Contatto il Pool Dispatcher per conoscere la complessità della transazione (ad esempio: 28)
  3. Verifico che l'hash abbia i primi  $\lceil 28/8 \rceil = 3$  byte a zero
  4. Verifico che i restanti  $24\%8 = 5$  bits siano nella forma  $00000XX \dots X$  effettuando un'operazione di AND bit-a-bit con una sequenza nella forma  $1111100 \dots 0$  generata dinamicamente grazie allo *shifting* di un vettore composto da soli 1
- Verifica del merkle tree, tramite un confronto dei sottoalberi radicati nella merkle root
- Verifica che le transazioni siano uniche, e quindi che nessuno tra i predecessori del blocco arrivato abbia tale transazione al suo interno
- Infine, il blocco viene inserito nel database (vedi figura 6d)

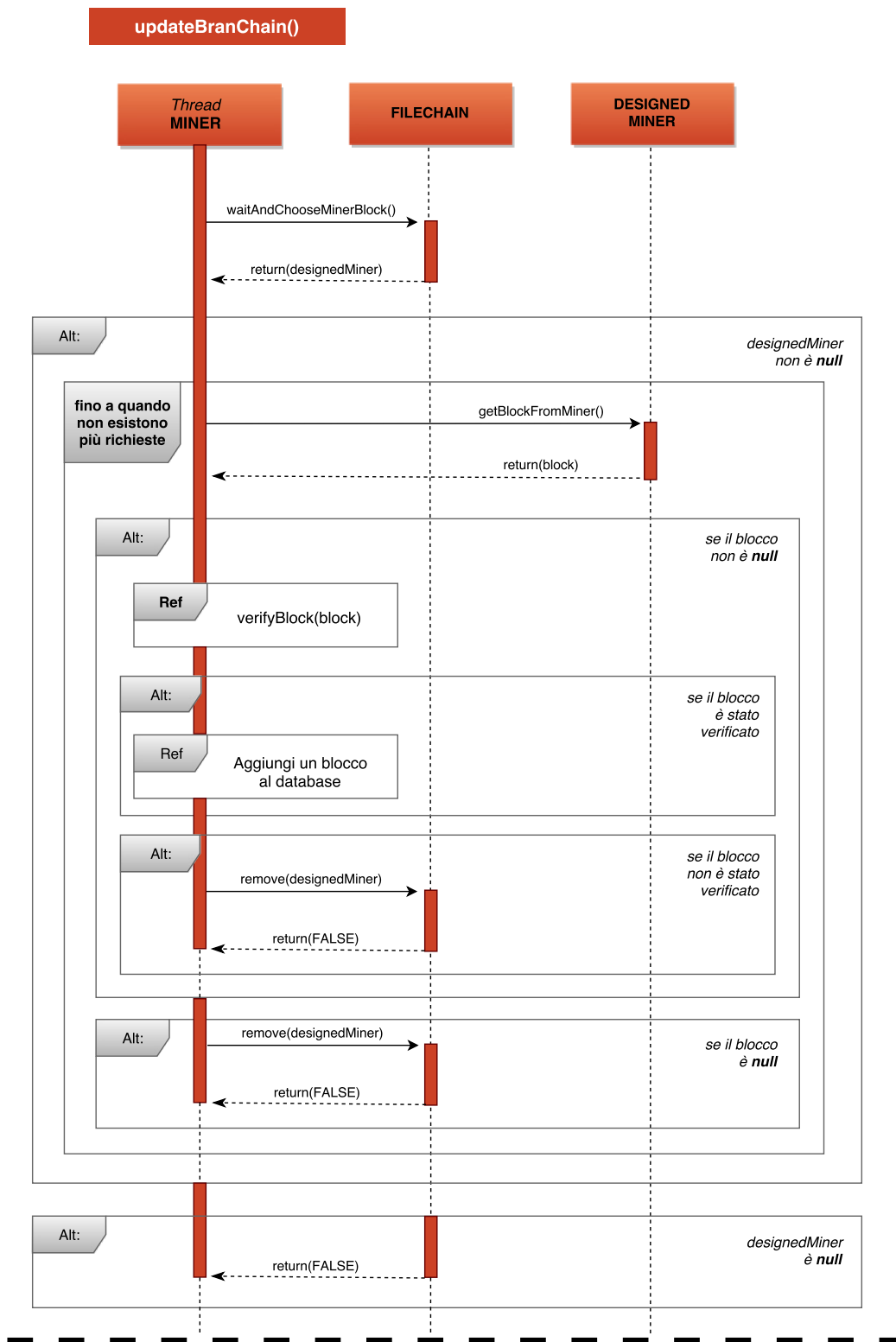


Figura 14: Flusso del metodo `updateBranChain()`, necessario quando la catena deve recuperare blocchi a ritroso non precedentemente inseriti per aggiornarsi al *level chain* della rete. Considera il caso in cui si ha il *branching* della catena.

## 3 Implementazione dell'architettura

Relativamente all'implementazione del progetto, consideriamo le seguenti entità costruite nei subpackage relativi a MINER.DAO.

### 3.1 Strutture dati

#### 3.1.1 Transaction

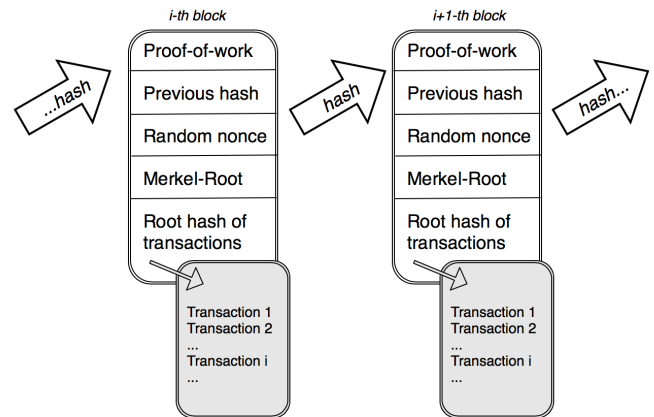
Una *transazione* comprende:

- *Hash file* di tipo String
- *Name file* di tipo String
- *Public key user* di tipo String
- *Index position in block* di tipo Integer
- *Hash transaction block* di tipo String

#### 3.1.2 Block

Il *blocco* (mostrato per comodità di lettura nell'immagine sulla destra) comprende:

- *Proof-of-work* di tipo String
- *Merkle-root* di tipo String
- *Public key miner* di tipo String
- *Nonce* di tipo Integer
- *Chain level* di tipo Integer
- *Creation time* di tipo String
- *Hash previous block* di tipo String
- *Digital sign* di tipo String



#### 3.1.3 User

L'*utente* comprende i seguenti attributi:

- *Public key user* di tipo String
- *Hash public key user* di tipo String
- *Name user* di tipo String



- *Surname user* di tipo String
- *Username* di tipo String
- *Password* di tipo String
- *Email* di tipo String

#### 3.1.4 Citation

La *citazione* comprende i seguenti attributi:

- *Hash cited* di tipo String
- *Hash citing* di tipo String

#### 3.1.5 Login

La fase di *login*, trattata come un'entità a se stante, comprende i seguenti attributi:

- *Hash public key user* di tipo String
- *Public key user* di tipo String
- *Username* di tipo String
- *Password* di tipo String
- *Role* di tipo String

### 3.2 Selezione delle possibili transazioni citabili

Una *citazione* è una lista di transazioni, ovvero viene applicata la struttura Java *List* a delle transazioni, come definite nella classe *Transaction.java*. Quando un utente, al momento dell'inserimento di un nuovo file verso il sistema (e quindi il Pool Dispatcher), decide di voler citare terzi già riconosciuti all'interno della catena di blocchi, seleziona tra un insieme di possibili transazioni quali citare per il proprio file.

