



UNIVERSITÀ DEGLI STUDI ROMA TRE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tesi Di Laurea

Analisi, progettazione e prove sperimentali di
un FulgurHub in TypeScript

Laureando

Federico Ginosa

Matricola 457026

Relatore

Alberto Paoluzzi

Correlatore

Federico Spini

Anno Accademico 2017-2018

Ad Ada Lovelace

Indice

1	Introduzione	9
2	Background	13
2.1	Blockchain	13
2.2	State channel	15
2.2.1	Payment channel	16
2.2.2	Inextinguishable payment channel	19
2.3	Fulgur Hub	21
2.3.1	Motivazioni	21
2.3.2	Caratteristiche	21
2.3.3	Lavori correlati	22
3	Analisi	25
3.1	Obiettivi	25
3.2	Descrizione generale dell'architettura	26
3.3	Casi d'uso	27
3.3.1	Sottoscrizione di un FulgurHub	28
3.3.2	Pagamento OnChain-OnChain	29
3.3.3	Pagamento OffChain-OffChain	30
3.3.4	Pagamento OffChain-OnChain	33
3.3.5	Pagamento OnChain-OffChain	33

3.3.6	Prelievo a caldo	35
3.3.7	Ricarica a caldo	36
3.3.8	Chiusura di un canale	36
3.3.9	Riscossione di un pending token	37
4	Progettazione e sviluppo	39
4.1	Smart contract	39
4.1.1	Responsabilità e requisiti	39
4.1.2	Motivazioni tecnologiche	40
4.1.3	Dettagli implementativi	42
4.2	Client	43
4.2.1	Responsabilità e requisiti	43
4.2.2	Motivazioni tecnologiche	43
4.2.3	Dettagli implementativi	45
4.3	Hub	49
4.3.1	Responsabilità e requisiti	49
4.3.2	Motivazioni tecnologiche	50
4.3.3	Dettagli implementativi	51
5	Prove sperimentali	53
5.1	Gli obiettivi	53
5.2	L'approccio adottato	54
5.3	Throughput del client	56
5.4	Throughput dell'hub	59
5.5	Profiling	62
5.6	Considerazioni	63
6	Conclusioni e sviluppi futuri	67

Elenco delle tabelle

1	Struttura di una propose	18
2	Struttura di un token	20
3	Campi propose aggiuntivi in un IPC	20
4	Throughput client al variare della RAM.	56
5	Throughput client al variare della latenza.	58
6	Throughput hub al variare della RAM.	59
7	Throughput hub al variare della latenza.	61
8	Throughput FulgurHub a confronto con le principali blockchain.	63

Elenco delle figure

1	Deploy on-chain dello smart contract di un payment channel. .	17
2	Apertura e deposito fondi on-chain in un payment channel. . .	17
3	Join e deposito fondi on-chain in un payment channel.	18
4	Architettura hub-and-spoke di FulgurHub	26
5	Sottoscrizione di un FulgurHub	29
6	Pagamento OnChain-OnChain in FulgurHub.	30
7	Pagamento OffChain-OffChain in FulgurHub.	32
8	Pagamento OffChain-OnChain in FulgurHub.	34
9	Pagamento OnChain-OffChain in FulgurHub.	35
10	Throughput client al variare della RAM	57
11	Barchart throughput client al variare della latenza.	58
12	Barchart throughput hub al variare della RAM	60
13	Barchart throughput hub al variare della latenza.	61
14	Pagamento OffChain-OffChain in FulgurHub.	62
15	Profiling pagamento OffChain-OffChain	62
16	FulgurHub a confronto con tecnologie alternative	64

Capitolo 1

Introduzione

2008 whitepaper Bitcoin Nel 2008 viene pubblicato il whitepaper di Bitcoin, una proposta di soluzione al problema della doppia spesa basata sull'utilizzo di una rete peer-to-peer [9].

