



UNIVERSITÀ DEGLI STUDI ROMA TRE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tesi Di Laurea

Studio dell'architettura payment channel per
blockchain basate su smart contract con
linguaggi turing completi

Laureando

Federico Ginosa

Matricola 457026

Relatore

Alberto Paoluzzi

Correlatore

Federico Spini

Anno Accademico 2017-2018

Questa è la dedica

Indice

1 Blockchain	9
2 Payment channel	17
Deploy dello smart contract	18
Apertura del canale	18
Join del canale	19
Modello propose-accept	19
Chiusura di un canale	21
Ritirare denaro da un canale	22
Argue di una propose	23
Free-option, controparte passiva e challenge close	24
Un approccio non punitivo	27
Inextinguishable Payment Channel	28

Elenco delle figure

1.1	Merkle tree	10
1.2	Blockchain con blocchi non manomessi	13
1.3	Blockchain con un blocco manomesso minato	14
1.4	Blockchain con blocchi manomessi	14
2.1	Propose firmata da Alice	20
2.2	Propose firmata da Bob	21
2.3	Stato corrente del payment channel	23
2.4	Primo automa temporizzato	26
2.5	Secondo automa temporizzato	28

Ringraziamenti

Grazie a tutti.

Capitolo 1

Blockchain

La blockchain è una struttura dati autenticata e distribuita. Generalmente una struttura dati permette due tipologie di operazioni:

- interrogazioni
- aggiornamenti

In un'ADS le interrogazioni forniscono assieme alla risposta una prova verificabile dell'integrità della soluzione fornita[3]. Cosa significa questo? Sia L una lista posseduta da Alice:

$$L = \{ L1 \rightarrow L2 \rightarrow L3 \rightarrow L4 \}$$

Partendo da questa lista, costruiamo un albero di questo tipo: per ciascun elemento della lista calcoliamo l'hash; poi per ciascuna coppia di hash calcoliamo l'hash della concatenazione, fino ad arrivare alla radice di quest'albero (vedi figura 1.1).

Alice possiede la lista L e l'intero albero, Bob invece possiede esclusivamente il nodo radice. Ipotizziamo che Bob voglia sapere se $L3$ sia contenuto in L ; fa una query di questo tipo:

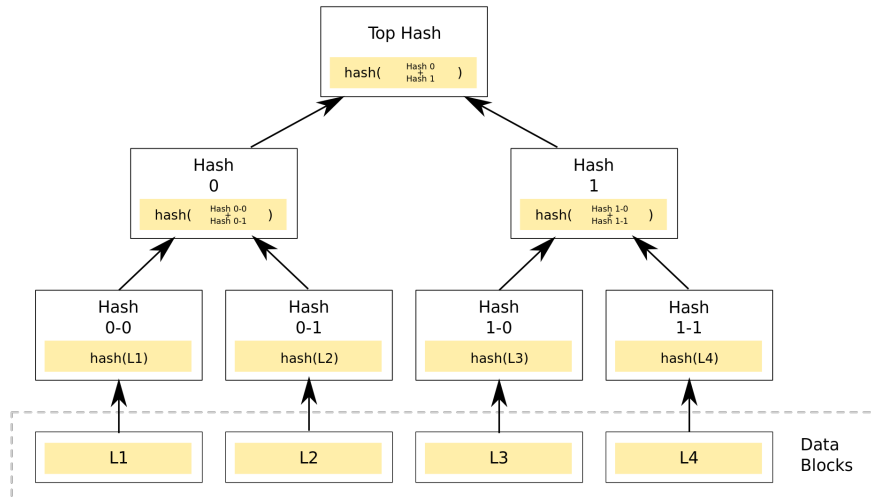


Figura 1.1: Merkle tree

query: L contains L3

Alice riceve questa query e risponde così:

result: true

proof: Hash(0), Hash(1-1), Hash(1)

Bob ottiene una risposta e oltre a questo ottiene una prova di essa. Infatti concatenando ed effettuando l'hash della prova fornita, coerentemente alla costruzione dell'albero, può facilmente ricostruire la root dell'albero, verificando che coincida con quanto posseduto[1]. Questa struttura dati autenticata prende il nome di Merkle Tree ed è uno degli elementi fondanti della blockchain. In particolare la blockchain è per l'appunto una catena di blocchi con un ordine definito. Ciascun blocco contiene un certo numero di transazioni, le quali a loro volta hanno un certo ordine. Per ciascun blocco viene costruito uno di questi alberi. Che utilità ha tutto questo? Sia B una blockchain con 500K blocchi e sia il peso di ciascun blocco pari a 1MB; il peso dell'intera blockchain