Introdurre questo problema ci porta a chiederci immediatamente dopo: quali sono queste possibili transazioni? Come gestire tutti quei nodi che potrebbero formarsi in rami destinati a morire durante il ciclo di vita della Filechain?

Consideriamo il la figura 15 ed assumiamo le seguenti notazioni:

- Lo schema è organizzato per *livelli* da sinistra verso destra dove i nodi celesti dell'ultimo livello sono gli ultimi aggiunti alla catena

- Benché esistano più diramazioni nella Filechain, consideriamo **catena effettiva** il sottografo che attraversa la catena passando per il ramo più lungo (e quindi quello che sopravvivrà nella catena)
- La linea rossa è denominata **livello di paranoia** ed identifica la quantità di livelli che l'utente sceglie di aspettare prima di dichiarare *stabile* la transazione; la stabilità implica che con alta probabilità la transazione apparterrà alla *catena effettiva*
- Ci riferiremo all'insieme dei nodi che si trovano alla destra della linea rossa come a **livelli concorrenziali**, ossia transazioni (spesso le stesse calcolate da più utenti contemporaneamente) che concorrono quindi per appartenere alla *catena effettiva* o rimanere su rami deceduti della catena dove non verranno presi ulteriormente in considerazione; è importante infatti per l'utente poter citare solo quelle transazioni che saranno ritenute stabili onde evitare ridondanze e citazioni non referenziabili
- Nel dettaglio, dato un livello concorrenziale  $i$  qualsiasi, chiamiamo **nodi concorrenziali** quei nodi del livello che concorrono da diverse ramificazioni della catena che si protraggono oltre il livello  $i$ -esimo (si tratta di nodi che attualmente non è possibile sapere se giaceranno sulla *catena effettiva*, o se rimarranno su *ramificazioni decedute*), mentre con **nodi esterni** chiameremo quei nodi foglia che compaiono nel livello ma fanno riferimento a branching precedenti al livello  $(i - 1)$ -esimo (si tratta di nodi che consideriamo comunque concorrenziali in quanto la ramificazione potrebbe protrarsi e vincere, ma con alta probabilità non faranno mai parte della *catena effettiva*, ma rimarranno su *ramificazioni decedute*)
- Un nodo da cui parte una diramazione della catena di blocchi è denominato **nodo comune** ai due (o più) branch; è da un nodo comune che inizia la concorrenza tra più transazioni (nel disegno è il nodo **a**)

Per ottenere una valida lista di *transazioni citabili*, vogliamo che vengano rispettati questi due punti:

1. Tutte le transazioni citabili non appartengono a più rami diversi
2. Senza perdita di generalità, l'insieme delle transazioni citabili è quello che proseguirà nella Filechain

Il che possiamo ottenerlo facilmente considerando il *livello di paranoia* e percorrendo a ritroso la catena fino al raggiungimento di un *nodo comune* per mezzo dell'algoritmo **alg** di cui viene fornito lo pseudocodice.

Raggiungere il nodo comune significa ritrovarsi nella situazione in cui  $|L_1| = 1$  dove il singleton coincide col *nodo comune*  $a$ ; in questo modo l'utente potrà scegliere tra un insieme ridotto di citazioni che però con alta probabilità saranno nella catena effettiva; questo procedimento ricorda i meccanismi con i quali si giudica una transazione "sicura" nell'ambito delle blockchain, ossia facendo

affidamento ad un valore (la *paranoia*) che rappresenta la quantità di blocchi verificati che un utente intende aspettare per ritenere stabile la catena fino alla propria transazione.

Listing 1: Alg: creare una lista di transazioni citabili

```
1  i ← ultimo livello della catena
2  L1 ← ∅
3  L2 ← {nodi del livelli i-esimo}
4  Fino a quando non raggiungo il nodo comune
5      L1 ←2
6      L2 ← {nodi father relativi ai nodi in L1}
7      Se il livello di paranoia è positivo
8          L2 ←2 ∪ {nodi esterni ai nodi in L1}
9      i ← (i - 1)
10     paranoid ← paranoid - 1
11 Restituisci la catena fino a quel punto
```

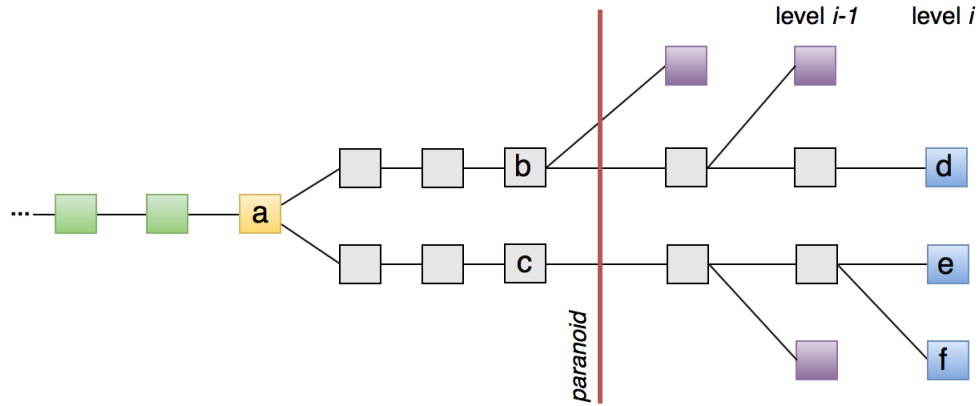


Figura 15: Consideriamo la seguente situazione (improbabile ma non impossibile); si supponga che la catena si trovi all' $i$ -esimo livello di costruzione, e che il nodo **e** voglia citare qualcuno imponendo un *livello di paranoia* pari a 3. L'algoritmo verrà inizializzato con i *nodi concorrenziali*, ovvero i nodi blu, e, iterativamente verranno aggiunti anche i *nodi esterni*, ossia i nodi viola che rientrano alla destra della linea (rossa) di paranoia (e che non sono father dei blocchi concorrenziali  $\{\mathbf{d}, \mathbf{e}, \mathbf{f}\}$ ). Andando a ritroso, una volta arrivati alla sinistra della linea rossa terminano i *livelli concorrenziali* e si procede fino al raggiungimento del *nodo comune* giallo **a** (**a** inclusa). A questo punto quindi l'utente **e** potrà citare tutto ciò che rientra nella *catena parziale effettiva* rappresentata dai nodi verdi. In futuro, quindi, solo una delle due (o più) diramazioni della catena sopravvivrà come *catena effettiva*; qualsiasi essa sia le transazioni mostrate sono comunque contenute, il che impedisce l'inconsistenza della catena, ovvero citazioni relative a transazioni in blocchi morti.