2009 pubblicazione protocollo Il 2009 vede la nascita della prima cryptovaluta, bitcoin, con la pubblicazione dell'implementazione del protocollo Bitcoin. Gli standard variano, ma sembra essersi formato un consenso nel riferirsi con Bitcoin maiuscolo al protocollo e con bitcoin minuscolo alla moneta in se [12].

Bitcoin 10 anni dopo Negli anni successivi decine di protocolli alternativi a Bitcoin sono fioriti. Le cryptovalute da argomento di nicchia hanno visto una costante crescita di adozione. Dal 2009 fino a oggi il numero di transazioni quotidiane è cresciuto più che linearmente, raggiungendo il suo attuale picco storico nel dicembre del 2017, con più di 450K transazioni in un giorno [13].

Limiti architetturali Questo crescente interesse nei confronti delle cryptovalute si scontra però con i limiti architetturali relativi alla scalabilità della

blockchain. Con gli attuali parametri di blocksize e blockinterval (1 megabyte e 10 minuti), il throughput massimo è compreso tra le 3 e le 7 tps (transazioni per secondo). Portando questi due parametri all'estremo, 1 megabyte e 1 minuto, si riuscirebbe a decuplicare il throughput, senza sacrificare il protocollo in termini di sicurezza [4]. Sebbene 30 - 70 tps rappresenterebbero un fondamentale miglioramento tecnologico di Bitcoin, il throughput raggiunto non sarebbe comunque confrontabile con quello di sistemi centralizzati analoghi come il circuito VISA, con le sue 56K tps [14].

Soluzioni Diverse sono le soluzioni proposte per risolvere i problemi di scalabilità della blockchain e possono essere suddivise in tre categorie:

- **Algoritmo di consenso** Alla base del whitepaper di Satoshi Nakamoto c'è il Proof Of Work. Modificando il meccanismo alla base della ricerca del consenso è possibile migliorare la scalabilità della blockchain. Ad oggi diverse sono le alternative proposte [6], [1], [7].
- **Sharding** Questo concetto non è nuovo nel mondo dei database. L'idea è quella di suddividere la blockchain in più parti. La ricerca del consenso avviene in ciascuno di queste parti. Anche da questo punto di vista diversi sono i lavori e le proposte [8], [15].
- **Off-chain** Layers è un famoso pattern architetturale. Un forte impiego di questo pattern è stato fatto nell'ambito del networking, vedi pila ISO/OSI. La scalabilità off-chain si realizza costruendo un secondo layer sopra alla blockchain, che permetta di ereditare le sue caratteristiche (sicurezza e distribuzione), aggiungendone delle altre, come la scalabilità.

Questi tre diversi approcci alla scalabilità della blockchain non sono in contrasto l'uno con l'altro, ma anzi possono essere applicati assieme in maniera sinergica. Nel lavoro di questa tesi ho approfondito l'ultima categoria, la

scalabilità off-chain. In particolare mi sono occupato delle seguenti attività:

- Analisi dello stato dell'arte relativa a soluzioni di scalabilità off-chain
- Realizzazione di un canale di pagamento inestinguibile (IPC)
- Analisi, progettazione e sviluppo di FulgurHub
- Prove sperimentali di FulgurHub

In questa tesi il capitolo due tratta il background necessario, in particolare si approfondisce il design di un payment channel e si introduce l'architettura di FulgurHub. Nel terzo capitolo si effettua l'analisi nel dettaglio di FulgurHub. Nel quarto capitolo si descrivono le fasi di progettazione e sviluppo di FulgurHub. Nel quinto capitolo si mostrano le prove sperimentali relative a quanto è stato implementato e si discutono i risultati in termini di performance e scalabilità.

Capitolo 2

Background

Questo capitolo descrive il background necessario. In particolare in sezione 2.1 si discute la blockchain: come funziona, i suoi casi d'uso e i suoi limiti. In sezione 2.2 si descrivono gli state channel, facendo un affondo sui payment channel e inextinguishable payment channel. Infine in sezione 2.3 si introduce il protocollo FulgurHub: le sue motivazione, le caratteristiche e i lavori correlati.