sarà di 500GB. Bob vuole sapere se tx_n sia contenuta nel blocco B_m , ma non ha a disposizione un nodo di calcolo con 500GB di archivio, cosa può fare? Semplicemente memorizza esclusivamente l'hash root di ciascun blocco. Ipotezzando che un hash abbia peso di 1 byte, lo spazio d'archiviazione necessario a Bob è minore di un 1MB. Bob avendo a disposizione l'hash root di ciascun blocco e la possibilità di contattare un full node¹, può verificare che una certa transazione sia contenuta in un certo blocco interrogando quest'ultimo. Il full node fornirà oltre alla risposta, la prova della risposta, il che garantisce a Bob che il risultato sia integro e che non sia stato manomesso da una terza parte. Per quanto riguarda gli aggiornamenti, generalmente devono rispettare determinate convenzioni. Come detto precedentemente la blockchain è una struttura dati distribuita; questo significa che non è memorizzata in un unico nodo, ma in più nodi che prendono il nome di peer. La distribuzione può essere effettuata con due diversi approcci:

- replicazione: ciascun peer possiede una copia della stessa struttura dati
- sharding: ciascun peer possiede una porzione della struttura dati
- replicazione/sharding: il mix delle due tecniche precedenti

Attualmente le implementazioni tecnologiche più comuni della blockchain si basano sul primo approccio, ovvero la replicazione, il che significa che esistono N nodi e che ciascun nodo possiede una copia della stessa struttura dati. Come detto gli aggiornamenti di una struttura dati devono seguire determinate convenzioni, una di queste convenzioni è la seguente: una volta inserito un blocco, questo blocco non può essere modificato. Come è possibile garantire una condizione simile in un sistema distribuito? Bada bene, questo problema non è risolto dal merkle tree, perché nel merkle tree si dà per scontato che Bob abbia quanto meno i root hash di ciascun blocco. In questo caso Bob deve ancora ottenere i root hash, quindi l'uso esclusivo del merkle tree non ci

¹nodo di calcolo che possiede l'intera blockchain

garantisce molto in tal senso. In altre parole ciò di cui abbiamo bisogno è un consenso della rete; la rete deve poter confermare a Bob che la struttura dati da lui posseduta non sia stata soggetta a manomissione. Un possibile approccio è il seguente: Bob sceglie un nodo casuale, scarica da esso l'intera blockchain e ne effettua l'hash. Poi chiede a ciascun nodo del sistema di votare per la validità della struttura dati. Ciascun nodo vota fornendo a Bob l'hash della struttura dati nello stato corrente. Se il risultato coincide con quanto posseduto da Bob per un numero di volte maggiore alla metà dei nodi presenti in rete, Bob prende per buona la struttura dati posseduta, altrimenti no. Qui sorge il primo problema: come facciamo ad essere certi in un sistema distribuito che un nodo fornisca un unico voto? Ovvero, come facciamo ad evitare che un nodo possa votare più volte? Estremizzando questo ragionamento, sia N il numero di nodi, un nodo x potrebbe votare $N/2 + 1$ volte, inducendo Bob a credere che una struttura dati manomessa in realtà non lo sia. Questo attacco prende il nome di sybil attack o pseudospoofing e l'approccio di voto democratico basato sull'entità del nodo non è immune ad esso. Un approccio alternativo è basato sulla proof of work; l'idea è che è possibile rendere difficile l'aggiunta o la modifica di nuovi blocchi. In particolare per aggiungere o modificare un nuovo blocco, occorre fornire una "prova di lavoro"; in altre parole bisogna dimostrare che per fare quell'aggiornamento della struttura dati si sia perso un certo quantitativo di tempo; questa prova è rappresentata da un nonce. Infatti per poter aggiungere un blocco alla catena, occorrerà fornire un numero, che se concatenato alle transazioni (o meglio al root hash) e successivamente hashato, restituisca un hash che abbia un numero D di zero iniziali. Questa operazione prende il nome di mining. D rappresenta la difficoltà corrente di mining e può variare, in particolare il valore di D viene calibrato sulla base del tempo impiegato dalla rete per minare gli ultimi blocchi. Di seguito lo pseudo codice di quello che potrebbe essere un algoritmo di mining.

```
def mining (block, D):
```

```

nonce = 0
do
  nonce++
  result = hash(nonce | block)
while (result.substring(0, D) == ('0' * D))

```

Il contenuto di un blocco è il seguente:

- nonce: come già detto è il valore che permette di variare il risultato dell'hash per far sì che rispetti una determinata proprietà (E.G. un certo quantitativo di zero iniziali)
- root hash: il nodo root del merkle tree costruito sulla base delle transazioni che si vuole aggiungere al blocco corrente
- prev_hash: l'hash del precedente blocco

Perché tutto questo dovrebbe risolvere il problema della manomissione di blocchi precedenti? In figura 1.2 è riportato lo stato corrente della blockchain *B*; ci sono quattro blocchi e tutti e quattro sono stati minati.