### 3.3 Protocollo di comunicazione

Il protocollo di comunicazione utilizzato è l'HTTP (HyperText Transfer Protocol) e viene gestito interamente dai controller contenuti nel package MINER.CONTROLLER. Lavora a livello applicativo e permette la trasmissione di informazioni in un'architettura tipica client/server.

All'interno della rete, più precisamente, viaggiano richieste di tipo GET e POST; la prima viene usata per ottenere i contenuti specificati nell'URI, la seconda invece per inviare informazioni al server (tipicamente l'URI identifica quali dati si sta inviando ed il contenuto viene incluso nel *body*).

Tali controller, che stanno a capo della gestione di questi messaggi, si pongono nel mezzo tra l'interfaccia ed il sistema, in modo da intercettare eventuali richieste mirate dall'utente al sistema; questo metodo permette l'efficiente gestione automatica dei messaggi HTTP tramite specifici metodi che ne identificano le differenti GET e POST, elaborando il tutto in background e rispondendo all'utente con un messaggio positivo o negativo (generalmente OK o una precisa *response*).

In CONTROLLERBLOCKREQUEST sono presenti i controller in grado di monitorare tutte le azioni relative ai blocchi. Si compone di:

- *newBlock (POST)*: ricezione di un blocco inviato da un miner presente nella rete (il cui valore è `/newBlock`) e risponde con `TRUE`
- *getBlock (GET)*: richiesta del blocco relativo al level chain indicato nell'URI (il cui valore è `/getBlockByChain`) e risponde con il blocco trovato
- *getBlock (GET)*: richiesta del blocco relativo all'hash indicato nell'URI (il cui valore è `/getBlockByHash`) e risponde con il blocco trovato
- *updateAtMaxLevel (GET)*: richiesta da parte di un miner della rete di tutti i blocchi a lui mancanti e risponde con il *chain level* del miner interrogato (in questo caso chi ha intercettato il messaggio)

In CONTROLLERPDEP sono presenti i controller relativi al Pool Dispatcher e all'Entry Point. Si compone di:

- *newUserConnection (POST)*: intercetta il tentativo di connessione di un utente (il cui valore è `/user_connect`)
- *newUserDisconnection (POST)*: intercetta il tentativo di disconnessione di un utente (il cui valore è `/user_disconnect`)
- *user\_ping (POST)*: intercetta il tentativo di ping (il cui valore è `/user_ping`) e risponde con un `ACK`

In `CONTROLLERUSERINTERFACE` sono presenti i controller in grado di intercettare i messaggi riferiti ad azioni generiche svolte dall'utente (sia nel ruolo generico che sotto le vesti di miner) verso il sistema. Si compone di:

- *startMining (GET)*: avvia l'operazione di mining richiesta dal miner e risponde con OK
- *stopMining (GET)*: interrompe l'operazione di mining come da richiesta del miner e risponde con OK
- *sendTransaction (POST)*: invia una transazione completa generata dallo user verso il sistema (precisamente, l'Entry Point) e quest'ultimo risponde con un ACK

### 3.4 Dettagli implementativi

Segue una descrizione sintetica ed essenziale del codice implementato per il funzionamento del progetto. Iniziamo dal descrivere i framework usati; il codice è stato interamente scritto in Java, tramite l'ausilio di:

**Spring**, framework open source nato per gestire la complessità nello sviluppo di applicazioni su Java; permette un facile testing del codice e fornisce una vasta quantità di tecnologie e strumenti che favoriscono il riutilizzo del codice già scritto, evitando così di appesantire ulteriormente il codice. La sua architettura si compone di 5 livelli: core e beans (per la creazione, gestione e manipolazione di oggetti di qualsiasi natura), data access (per l'accesso dei dati, tramite l'appoggio di Hibernate), aspect oriented (funzionalità di programmazione orientata agli aspetti, che permette di evitare l'utilizzo di Enterprise JavaBeans per gestire le transazioni), web (utile allo sviluppo di applicazioni web tramite la realizzazione del pattern architeturale Model-View-Controller) ed infine testing.



**JSON** (JavaScript Object Notation), formato adatto all'interscambio di dati fra applicazioni client-server, utile per la porzione di codice che mette in comunicazione l'applicativo con il database e ne permette l'esecuzione di query e la memorizzazione delle informazioni.

- Nel dettaglio, è stata coinvolta sia la libreria **Jackson**



**JSON API** che la recente libreria Google **GSON** per la manipolazione degli oggetti Java in grado di convertirli efficientemente in oggetti JSON (e viceversa) e in Java Beans (e viceversa).



**Apache Maven**, si tratta di un sistema dichiarativo per la build automation, utile durante la gestione del progetto Java; sfrutta un costrutto XML particolare denominato POM che descrive le dipendenze fra il progetto e le varie versioni di librerie che sono necessarie all'esecuzione.



- **MySQL**, è un RDBMS (Relational Database Management System) open-source, ovvero uno strumento di amministrazione per la creazione e gestione di database coinvolti in applicazioni web basate sul modello relazionale introdotto da Edgar F. Codd.



- **Hibernate**, piattaforma open source che fornisce un servizio di object-relational mapping (ORM) ovvero la gestione della persistenza dei dati sul database attraverso la rappresentazione ed il mantenimento su database relazionale di un sistema di oggetti Java.



- **AngularJS**, è un framework web open source sviluppato principalmente da Google per affrontare le difficoltà nello sviluppo di applicazioni a singola pagina, semplificandone lo sviluppo ed il testing favorendo l'architettura MVC. Lavora su attributi incapsulati nelle pagine HTML interpretandoli come direttive in grado di legare la pagina stessa al modello rappresentato da variabili JavaScript.



È stato inoltre necessario l'utilizzo del seguente programma per l'autenticazione nella VPN:

- **OpenVPN**, programma usato per creare tunnel crittografati punto-punto tra hosts, permettendo l'autenticazione tra loro per mezzo di chiavi condivise; sfrutta librerie di cifratura OpenSSL ed il protocollo SSLv3/TLSv1. Il suo corrispettivo in ambiente OSX è Tunnelblick.



Vediamo ora in dettaglio la struttura del codice.

### 3.4.1 I package

Segue l'elenco dei package di CS.SCRS contenuti in SRC/MAIN/JAVA.