2.1 Blockchain

Il problema La blockchain risolve il problema del double spending in un sistema peer-to-peer completamente decentralizzato [9]. Questo permette di memorizzare in maniera immutabile dei pagamenti in un registro pubblico, avendo la certezza che nessuno possa spendere più volte lo stesso token.

Il caso d'uso Il caso d'uso tipico della blockchain è l'invio e la ricezione di pagamenti. La transazione rappresenta un pagamento. Essa può essere

immaginata come un arco che unisce due nodi. Il nodo iniziale rappresenta il pagante, il nodo finale il pagato. Tutte queste transazioni vengono memorizzate su un registro pubblico detto ledger. Sebbene le transazioni siano pubbliche, l'identità reale è disaccoppiata dagli indirizzi pubblici di paganti e pagati.

Cos'è la blockchain La blockchain è una lista concatenata di blocchi. Ciascun blocco contiene: l'hash del precedente blocco, il merkle root relativo alla lista di transazioni associate al blocco corrente e un nonce. In Bitcoin un nuovo blocco viene aggiunto ogni dieci minuti e il merkle root rappresenta una prova succinta di una lista di transazioni di dimensione minore o uguale a 1 megabyte.

Come funziona la PoW I blocchi vengono aggiunti dai miner. I miner sono dei nodi della rete che si occupano di trovare un nonce che faccia sì che l'hash del blocco corrente abbia un numero di zeri iniziali pari a D . Questo valore D rappresenta la difficoltà corrente di mining della rete. La difficoltà è autoregolata dal protocollo e aumenta o diminuisce a seconda del tempo necessario per minare i precedenti blocchi. Un miner che riesce a presentare un nonce e un blocco valido ottiene in cambio le fee delle singole transazioni e una coinbase.

Cos'è uno smart contract Inviare un pagamento in Bitcoin significa sbloccare uno o più UTXO (Unspent Transaction Output). Sbloccare un UTXO significa presentare una prova crittografica della proprietà di un certo token. La verifica della prova crittografica viene effettuata da tutti i nodi della rete eseguendo un ASFND (automa a stati finiti non deterministico). Il protocollo Bitcoin permette di implementare e deployare sulla rete degli automi anche più complessi. Script è il linguaggio di programmazione stack-based non Turing-completo che permette di descrivere questi automi in Bitcoin.

Quando la complessità degli automi aumenta, si parla di smart contract, ovvero di contratti che permettono lo sblocco di fondi previa verifica di un insieme complesso di regole.

Smart contract Turing-completi Sebbene abbia senso parlare di smart contract in Bitcoin, l'uso del termine in questo contesto è stato introdotto solo nel 2014, con la pubblicazione del whitepaper di Ethereum [3]. Ethereum è un protocollo che eredita gran parte delle caratteristiche di Bitcoin e in più introduce la EVM (Ethereum Virtual Machine) la macchina virtuale che esegue gli smart contract. Gli smart contract in Ethereum vengono descritti in Solidity, un linguaggio di programmazione C-like Turing-completo. La turing completezza permette di descrivere un più ampio spettro di regole.

Scalabilità off-chain Nel Capitolo 1 sono stati introdotti i limiti architeturali della blockchain e le tre categorie di approcci risolutivi. La scalabilità off-chain è una delle tipologie di soluzioni atte a superare i limiti di scalabilità della blockchain. Questo approccio riduce sensibilmente le interazioni necessarie sulla blockchain, spostandole fuori di essa, senza compromettere le proprietà di sicurezza.

2.2 State channel

Gli state channel permettono a due parti di modificare in maniera sicura porzioni della blockchain, dette depositi di stato. Questi depositi di stato sono memorizzati all'interno di indirizzi multisignature o di smart contract (a seconda dell'implementazione). Le parti modificano lo stato dello state channel scambiando messaggi off-chain. Questi messaggi descrivono un aggiornamento dello stato, per esempio la prossima mossa in una partita di tris [16].