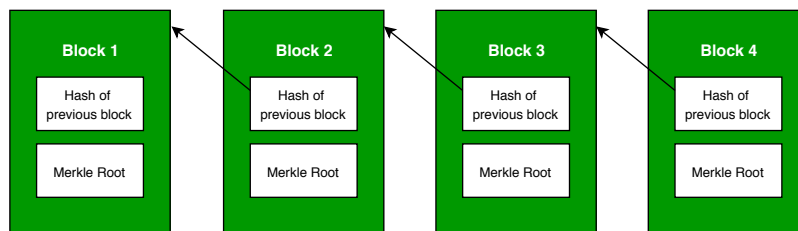


Figura 1.2: Blockchain con blocchi non manomessi

Immaginiamo ora che un nodo malevolo voglia manomettere il blocco numero due della catena.

Come è possibile vedere in figura 1.4, manomettere il blocco numero due, implica invalidare i blocchi 2, 3 e 4; infatti variando il valore del merkle root, varia anche il valore dell'hash, che con tutta probabilità non avrà più i primi *D* caratteri uguali a zero. Questo significa che il nodo malevolo dovrà calcolare

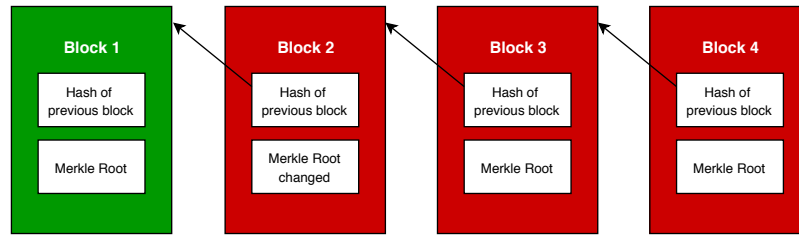


Figura 1.3: Blockchain con un blocco manomesso minato

un nuovo nonce per validare il blocco numero due. Se il nodo malevolo dovesse imbarcarsi in questa impresa, il risultato finale sarebbe quello riportato in figura 1.4, ovvero avrà validato il blocco numero 2, ma i blocchi numero 3 e 4 saranno ancora invalidi, questo perché essi codificano al loro interno l'hash del blocco precedente, che a questo punto sarà chiaramente diverso.

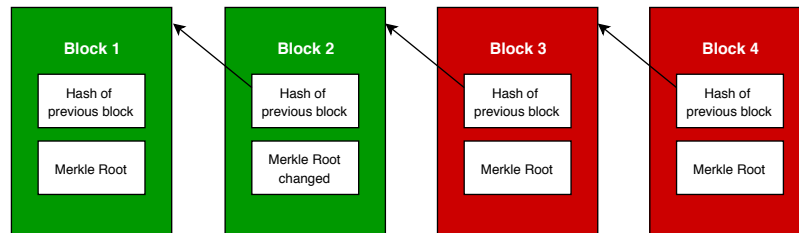


Figura 1.4: Blockchain con blocchi manomessi

Ciò che dovrà fare dunque il nodo malevolo è ricalcolare un nonce corretto per ciascun blocco successivo a quello manomesso. Questa operazione porterà via un gran quantitativo di tempo, durante il quale la rete avrà calcolato nuovi blocchi, di cui il nodo malevolo dovrà calcolare un nuovo nonce; è chiaro che in questa corsa contro il tempo, si ha qualche possibilità di vincere solo se si è in possesso di un gran quantitativo di potenza computazionale. In particolare sarà possibile riuscire a disfare un blocco confermato e ricalcolare il nonce dei successivi, solo se si possiede più del 50% della potenza computazionale dell'intera rete[2]. Portiamo all'estremo questo ragionamento e ipotizziamo che esista un nodo con potenza computazionale maggiore del 50%, come possiamo

evitare che il nodo disfi anche in questa situazione blocchi precedenti? L'idea qui è di assegnare a ogni miner capace di calcolare il nonce corretto dell'ultimo blocco una ricompensa. A questo punto, un nodo con tutta questa potenza computazionale, preferirà minare e ottenere i reward di ciascun blocco invece che manomettere blocchi passati, minando in questo modo la bontà del network del quale è padrone indiscusso[2].