- CONFIG, contiene i package *config.network*, *config.persistence*, *config.rest* e *config.ui* ed ha il compito di configurare il sistema
- CONFIG.NETWORK, contiene le classi relative alla configurazione della rete
- CONFIG.PERSISTENCE, contiene le classi relative alla configurazione della persistenza del database
- CONFIG.REST, configura la connessione dei miner
- CONFIG.UI, configura l'ambiente di lavoro
- MINER, contiene i package *miner.controllers*, *miner.dao.block*, *miner.dao.transaction*, *miner.dao.user* e *miner.models* ed ha il compito di gestire il mining
- MINER.CONTROLLERS, contiene i controller relativi ai miner
- MINER.DAO.BLOCK, contiene i metodi relativi ai blocchi
- MINER.DAO.CITATIONS, contiene i metodi relativi alla gestione delle citazioni
- MINER.DAO.LOGIN, contiene i metodi relativi alla gestione dell'autenticazione dell'user
- MINER.DAO.TRANSACTION, contiene i metodi relativi alle transazioni
- MINER.DAO.USER, contiene i metodi relativi agli user
- MINER.MODELS, si occupa della gestione della catena di blocchi e del *Merkle Tree*
- SERVICE.BEAN, gestisce i *bean*
- SERVICE.CONNECTION, per la gestione delle connessioni alla rete e degli IP dei miner
- SERVICE.IP, implementa la connessione al sistema da parte degli user
- SERVICE.MINING, contiene i metodi necessari a minare e verificare blocchi
- SERVICE.POOLDISPATCHER, implementa le richieste da indirizzare al Pool Dispatcher
- SERVICE.REQUEST, implementa tutte le richieste asincrone indirizzate al sistema
- SERVICE.UTIL, contiene metodi di utility (relative al sistema in generale)



### 3.4.2 Le classi

Verranno elencate adesso tutte le classi coinvolte nel progetto; per ciascuna classe segue una breve descrizione che ne mostra i vari metodi in essa contenuti. Nel caso di dicitura *get/set* s'intende l'esistenza di entrambi i metodi nella classe.

Il package CONFIG si compone delle seguenti classi:

- KEYSCONFIG, si occupa della configurazione delle chiavi pubbliche e private, con i relativi metodi *get/set*
- MINERCONFIG, crea diversi *bean* con lo scopo di rendere raggiungibili alcuni servizi usati di frequente nel codice. I *bean* sono sostanzialmente oggetti astratti che sono alla base del progetto e permettono il facile raggiungimento di porzioni di codice utili tramite classi differenti.
- MVCCONFIG, si occupa della configurazione del pattern architetturale MVC (*vedi "dettagli implementativi"*), il quale, sfruttando il paradigma *Model-View-Controller*, suddivide la gestione del sistema in tre differenti aree concettuali dando ordine al codice ed organizzandone la struttura come previsto da Spring

Il package CONFIG.NETWORK si compone delle seguenti classi:

- ACTIONS, implementa le varie interazioni possibili tra l'utente ed il sistema, quali *connect*, *disconnect*, *keepAlive* e *sendTransaction*, attraverso i relativi metodi *get/set*
- ENTRYPOINT, contiene i metodi relativi agli attributi dell'Entry Point, ossia *IP*, *port* e *baseUri*
- NETWORK, gestisce le componenti di base della rete, attraverso i metodi *get/set*; tali componenti sono il valore del *timeout* (espresso in secondi), l'*Entry Point*, il *Pool Dispatcher* e l'insieme delle *azioni* che permettono l'interazione tra utente e sistema
- POOLDISPATCHER, il quale, attraverso i metodi *get/set*, gestisce la *baseUri* del Pool Dispatcher

Il package CONFIG.PERSISTENCE si compone delle seguenti classi:

- DATABASE, si occupa della gestione dei parametri del *database*, necessario alla completa realizzazione del framework *JPA* (*vedi dopo*), attraverso i metodi *get/set*
- HIBERNATE, gestisce le proprietà di base per *Hibernate*, framework utilizzato per il collegamento tra Java e SQL

- JPA, Il framework *Java Persistence API* si occupa della gestione della persistenza dei dati implementandone le proprietà basilari
- PERSISTENCEJPACONFIG, è la classe che configura le proprietà sia di *JPA* che di *Hibernate*

Il package CONFIG.REST si compone delle seguenti classi:

- CONNECTION, gestisce, tramite i metodi *get/set*, i tempi di timeout relativi a: richieste, connessioni e lettura
- RESTCONFIG, si occupa della connessione dei miner

Il package CONFIG.UI si compone delle seguenti classi:

- AUICONFIG, la cui A nel nome indica il termine *astratta*, è la classe che rappresenta la configurazione dell'ambiente utilizzato
- DEVELOPMENT, gestisce le risorse statiche in fase di *prototipazione* tramite: path delle risorse associate all'ambiente, estensione dei file view e lista delle risorse associate.
- IUICONFIG, la cui I nel nome indica il termine *interfaccia*, è la classe utilizzata per recuperare la configurazione dell'ambiente di lavoro attuale
- PRODUCTION, gestisce le risorse statiche in fase di *produzione* tramite: path delle risorse associate all'ambiente, estensione dei file view e lista delle risorse associate
- RESUORCE, gestisce il *pattern* e la *location* delle risorse, tramite i metodi *get/set*.
- UiCONFIG, invece, è la classe che implementa l'interfaccia *IUiConfig*

Il package MINER si compone delle seguenti classi:

- FIL3CHAIN, è la classe main che avvia l'applicazione e si limita ad informare l'utente del corretto avvio

Il package MINER.CONTROLLERS (vedi *protocollo di comunicazione*) si compone delle seguenti classi:

- CONTROLLERBLOCKREQUEST, è il controller che intercetta l'arrivo di un nuovo blocco, restituisce un blocco a partire da *chain level* o *hash*, o il massimo *chain level* raggiunto
- CONTROLLERPDEP, è il controller che intercetta la connessione di un utente, la disconnessione e i ping

- CONTROLLERSTATISTICS, è il controller che si occupa di raccogliere le informazioni dalla Filechain per organizzarle mediante un *albero d-ario*<sup>1</sup> e mostrandola all'utente per mezzo della comunicazione con un *WidgetModel*
- CONTROLLERUSERINTERFACE, è il controller che implementa i metodi *startMining()*, *stopMining()* e *sendTransaction()*

I prossimi 5 package fanno riferimento ad un DAO (Data Access Object), ovvero un pattern architetturale per la gestione della persistenza; attraverso una classe (dotata di relativi metodi) si rappresenta un'astrazione dell'entità tabellare di un RDBMS tipica delle applicazioni web.

Questa metodica permette di stratificare e gestire al meglio l'accesso ai dati di una tabella tramite le query contenute nei metodi della classe e, grazie al suo accostamento al paradigma MVC, consente la separazione totale delle componenti dell'applicazione, forzandone la chiamata delle query dalle classi della business logic.

Diremo, generalmente, che il pattern DAO gestisce la persistenza con un alto livello di astrazione e garantendone una facile manipolazione dei dati.