2.2.1 Payment channel

Un payment channel è una particolare tipologia di state channel. I messaggi scambiati off-chain rappresentano dei pagamenti, ovvero l'aggiornamento del bilancio delle parti. Instaurare un payment channel richiede una sola operazione on-chain da ciascuna parte. L'operazione on-chain viene eseguita su uno smart contract dedicato al singolo payment channel. Questa unica operazione on-chain abilita un numero potenzialmente illimitato di pagamenti off-chain, nei limiti del balance iniziale delle parti. I messaggi off-chain possono essere scambiati mediante qualunque mezzo, comunemente una connessione http. Un payment channel permette dunque di spostare i problemi di scalabilità dalla blockchain a un server http, ma la letteratura riguardo a come far scalare quest'ultimo è consolidata.

Architettura Un payment channel permette di effettuare un numero illimitato di transazioni off-chain tra due parti. Ciascuna parte deve mettere a disposizione un server http che permetta l'invio e la ricezione di pagamenti. Una delle due parti deploys lo smart contract associato e apre il canale. In un secondo momento la controparte effettua il join del canale, stabilendone la definitiva apertura. In questa progettazione si è presa come riferimento la blockchain di Ethereum.

Deploy Il deploy è la prima fase di inizializzazione. Alice deploys lo smart contract del relativo canale. L'operazione di deployment è richiesta per ciascun singolo payment channel. Questa fase permette di ottenere l'indirizzo di un contratto, che nelle successive fasi verrà adottato per richiamare le operazioni on-chain che si intende richiamare. In questa fase lo stato del payment channel è detta *INIT*.

Apertura Alice apre il canale e blocca un quantitativo arbitrario di fondi all'interno dello smart contract. Questi fondi rappresentano il bilancio iniziale



Figura 1: Deploy on-chain dello smart contract di un payment channel.

di Alice. Si fa notare come la fase di deploy e di apertura possano essere svolte con un'unica operazione on-chain. Oltre a depositare i fondi, Alice con questa operazione porta in catena il suo indirizzo ip e l'indirizzo ethereum di Berto. Terminata la procedura lo stato del canale diventa *OPENED*.

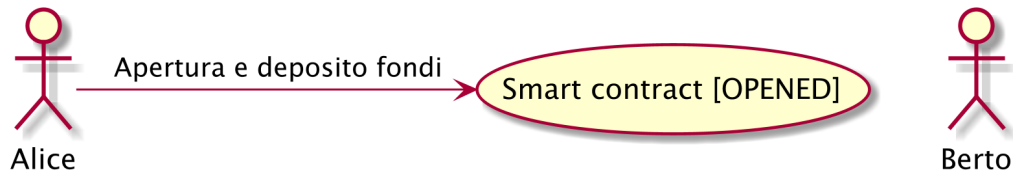


Figura 2: Apertura e deposito fondi on-chain in un payment channel.

Join In un secondo momento Berto effettua il join del canale di pagamento aperto da Alice. Anche questa operazione viene effettuata on-chain. Berto deposita i fondi che corrisponderanno al suo bilancio iniziale e porta in catena il proprio indirizzo ip. Con questa operazione il canale è definitivamente stabilito e lo stato passa da *OPENED* a *ESTABLISHED*.

Schema propose/accept I pagamenti off-chain avvengono mediante lo schema propose/accept. Alice (o Berto) propone un aggiornamento dello stato del canale firmando questa proposta con la propria chiave privata. Berto riceve la proposta, ne verifica la validità ed eventualmente l'accetta inviando la proposta controfirmata ad Alice. Il pagamento è avvenuto, senza la necessità di ulteriori tempi di attesa o conferme.