Capitolo 2

Payment channel

In questo capitolo viene descritta una possibile implementazione di payment channel basata su smart contract, analizzando e mettendo a confronto vari approcci e motivando le scelte infine adottate.

Un payment channel è un canale virtuale che permette a due entità di scambiare valore facendo un uso limitato della blockchain, esso eredita tutte le proprietà di sicurezza della blockchain e inoltre garantisce:

- maggiore riservatezza: mentre le operazioni effettuate su blockchain sono pubbliche, quelle off-chain non lo sono
- maggiore scalabilità: la blockchain per sua natura permette di effettuare un numero limitato di operazioni al secondo, cosa non vera per le operazioni off-chain
- pagamenti istantanei: mentre nella blockchain le transazioni devono essere minate e successivamente confermate, in un payment channel le transazioni sono pressoché istantanee

Deploy dello smart contract

La prima azione che occorre effettuare per instaurare un payment channel tra due entità consiste nel deploy del relativo smart contract sulla blockchain. Questa operazione permette di ottenere il relativo indirizzo del contratto, che nelle successive fasi verrà adottato per richiamare le varie operazioni on-chain che si intende adottare. Il payment channel può trovarsi in uno di N stati; in fase di deploy lo stato è *INIT*.

Apertura del canale

La seconda operazione che è necessario effettuare consiste nell'apertura del canale. Alice apre il canale e blocca un quantitativo arbitrario di fondi all'interno dello smart contract; questo valore rappresenta il suo bilancio iniziale nel canale corrente. Oltre a bloccare i fondi, vengono fornite le informazioni necessarie all'apertura del canale, in particolare:

- **ethereumAddressB**: ovvero l'indirizzo ethereum della controparte, (E.G Bob)
- **host**: un host associato all'utente corrente, successivamente questo aspetto verrà chiarito
- **gp**: ovvero il grace period, anche questo aspetto verrà chiarito successivamente

L'apertura del canale può essere effettuata esclusivamente quando il payment channel si trova in stato di *INIT*, in qualunque altro caso verrà sollevata un'eccezione. Terminata l'esecuzione della procedura di open, lo stato del payment channel passerà da *INIT* a *OPENED*.

Join del canale

La terza e ultima operazione necessaria a instaurare un payment channel tra due entità consiste nel join del canale. Questa operazione può essere effettuata solo dall'utente con public address uguale a **ethereumAddressB** espresso da Alice in fase di apertura. Inoltre questa operazione può essere effettuata esclusivamente quando lo stato del canale è *OPENED*. Anche in questo caso l'utente richiamando la procedura bloccherà un numero arbitrario di fondi, che rappresenterà il bilancio iniziale dell'utente che ha effettuato il join, ovvero Bob. Inoltre Bob in fase di join fornirà il proprio **host** (anche qui, l'utilità di questa informazione verrà espressa successivamente). Una volta eseguita questa procedura, lo stato del canale passerà da *OPENED* ad *ESTABLISHED*.

Modello propose-accept

Quelle descritte fino a questo punto sono delle azioni che devono essere svolte in catena, necessarie a mettere in piedi un canale. Da questo punto in poi Alice e Bob possono scambiarsi dei pagamenti istantanei senza toccare la catena. L'approccio adottato è detto di propose-accept e si basa su due endpoint http pubblici esposti. Questi endpoint sono rispettivamente disponibili sotto l'host dichiarato in fase di apertura del canale.

- **/propose**: questo endpoint permette a uno delle due entità di richiedere alla controparte di concordare una nuova propose. Una propose è un'entità che propone di spostare una certa quantità di valore dall'entità richiedente alla controparte.
- **/accept**: questo endpoint permette a una delle due entità di accettare una propose precedentemente ricevuta sull'endpoint `/propose`.