Un'importante nota riguarda la presenza di un *repository* per ogni entità trattata nel DAO: altro non si tratta che di interfacce in grado di mappare funzionalità che sono relative ad un database. Queste astrazioni permettono di definire quali operazioni sarà possibile richiamare dalla business logic per ottenere le informazioni desiderate.

Il package MINER.DAO.BLOCK si compone delle seguenti classi:

- BLOCK, è la classe che include i costruttori dell'oggetto *Block*, oltre ai vari metodi *get/set* relativi ai suoi attributi
- BLOCKREPOSITORY, interfaccia relativa all'oggetto *Block* tramite la quale è possibile sfruttare hash e level chain per estrarre blocchi dal database

Il package MINER.DAO.CITATIONS si compone delle seguenti classi:

- CITATION, è la classe che include i costruttori dell'oggetto *Citation*, oltre ai vari metodi *get/set* relativi ai suoi attributi; introduce a sua volta la classe Key a causa della natura della relazione che dovrà gestire esclusivamente coppie di hash di chiavi
- CITATIONREPOSITORY, interfaccia relativa all'oggetto *Citation* tramite la quale è possibile estenderne i metodi e renderne più dettagliato il suo utilizzo

Il package MINER.DAO.LOGIN si compone delle seguenti classi:

---

<sup>1</sup>struttura dati ad albero con un numero *d* di figli per ciascun nodo

- LOGIN, è la classe che include i costruttori dell'oggetto *Login*, oltre ai vari metodi *get/set* relativi ai suoi attributi
- LOGINREPOSITORY, interfaccia relativa all'oggetto *Login* tramite la quale è possibile estenderne i metodi e renderne più dettagliato il suo utilizzo

Il package MINER.DAO.TRANSACTION si compone delle seguenti classi:

- TRANSACTION, è la classe che include i costruttori dell'oggetto *Transaction*, oltre ai vari metodi *get/set* relativi ai suoi attributi
- TRANSACTIONREPOSITORY, interfaccia relativa all'oggetto *Transazione* tramite la quale è possibile effettuare ricerche di blocchi (e dunque liste di transazioni) tramite hash

Il package MINER.DAO.USER si compone delle seguenti classi:

- USER, è la classe che include i costruttori dell'oggetto *User*, oltre ai vari metodi *get/set* relativi ai suoi attributi
- USERREPOSITORY, interfaccia relativa all'oggetto *User* tramite il quale è possibile effettuare la ricerca di utenti tramite nome

Il package MINER.MODELS si compone delle seguenti classi:

- DNDTREE, è la classe che genera un albero *d*-ario, completa dei metodi *get/set*
- FILECHAIN, è la classe con tutti i metodi relativi alla *filechain*: inizializzazione, update della blockchain, gestione dei branch, richiesta del chain level ai miner, richiesta di blocchi ai miner, verifica dei blocchi, avvio del mining e sua gestione
- IP, classe che include i metodi *get/set* per la gestione degli indirizzi *IP*
- MERKLETREE, classe per la costruzione di un *Merkle Tree* (vedi capitolo precedente) a partire da una lista di hash di transazioni
- PAIRS, contiene metodi per la gestione di coppie di valori
- REQUESTBLOCKLIST, contiene metodi per la gestione della lista di blocchi
- REQUESTIPLIST, contiene metodi per la gestione della lista di *IP*
- WIDGETMODEL, è la classe che costruisce banner considerabili come widget il cui scopo è racchiudere e mostrare le informazioni raccolte durante la raccolta delle statistiche

Il package SERVICE.BEAN si compone delle seguenti classi:

- BEANSMANAGER, contiene i metodi *get/set* relativi a *connectionService*, *iPService*, *asyncRequest*, *filechain* e *restTemplate*

Il package SERVICE.CONNECTION si compone delle seguenti classi:

- CONNECTIONSERVICEIMPL, classe contenente i metodi per: l'inizializzazione dei valori delle entità della rete, la selezione dell'indirizzo *IP* da utilizzare nella sessione corrente, e l'ottenimento di una lista di indirizzi *IP*; gestisce, inoltre, la prima connessione all'Entry Point effettuata da un user

Il package SERVICE.IP si compone delle seguenti classi:

- IPSERVICEIMPL, classe che si occupa di gestire gli accessi alla lista *IP* tramite vari metodi *get/set*

Il package SERVICE.MINING si compone delle seguenti classi:

- IMININGSERVICE, la cui *I* nel nome indica il termine *interfaccia*, è la classe contenente il metodo necessario a minare un blocco
- MININGSERVICEIMPL, è la classe che implementa *IMiningService*
- VERIFYSERVICEIMPL, contiene invece i metodi necessari alla verifica di: un singolo blocco, la *proof of work*, la firma di un blocco, il merkle root di un blocco e l'unicità di una transazione

Il package SERVICE.POOLDISPATCHER si compone delle seguenti classi:

- POOLDISPATCHERSERVICEIMPL, comprende un insieme di metodi statici per le richieste di funzionalità dirette al Pool Dispatcher; questi metodi consistono nella richiesta della complessità attuale e della complessità al tempo della creazione di un blocco

Il package SERVICE.REQUEST si compone delle seguenti classi:

- ASYNCREQUEST, comprende l'insieme dei metodi per le richieste asincrone; questi metodi consistono nel trovare il massimo chain level, nel ping di uno user, nell'ottenere i blocchi a partire dall'hash o dal chain level, nell'ottenere la lista di *IP* dei miner collegati dall'Entry Point, e nell'eseguire le richieste di tipo GET e POST

Il package SERVICE.UTIL si compone delle seguenti classi:

- AUDIOUTIL, classe che contiene l'audio riprodotto al termine di ciascun blocco minato
- CONVERSIONS, classe che contiene i metodi per convertire una stringa JSON in oggetto e viceversa
- CRYPTOUTIL, classe che contiene i metodi relativi alla crittografia, come: generazioni di chiavi *RSA*, operazioni di cifratura e decifratura di una stringa, firma di un messaggio e conseguente verifica

### 3.5 Interfaccia grafica

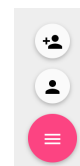
Verrà mostrato di seguito un esempio di navigazione nell'applicazione server front-end, ovvero scoprire tutte le funzionalità che sono a portata di click per lo user che interagisce direttamente col sistema.