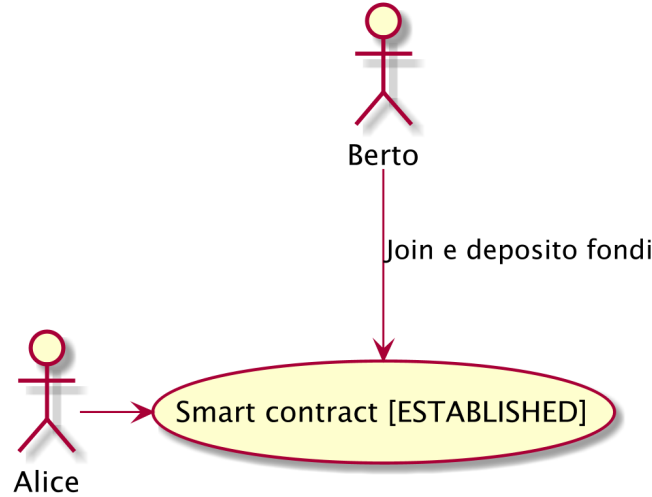


Figura 3: Join e deposito fondi on-chain in un payment channel.

Gli endpoint pubblici Ciascuna controparte di un payment channel mette a disposizione un server http. Gli endpoint pubblici sono detti `/propose` e `/accept`. L'endpoint `/propose` permette di ricevere una proposta di aggiornamento di bilancio. L'endpoint `/accept` permette di ricevere una proposta precedentemente inviata, controfirmata in Tabella 1.

Tabella 1: Struttura di una propose

Campo	Descrizione
seq	Il numero di sequenza
balance _a	Il balance di chi ha aperto il canale
balance _b	Il balance di chi ha effettuato il join del canale
sign	La firma della propose

Richiesta di chiusura Chiudere un canale significa aggiornare il balance on-chain delle parti in modo tale che corrisponda a quello dell'ultima propose comunemente accordata. La prima fase di questo processo è detta richiesta di chiusura. In particolare si porta in catena l'ultima propose comunemente firmata. In questo modo lo stato del canale passa da *ESTABLISHED* a *CLOSED*. La richiesta di chiusura può essere effettuata da Alice o da Berto.

Finalizzazione della chiusura L'operazione di finalizzazione della chiusura viene effettuata da tutte e due le parti. Essa corrisponde al ritiro on-chain

propose significa portare in catena una propose firmata da Alice con numero di sequenza maggiore rispetto a quella presentata. Nel caso in cui la discussione abbia successo, Alice viene punita; la punizione consiste nel trasferimento di tutti i suoi fondi a Berto.

Il problema della free-option Quando Alice invia una propose a Berto senza ricevere la controfirma, Berto ha il vantaggio di poter scegliere di chiudere il canale con due propose, la penultima o l'ultima. Inviare una propose coincide con inviare un pagamento, quindi sebbene Berto possa decidere di presentare in catena la penultima propose, questa descriverà uno stato per lui più svantaggioso.

2.2.2 Inextinguishable payment channel

I payment channel permettono di trasferire un volume di coin limitato. Il valore trasferibile è fissato alla somma del balance di Alice e di Berto. Spesso questi canali sono sbilanciati, ovvero una delle due controparti effettua più pagamenti dell'altra. Un canale sbilanciato nel tempo prosciuga il balance di una delle due parti, rendendo il payment channel inutilizzabile. L'unica soluzione consiste nel chiudere il payment channel corrente e aprirne un nuovo, caricando nuovi fondi. Questa soluzione richiede delle operazioni on-chain onerose (deploy, apertura e join). Gli IPC (inextinguishable payment channel) superano questo problema, proponendo dei canali di pagamento che permettono di caricare e scaricare a caldo i balance.

Schema detach/attach Questo schema rappresenta un'estensione dello schema propose/accept. Esso permette di staccare un token off-chain e di attaccarlo on-chain. Un token rappresenta un certo quantitativo del bilancio. La struttura di un token è illustrato in Tabella 2.

Tabella 2: Struttura di un token

Campo	Descrizione
seq	Numero di sequenza del token
value	Valore del token
sign	Firma del token

Anche la struttura dati relativa a una propose viene estesa. I campi aggiunti sono illustrati in Tabella 3.