Struttura di una propose

Una propose, ovvero una proposta di pagamento contiene essenzialmente quattro informazioni:

- **seq**: è il numero di sequenza, anche detto nonce; esso è un numero progressivo che parte da zero
- **contract__address**: l'indirizzo di riferimento del contratto che si sta utilizzando
- **balance__a**: il bilancio che si sta concordando per chi ha aperto il canale (E.G. Alice)
- **balance__b**: il bilancio che si sta concordando per chi ha fatto join del canale (E.G. Bob)

Poniamoci in una situazione d'esempio. Alice deploia e apre il canale, bloccando 1 eth. Bob effettua il join del canale e anche lui blocca 1 eth. Attualmente il bilancio di Alice e Bob è per entrambi di 1 eth e i fondi complessivi bloccati nel payment channel sono pari a 2 eth. Ora Alice vuole inviare 0.2 eth a Bob; per farlo:

1. Costruisce un'opportuna propose (vedi figura 2.1)
2. Effettua l'hash di quest'ultima
3. Firma l'hash della propose
4. Invia hash firmato e i valori in chiaro della propose

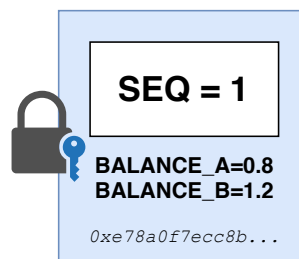


Figura 2.1: Propose firmata da Alice

Bob riceve la propose e:

1. Verifica di essere d'accordo con il contenuto
2. Verifica che la firma di Alice sia valida
3. Se decide di accettare la propose, in maniera speculare ad Alice effettua l'hash del contenuto della propose e lo firma con la sua di chiave privata
4. Infine invia la propose controfirmata ad Alice, tramite il suo endpoint pubblico /accept (vedi figura 2.2)

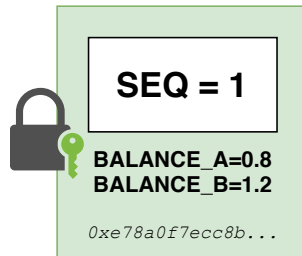


Figura 2.2: Propose firmata da Bob

A questo punto il bilancio off-chain di Alice e Bob è aggiornato, in particolare è rispettivamente di 0.8 ether per Alice e 1.2 ether per Bob. Seguendo lo stesso principio Alice e Bob possono concordare un numero arbitrario di propose (quindi di pagamenti), avendo sempre l'accortezza di aumentare il numero di sequenza; per esempio una successiva propose valida sarà:

```
seq=2; contract_address=0xe78...; a=0.9; b=1.1
```

In questa propose Bob ha inviato 0.1 eth ad Alice.

Chiusura di un canale

Come detto con questo modello di propose e accept Alice e Bob possono concordare un numero arbitrario di pagamenti. Questi pagamenti sono

pressoché istantanei e aggiornano il bilancio off-chain delle controparti. Ma cosa significa aggiornare il bilancio off-chain? Quando Alice invia una propose firmata e Bob l'accetta, non avviene un vero e proprio pagamento sulla rete ethereum, ma le due parti stipulano un accordo che firmano e che non possono rescindere. In fase di chiusura del contratto Alice o Bob potranno presentare l'ultima propose concordata e ritirare quanto gli spetta. Entrambe le parti possono chiudere il canale in qualsiasi momento (basta che il canale si trovi in stato *ESTABLISHED*). Vediamo in particolare come avviene la chiusura. Ipotizziamo che Alice voglia chiudere il canale e ritirare dunque 0.9 ether. Alice eseguirà la procedura close dello smart contract. I parametri formali della procedura close sono:

- seq
- balanceA
- balanceB
- sig: propose corrente firmata da Bob

Il contratto verificherà la propose firmata da Bob e presentata da Alice e in caso positivo aggiornerà lo stato del canale in *CLOSED*. Quando Alice presenta una propose valida e porta lo stato del canale in *CLOSED* non ritira ancora i suoi ether, ma inizializza un timer. Questo timer ha la durata di `gracePeriod`, la variabile presentata in fase di apertura del canale.

Ritirare denaro da un canale

Come detto nel paragrafo precedente la close di un payment channel porta il canale in uno stato di *CLOSED* e inizializza un timer di durata pari a `gracePeriod`. Quando un canale si trova in *CLOSED* e il timer è scaduto, entrambe le parti possono ritirare quanto riportato nella propose riportata in

chiusura. Il ritiro del denaro avviene mediante una procedura denominata *withdraw*.

Argue di una propose

Prima di andare in chiusura lo stato corrente del payment channel è quello illustrato in figura 2.3.