Inizialmente, ci si troverà sulla **home page**:

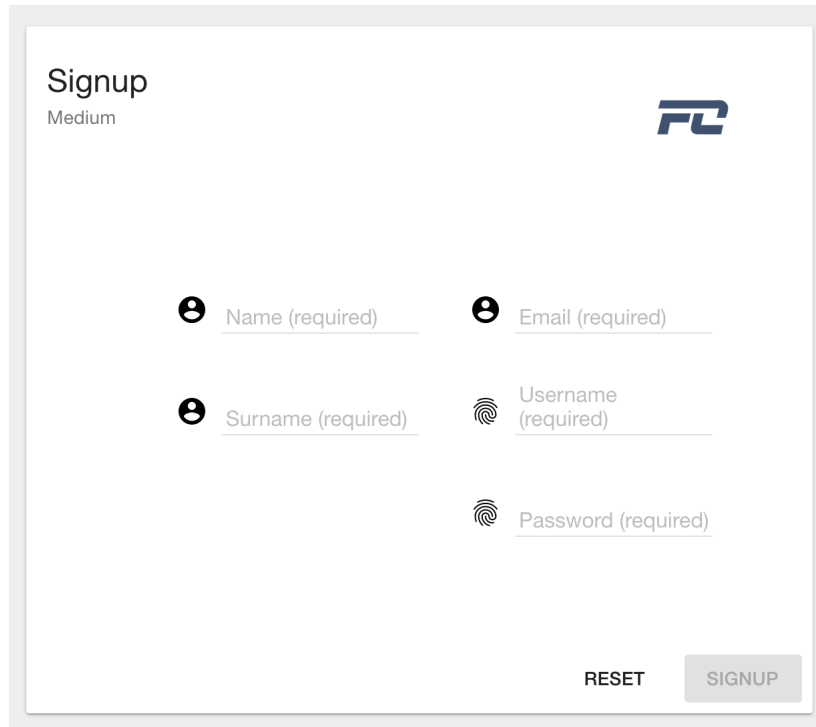


Figura 16: Home page della Filechain.

In basso a destra troviamo l'icona del **menù**, dal quale sarà possibile effettuare la registrazione (*signin*) al sistema e l'operazione di autenticazione (*login*) per mezzo dell'utilizzo della coppia username, password decisa al momento della registrazione.



Accedendo al **signup**, ci si troverà di fronte al seguente pannello, da compilare con i campi richiesti (in particolare gli obbligatori):

A screenshot of a web form titled "Signup" with the subtitle "Medium" and a logo in the top right corner. The form contains five input fields: "Name (required)", "Email (required)", "Surname (required)", "Username (required)", and "Password (required)". Each field has a small icon to its left (person for names, fingerprint for username and password). At the bottom right, there are two buttons: "RESET" and "SIGNUP".

Signup  
Medium

Name (required)

Email (required)

Surname (required)

Username (required)

Password (required)

RESET SIGNUP

Figura 17: Registrazione alla Filechain.

Nel caso di corretta compilazione, la registrazione andrà a buon fine e verrà mostrato il seguente messaggio di conferma:

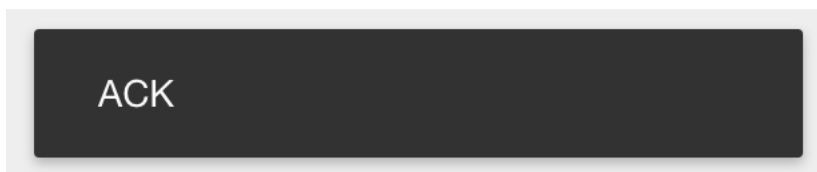


Figura 18: Messaggio in caso di corretta registrazione.

Accedendo invece al **login**, ci si troverà di fronte al seguente pannello di accesso, da compilare con la coppia username e password scelta durante la fase di registrazione:

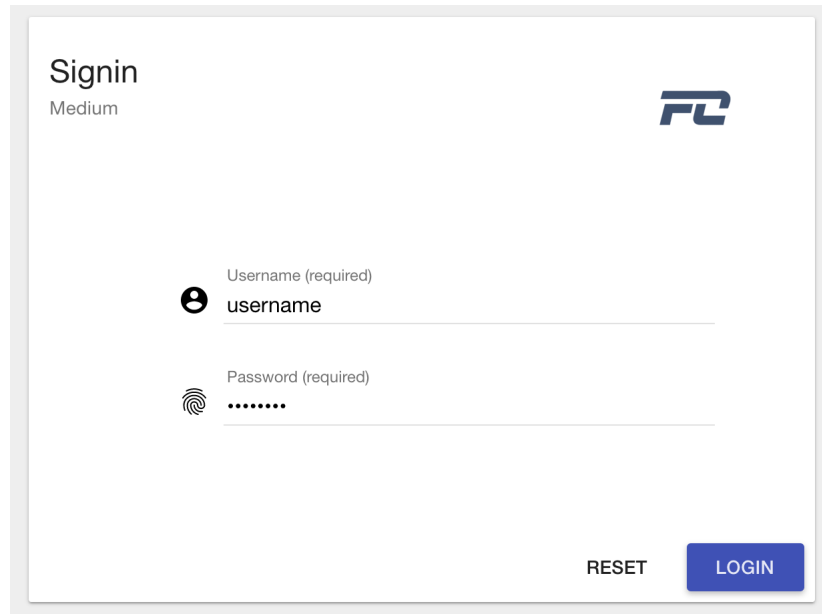
The image shows a 'Signin' form for 'Medium'. In the top right corner is the Filechain logo, which consists of the letters 'FC' in a stylized, italicized font. The form has two input fields: the first is for the 'Username (required)' and contains the text 'username'; the second is for the 'Password (required)' and is filled with dots. To the right of the password field are two buttons: a 'RESET' button and a blue 'LOGIN' button.

Figura 19: Login alla Filechain.

Il che ci condurrà alla **home page del wallet** dello user autenticato al sistema, come mostrata alla pagina successiva.



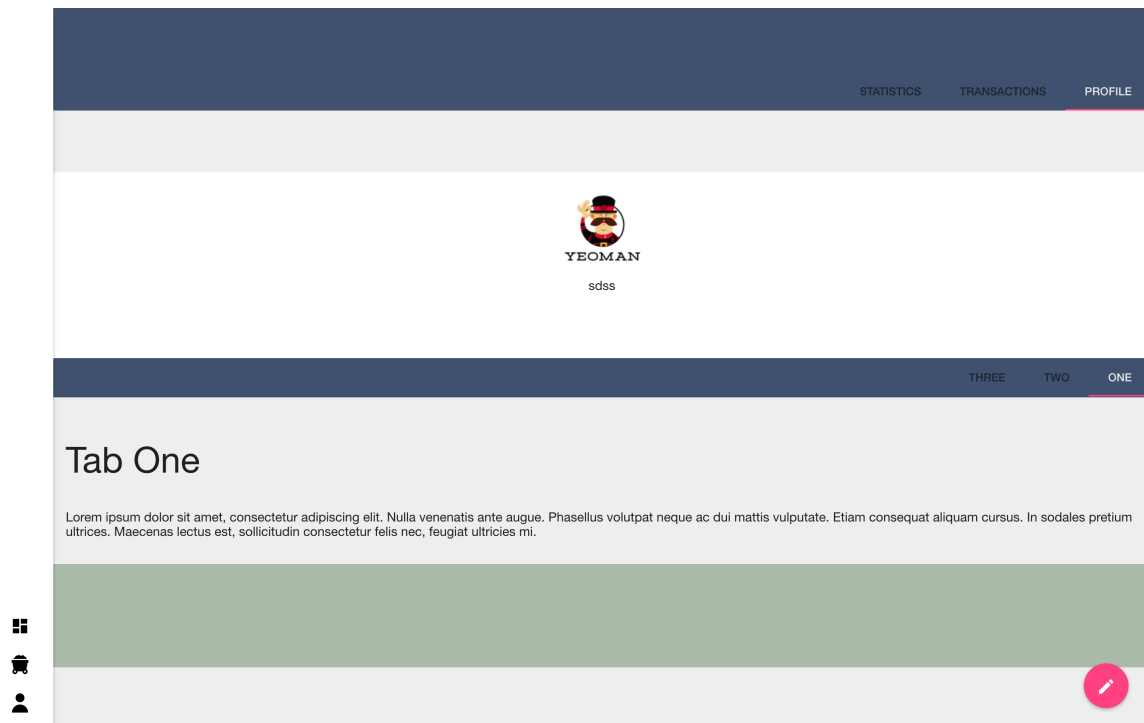


Figura 20: Home page della Filechain con user autenticato.

Questa pagina si presenta con tre zone di interesse: la sidebar sulla sinistra comprende il menù, che vedremo più tardi; i due riquadri orizzontali permettono di accedere a due zone del sistema: le tab, ovvero uno spazio riservato a comunicazioni, avvisi ed eventuali banner, e la zona di navigazione in alto a destra che ci permette di raggiungere facilmente:

- **Profile**, dal quale è possibile vedere un riepilogo generale delle informazioni dell'utente
- **Transactions**, dal quale è possibile vedere il totale delle transazioni contenute nella Filechain dell'utente autenticato, secondo un dato livello di paranoia (vedi figura 21a)
- **Statistics**, dal quale è possibile vedere le statistiche inerenti la Filechain (vedi figura 21b)

Transaction Filename
hack_wii_minimal.nus
sfondo.jpg
full_EUR.nus
CHANGELOG.FRANCAIS.txt
speedcam_type_5
full_JAP.nus
full_USA.nus

(a) Transazioni visibili all'utente.

Connected ips	Blocks	Hash Power	Max Chain Level
0	15	0	13

Fil3chain

< Page 1 >

Block details

ok	ok
ok	ok
ok	ok

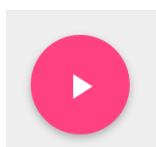
Card footer

(b) Statistiche della Filechain visibili all'utente.

Figura 21: Pagine raggiungibili dall'utente tramite il menù di navigazione posto sulla sinistra.

Tramite il **menù** sulla sinistra è possibile accedere a:

- **Filechain**, accede direttamente alle statistiche della Filechain
- **Miner**, che permette di avviare il servizio di mining (per mezzo del bottone nella figura 21a)
- **Wallet**, che torna nella home page del wallet dello user



L'user autenticato può avviare l'operazione di **mining** per mezzo del bottone riportato nell'immagine a sinistra, situato nella area *Miner* raggiungibile dal *menù*.

Affinché l'operazione possa avere effettivamente inizio, bisogna specificare l'IP della propria scheda di rete tramite il seguente pannello:

Figura 22: Selezione della scheda di rete.

A questo punto verrà restituita a video la seguente stringa di conferma:

```
{"selected_ip":"127.0.0.1","startTime":1468245568071}
```

Figura 23: Conferma di avvio del *mining*.

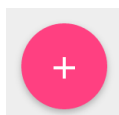
Muovendoci invece tra le **transazioni**, dalla schermata a figura 21a, vediamo come *inserire una nuova transazione nella Filechain*.

Si osservi che, cliccando su una qualsiasi delle transazioni presenti nella blockchain, è possibile vederne tutti i dettagli relativi come mostrato nel seguente esempio:

The screenshot displays the Filechain application interface. At the top, a dark blue header contains three navigation tabs: 'STATISTICS', 'TRANSACTIONS' (which is highlighted with a red underline), and 'PROFILE'. On the left side, there is a vertical sidebar with three icons: a grid, a shopping cart, and a person. The main content area is divided into several sections. The top section, titled 'Transaction Filename', shows 'hack\_wii\_minimal.nus'. Below this, a larger section titled 'sfondo.jpg' contains a 'Transaction details' table. The table lists the following information: Transaction Id (21b54a9b7d9010eb8b8786ce3c8675606819dd58f965c85cb87a3965bf680cf8), Hash File (1093a9d49d1852184fde9916a5a6f3bf85af247539201a24ecea69be19e60cd), File name (sfondo.jpg), Index in Block, Block Container (0000000f15e88ff7cd4a6d5b9b4a36e708445c33fc17331fccdec91878c8589d), and Citations (an empty array). Below the table, another 'Transaction Filename' section shows 'full\_EUR.nus'. At the bottom of the main content area, another 'Transaction Filename' section is partially visible. On the right side of the interface, there is a red circular button with a white plus sign.

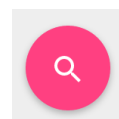
Transaction details	
Id Transazione	21b54a9b7d9010eb8b8786ce3c8675606819dd58f965c85cb87a3965bf680cf8
Hash File	1093a9d49d1852184fde9916a5a6f3bf85af247539201a24ecea69be19e60cd
File name	sfondo.jpg
Index in Block	
Block Container	0000000f15e88ff7cd4a6d5b9b4a36e708445c33fc17331fccdec91878c8589d
Citations	[]

Figura 24: Dettagli della transazione relativa al file *sfondo.jpg*.



Per mezzo del bottone riportato nell'immagine a sinistra, è possibile accedere alla sottoarea del servizio che permetterà l'**upload** del file sulla Filechain.


A questo punto, il bottone diventerà quello mostrato nell'immagine a destra che permetterà di selezionare il file che si intende inserire nella Filechain, e si verrà reindirizzati al pannello automaticamente compilato relativo a tale file.



Il pannello si presenterà in questa forma:

### Transaction

This card shows the details of the file inserted in the transaction



<b>File name</b>	1_bitcoin.png
<b>Last Modified Date</b>	Mar 11, 2016 3:45:15 PM
<b>Type</b>	image/png
<b>Size</b>	77852
<b>Sha256</b>	65af0d34ae13ebb1e1c7cd8d12131218fcd0ddb217467baba8b36e7b83f6e3

[DELETE](#) [ADD CITATIONS](#) [SUBMIT](#)

Figura 25: Inserimento di una nuova transazione.

A questo punto si potrà interrompere l'operazione sostenuta finora, eliminando per mezzo del pulsante **DELETE**, o scegliere di aggiungere citazioni al file richiamando precedenti transazioni contenute nella Filechain; tutto ciò è gestibile per mezzo del pulsante **ADD CITATIONS** che reindirizzerà l'utente alla pagina delle citazioni, come mostrato nella figura alla seguente pagina.