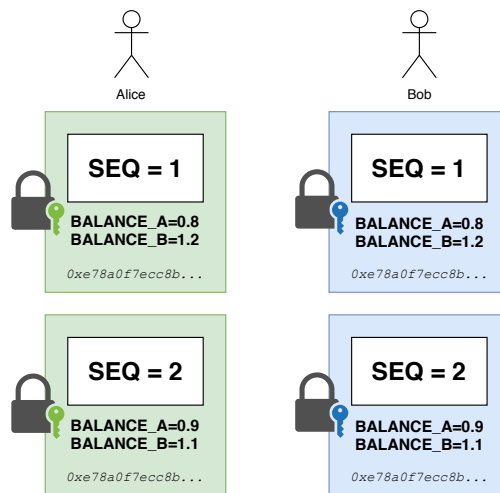


Figura 2.3: Stato corrente del payment channel

Ovvero Alice e Bob hanno concordato rispettivamente 2 proposte. Nella prima proposta Alice ha inviato un pagamento di 0.2 ether a Bob e nella seconda Bob ha inviato un pagamento di 0.1 ether ad Alice. A questo punto se Bob volesse chiudere il canale e fosse onesto, dovrebbe presentare in chiusura la proposta con $\text{seq}=2$. Ipotizziamo però che Bob sia malevolo e presenti in chiusura la proposta con $\text{seq}=1$; dal suo punto di vista questa proposta è più vantaggiosa, in quanto gli vedrebbe accreditati 0.1 ether in più. Per come è congegnata la close dello smart contract presentato nessuno impedisce a Bob di presentare questa proposta e portare lo stato del canale in *CLOSED*.

Questo chiaramente non va bene, l'unica propose valida è l'ultima proposta. A questo punto si propone un meccanismo di arguing per contrastare questa tipologia di attacco. In particolare, quando una controparte presenta una propose in chiusura, essa produce un evento pubblico, che indica il sequence number della propose. Quando Alice riceve un evento di close relativo a un suo canale di interesse, deve controllare il sequence number della propose presentata e verificare che coincida con il sequence number dell'ultima propose concordata. In caso questo non fosse vero, Alice deve presentare tramite una procedura dello smart contract denominata *argue*, l'ultima propose valida. I parametri della procedura *argue* sono gli stessi della procedura *close*, il comportamento chiaramente è diverso. La procedura *argue* infatti verifica che la propose presentata sia valida e che il sequence number sia maggiore rispetto a quello dell'ultima propose presentata. Nel caso in cui questo fosse vero, il contratto punirebbe la controparte malevola inviando tutti i fondi del payment channel ad Alice, ovvero a chi ha richiamato *argue*. La procedura di *argue* può essere effettuata solo quando il canale è in *CLOSED* e il timer di durata pari al *gracePeriod* non è scaduto. Questo meccanismo permette di essere certi che le due parti siano oneste e presentino sempre e solo l'ultima propose valida.

Free-option, controparte passiva e challenge close

Un altro tipo di attacco al quale potrebbe essere soggetto il payment channel così proposto è il seguente.

seq	balance_a	balance_b	firma
1	0.8	1.2	A/B

seq	balance_a	balance_b	firma
2	0.9	1.1	A

Come è possibile notare sono presenti due proposte. La prima proposta è stata firmata sia da A che da B. La seconda proposta è stata proposta e firmata da Alice, che rimane in attesa della risposta di Bob. Questa situazione è particolarmente pericolosa per due motivi:

1. Bob è in una condizione più vantaggiosa, infatti ha la possibilità di presentare in chiusura due proposte, sia quella con seq=1 che quella con seq=2. Questo è possibile in quanto Alice non possiede ancora la proposta con seq=2 firmata da Bob e non può sfruttare la procedura *argue*.
2. Alice non può chiudere il canale fin tanto che Bob non risponde; se Alice dovesse decidere di chiudere il canale, dovrebbe presentare la proposta con seq=1; in quel caso Bob potrebbe richiamare la procedura *argue* e presentando la proposta con seq=2 e ponendo ingiustamente la controparte.

Per quanto riguarda il primo problema, ovvero la free-option, è vero che Bob ha questa scelta, ma è anche vero che le proposte rappresentano dei pagamenti, quindi anche se Bob presentasse in chiusura seq=1 andrebbe contro i suoi interessi, in quanto si priverebbe di 0.1 ether. Riguarda lo stato di stallo in cui si trova Alice, si propone un approccio basato su sfida. In particolare è possibile prevedere una nuova procedura dello smart contract denominata *challengeClose*. Quando Alice non riceve risposta da Bob e intende chiudere il canale, invece di presentare una proposta più vecchia rischiando di essere punita, sfida Bob a chiudere. Nel momento in cui questa funzione viene richiamata, lo stato del canale viene aggiornato in *CHALLENGED* e inoltre viene inizializzato un timer di durata **gracePeriod**. Fin tanto che lo stato

del canale è *CHALLENGED* e il timer non è scaduto, Bob può presentare in chiusura una propose. Quando il timer scade, Alice può richiamare il metodo *withdraw* e ritirare tutti i fondi del payment channel. Per chiarezza riportiamo in figura 2.4, un automa temporizzato che descrive i cambiamenti di stato dello smart contract di cui ci siamo serviti.