Citation

Transaction Filename

hack\_wii\_minimal.nus

☐

Transaction Filename

sfondo.jpg

☒

Transaction Filename

full\_EUR.nus

☒

Transaction Filename

CHANGELOG.FRANCAIS.txt

☐

Transaction Filename

speedcam\_type\_5

☐

Transaction Filename

full\_JAP.nus

☐

CANCEL


ADD

Figura 26: Elenco delle transazioni citabili dall'utente nel momento in cui si intende inserire un nuovo file nella Filechain; in questo esempio si vogliono citare due transazioni: *sfondo.jpg* e *full\_EUR.nus*

Premendo ADD, si confermano le citazioni selezionate ed il pannello relativo alla transazione viene automaticamente aggiornato, come mostrato nell'esempio:

### Transaction

This card shows the details of the file inserted in the transaction



File name	1_bitcoin.png
Last Modified Date	Mar 11, 2016 3:45:15 PM
Type	image/png
Size	77852
Sha256	65af0d34ae13ebb1e1c7cd8d12131218fcd0ddbad217467baba8b36e7b83f6e3
Citations	<div><div>1093a9d49d1852184fde9916a5a6f3bf85af247539201a24ecea69be19e60cd</div><div>X</div></div> <div><div>9e4e20307df8857826a05615ec0ef0cfb12d81eb75e71be05fbf1ba98a61dec6</div><div>X</div></div>

DELETE

ADD CITATIONS

SUBMIT

Figura 27: Conferma di avvio del *mining*.

In definitiva, confermando la transazione ed inviandola al sistema dopo un breve upload (la cui durata può variare a seconda della dimensione del file) per mezzo del pulsante SUBMIT, verrà riportato il seguente messaggio di conferma:

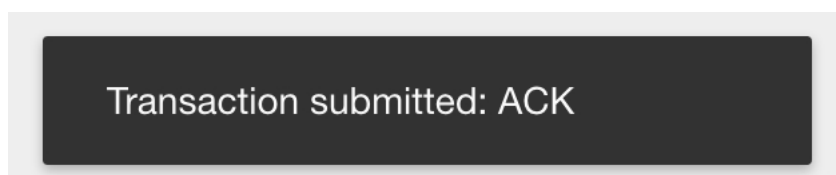


Figura 28: Conferma sulla transazione correttamente eseguita.

## 4 Criticità e conclusioni

Si vuole porre l'attenzione su alcuni punti di criticità, i quali non dipendono strettamente dall'architettura del sistema.

Il diritto d'autore è un ambito spinoso, dove piccole imprecisioni possono portare all'indimostrabilità della proprietà intellettuale.

Cerchiamo ora di analizzare alcune situazioni ambigue che potrebbero sorgere durante l'uso reale di questo sistema.

**Invio di file nello stesso istante** Un fattore critico è il tempo. Ad esempio, due utenti inviano due file *simili* tra loro, causando un conflitto. A questo punto, bisogna capire se i due file possono essere caricati insieme o se uno dei due deve essere rigettato (scegliendo come possibile criterio il *timestamp*).

Nel nostro caso, non avendo implementato il riconoscimento di similarità tra file, tali file verranno entrambi caricati nel sistema. Sarà successivamente compito dei relativi autori far valere i propri diritti.

**This is mine!** Come spesso accade nella realtà, c'è chi ha interesse ad appropriarsi delle idee altrui. Supponiamo che un utente entri in possesso di un file non suo e che, a questo punto, lo invii firmandolo a suo nome. L'utente che è realmente autore del file, provando ad inviarlo a sua volta in un secondo tempo, potrebbe trovare la propria richiesta rigettata. Come bisogna comportarsi?

Il sistema da noi progettato si accerta che, una volta che i file siano inseriti al suo interno, essi rimangano impressi inalterati in modo da poterne dimostrare la proprietà, dove per proprietario in questo caso s'intende l'utente che ha inserito il file nel sistema.

Se una violazione avvenisse prima dell'inserimento, dovrà essere compito del reale autore dimostrare la proprietà intellettuale di tale file, avendo a propria disposizione i normali mezzi usati comunemente, prima fra tutti la denuncia del furto agli organi di competenza.

**Durata** La normativa prevede una durata del copyright limitata nel tempo, variabile significativamente a seconda della categoria merceologica tutelata (medicinali, brani musicali, software, ecc.).

Il periodo di copyright dovrebbe consentire di avere un adeguato margine di guadagno e di recuperare i costi che precedono l'entrata in produzione e la



distribuzione del prodotto. In linea di principio, la durata è proporzionale ai costi da remunerare.

Bisogna quindi mantenere nella blockchain tale file fino alla durata massima prevista, che in certi casi si estende anche dopo la morte dell'autore; in quest'ultimo caso potrebbe essere ceduto in eredità se previsto dalla legge.

**Modifiche successive** I diritti d'autore possono essere ceduti, decadere, o anche essere revocati se ottenuti indebitamente.

Il sistema deve tener conto di tali possibili modifiche ed essere in grado di adattarsi di conseguenza, senza per questo inficiare la sicurezza sulla quale si basa il concetto stesso di blockchain.

**Dimensioni della catena** Un problema intrinseco della blockchain è il continuo aumentare della sua dimensione.

È stato osservato che, all'interno del progetto *Bitcoin*, la dimensione della blockchain raddoppia ogni anno. Risulta chiaro che, in un sistema decentralizzato dove la blockchain è mantenuta in ogni nodo, il continuo e così rapido aumento della mole di dati è un problema da tenere in considerazione.

#### 4.1 Possibili sviluppi

Uno dei possibili sviluppi futuri, che si potrebbe considerare per un'eventuale espansione del progetto, riguarda i *ruoli* assunti dagli utenti durante l'interazione diretta con il sistema tramite la sua GUI.

Si potrebbero concedere facilmente così permessi differenti a seconda di quali categorie di utente si desidera avere all'interno del sistema. Per lo sviluppo ristretto a questo progetto, abbiamo considerato tutti gli utenti correttamente registrati ed autenticati al sistema come miners senza perdita di generalità.

#### 4.2 Conclusioni

Le proprietà della blockchain sono ampiamente applicabili in una vasta moltitudine di situazioni. Si pensi, ad esempio, al già citato caso delle valute elettroniche, o alla certificazione della filiera di prodotti di particolare valore.

Nel sistema da noi progettato, è stato preso in considerazione il mondo del diritto d'autore, cercando di sfruttare gli indubbi vantaggi della blockchain per poter affermare senza rischio di smentita la proprietà intellettuale su dei file.

All'interno del caso di studio preso in considerazione, i requisiti che il sistema dovrebbe avere corrispondono in modo puntuale ai vantaggi che la blockchain fornisce, come robustezza, correttezza, consistenza e protezione, per citarne tra i più lampanti. Quindi, nel problema analizzato, viene naturale pensare a tale tecnologia come una soluzione elegante ed efficace.

Una volta creata la struttura, il sistema progettato può essere ritoccato per accomodare di volta in volta i bisogni del cliente. Come ad esempio la necessità di accedere al servizio tramite programma per pc, sito internet o app mobile, o implementare una tecnologia di riconoscimento di file simili.