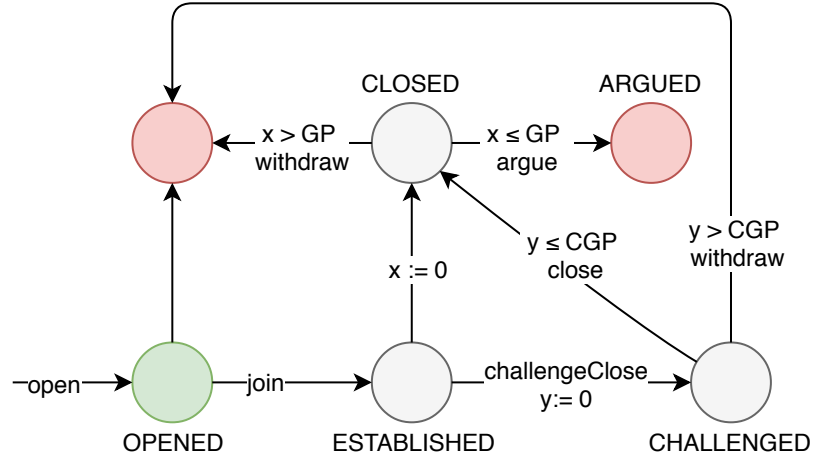


Figura 2.4: Primo automa temporizzato

Ritorniamo alla situazione precedentemente descritta, ovvero quella in cui Alice ha inviato la propose con $\text{seq}=2$ a Bob e Bob si trova in una situazione di vantaggio. Come abbiamo già detto, Bob potrebbe irrazionalmente non controfirmare più alcuna propose di Alice, lasciando quest'ultima in uno stato di stallo. Abbiamo detto di aver superato questo problema introducendo la *challengeClose* di cui Alice può servirsi per sbloccare la situazione. In questa situazione però cosa accadrebbe se fosse Bob a richiamare la *challengeClose*? Alice sarebbe costretta a chiudere e presentare una propose con un vecchio stato e a quel punto Bob potrebbe richiamare la procedura *argue* e punire ingiustamente Alice. Questo problema è facilmente superabile raffinando la procedura di *argue*. Attualmente essa verifica che sia stata presentata una propose con numero di sequenza maggiore rispetto a quella presentata per punire la controparte; un approccio diverso potrebbe essere questo: se la

propose presentata ha un numero di sequenza maggiore almeno di 2 allora la controparte viene punita, in caso contrario semplicemente viene aggiornata l'ultima propose con quella presentata.

Un approccio non punitivo

La combinazione di *challengeClose* e *argue* permette di parare entrambe le controparti da vari attacchi di cui abbiamo discusso nei precedenti paragrafi; questo ci ha portati ad aumentare sensibilmente la complessità dello smart contract, introducendo due nuove funzioni e due nuovi stati (*ARGUED* e *CHALLENGED*). In questo paragrafo presentiamo una soluzione alternativa, che non si basa su un approccio punitivo e che ci permette di ridurre drasticamente la complessità dello smart contract; questa soluzione prevede di eliminare le procedure *argue*, *challengeClose* e i relativi stati, modificando però il comportamento della *close*. Nel approccio precedente la *close* era una procedura che poteva essere richiamata un'unica volta; quanto affermato con la *close* poteva essere discusso con la procedura *argue* entro lo scadere di un timer. In questa soluzione alternativa invece, la *close* può essere richiamata più volte prima dello scadere del timer; in particolare può essere utilizzata fin tanto che viene portata in chiusura una propose valida e con numero seq maggiore rispetto all'ultima propose presentata, vedi figura 2.5. In questo modo se una controparte malevola propone una propose vecchia, la controparte vittima può sempre presentarne una più nuova andando a sovrascrivere la precedente. Per quanto riguarda la fase di stallo descritta nel precedente paragrafo, questa non si verifica, in quanto chiunque può sempre portare in chiusura una propose valida, senza dover temere di essere eventualmente punito dalla controparte.

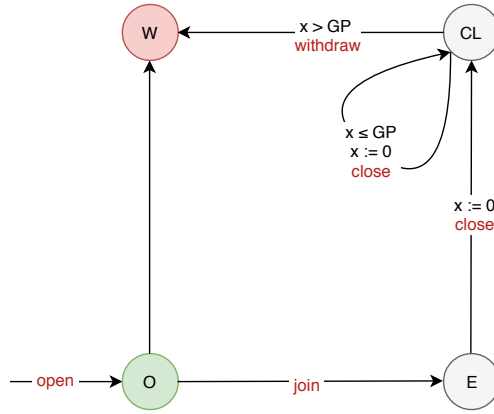


Figura 2.5: Secondo automa temporizzato

Inextinguishable Payment Channel

Il tipo di payment channel che abbiamo descritto fino ad ora permette a due entità di scambiare del valore; questo valore però può essere effettivamente ritirato (off-chain) solo chiudendo il payment channel. Inoltre spesso i canali sono sbilanciati, ovvero un'entità spende più di quanto riceve in cambio e questo può portare ad una situazione in cui una delle due entità abbia 0 ether; se questa stessa entità con 0 ether volesse effettuare un ulteriore nuovo pagamento, sarebbe costretta a dover instaurare un nuovo payment channel, perdendo tempo e denaro in fee per il deploy di un nuovo canale e l'operazione di apertura associata ad esso. In questo paragrafo proponiamo un payment channel da quale possa essere effettuato un *hotRefill* e un *hotWithdraw*, ovvero delle procedure off-chain che ci permettano di ricaricare un canale o ritirare dei soldi, senza però essere costretti a chiudere il payment channel.

Hot withdraw

Ipotizziamo di avere un canale *ESTABLISHED* tra Alice e Bob, il cui stato è sintetizzato di seguito:

Alice	Bob	Seq
1	1	
0.2	1.8	seq=1

Ora Bob vuole ritirare 0.8 ether e comunica questa sua volontà ad Alice. Alice in cambio invia un token a Bob di questo tipo:

`t = [hash(seq=1, amount=0.8)]_a`

Il token non è altro che l'hash della concatenazione del numero di sequenza dell'ultima propose firmata da entrambe le controparti e il quantitativo di ether che Bob intende ritirare, il tutto firmato con la chiave privata di Alice. A questo punto si introduce una nuova procedura all'interno dello smart contract la cui interfaccia è di seguito esposta:

`hotWithdraw(seq, amount, t)`

A questo punto Bob esegue in catena questo metodo, passando come parametri rispettivamente:

- seq: il numero di sequenza codificato nel token precedentemente realizzato
- amount: il quantitativo di ether che Bob intende ritirare
- t: il token firmato da Alice

La procedura verifica la validità del token e in caso positivo sposta il quantitativo di ether concordato nel balance off-chain di Bob. Le successive proposte che Alice e Bob concorderanno, dovranno prendere in considerazione il fatto

che è stato rilasciato un token per effettuare questo *hotWithdraw* e che quindi il bilancio on-chain di Bob è diminuito.

Per chiarezza questa propose non sarà valida:

Alice	Bob
0.3	1.7

Questa invece sarà valida:

Alice	Bob
0.3	0.9

Double spending di un token

Per come è descritto l'uso del token e dello smart contract, nessuno impedisce a Bob di eseguire più di una volta la procedura *hotWithdraw* ritirando più volte 0.8 ether pur non avendo l'autorizzazione da parte di Alice; chiaramente questo non va bene. Per evitare il fenomeno del double spending occorre memorizzare all'interno dello smart contract l'uso di questo token. In particolare si propone di associare a ciascuna controparte una mappa del tipo:

```
mapping (uint256 => uint256) proofOfDetachment;
```

Dove la chiave corrisponde al numero di sequenza al quale si riferisce il particolare token che si sta utilizzando e il valore è relativo all'amount che si è ritirato. A questo punto *hotWithdraw* oltre a verificare la validità del token, verificherà che non sia stato già utilizzato.

```
require(currentCounterpart.proofOfDetachment[seq] == 0);  
currentCounterpart.proofOfDetachment[seq] = amount;
```

Token speso in ritardo

Un possibile problema introdotto da questo modello è il seguente: Bob riceve il token, ma non lo spende; a questo punto presenta sulla blockchain una propose con seq=1 e prima che Alice effettui il withdraw, fa il withdraw e poi spende il token. In questo modo Bob presenta legittimamente una propose in cui non gli è stato ancora stato sottratto il token. Per gestire il problema, si limita l'uso della procedura *hotWithdraw* solo in stato *ESTABLISHED*.

[1] Ralph C Merkle. 1989. A certified digital signature. In *Conference on the theory and application of cryptology*, 218–238.

[2] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[3] Roberto Tamassia. 2003. Authenticated data structures. In *European symposium on algorithms*, 2–5.