Tabella 3: Campi propose aggiuntivi in un IPC

Campo	Descrizione
hash token	L'hash relativo al token
type of propose	attach/detach

Ritiro a caldo Alice vuole ritirare a caldo 0.5 eth; effettua il detach off-chain di un token; invia a Berto una propose contenente un token di 0.5 eth che scala da balance_a . Berto risponde con propose e token firmati. Il token firmato rappresenta la PoD (Proof of Detachment). Alice effettua l'attach in catena della PoD e ritira a caldo 0.5 eth.

Ricarica a caldo Alice vuole ricaricare a caldo il canale di 0.5 eth; effettua l'attach on-chain di un token depositando nello smart contract 0.5 eth. Questa operazione on-chain viene notificata a Berto; tale notifica rappresenta la PoA (Proof of Attachment). A questo punto Alice invia a Berto una propose in cui effettua l'attach di un token di pari valore e incrementa di 0.5 eth il proprio balance. Berto risponde con la propose firmata, confermando la

ricarica a caldo.

Double spending di un token Quando Alice ritira a caldo presentando un token, lo smart contract associa una PoA (Proof of Attachment) relativa al numero di sequenza del token corrente. Questo permette allo smart contract di non accettare token già spesi.

2.3 Fulgur Hub

2.3.1 Motivazioni

Sebbene i canali di pagamento e gli IPC rappresentino un punto di svolta per la scalabilità off-chain, essi sono degli strumenti rudimentali e con una esperienza utente limitata. Non è infatti pensabile dover inizializzare un canale di pagamento con ciascun individuo con cui si voglia instaurare un rapporto economico. Fulgur Hub nasce dalla necessità di migliorare l'esperienza utente degli IPC e di potenziare alcune delle loro caratteristiche [11].

2.3.2 Caratteristiche

Transazioni istantanee ed economiche In Bitcoin la conferma di una transazione richiede 60 minuti. In un IPC basta lo scambio di due messaggi su protocollo http per effettuare e confermare un pagamento. Questo apre nuove prospettive economiche, ad esempio una macchina in cloud potrebbe essere pagata dopo ogni secondo di utilizzo o si potrebbe vedere il proprio stipendio accreditato dopo ogni minuto di lavoro effettuato; FulgurHub abilita questi casi d'uso.

Transazioni tra più di due entità In un IPC i pagamenti possono essere effettuati tra due partecipanti. FulgurHub consente di effettuare pagamenti tra gli N utenti registrati ad un FulgurHub.

Pagamenti ibridi FulgurHub permette di effettuare dei pagamenti ibridi. Ciascun utente infatti possiede due balance, uno on-chain e uno off-chain e può decidere di spostare dei fondi da uno stato off-chain a uno stato on-chain e viceversa. Inoltre abilita i pagamenti tra utenti di due FulgurHub diversi.

Autogestito In un IPC l'utente deve costantemente verificare e accettare la validità di un pagamento, oltre a contestare eventuali comportamenti scorretti della controparte. In FulgurHub i server degli utenti e dell'hub si occupano di gestire autonomamente diversi scenari, limitando allo stretto necessario l'intervento manuale.

Pagamenti trustless Caratteristica essenziale è che un utente onesto abbia la certezza di non perdere i propri fondi. In sistemi centralizzati questa garanzia esiste perché ci si fida di un'entità centrale, come una banca o un servizio di e-payment. In un FulgurHub questa garanzia è data dal protocollo stesso, in questo senso i pagamenti sono trustless.

Passività e anonimato FulgurHub è un sistema passivo; questo significa che l'hub non contatta mai gli utenti, ma solo quest'ultimi contattano l'hub. Questo permette agli utenti di non dover fornire il loro indirizzo ip reale e quindi di poter effettuare pagamenti anche dietro una rete come Tor.

2.3.3 Lavori correlati

Tumblebit Si tratta di un hub di pagamenti anonimo basato su Bitcoin. L'approccio di centralizzazione garantisce anonimato e pagamenti trustless.

Sfortunatamente il particolare payment channel adottato è unidirezionale e ha un tempo di vita limitato [5].

CoinBlesk Un bitcoin wallet che usa un server centrale che permette di eseguire dei pagamenti virtuali. Supporta micropagamenti istantanei, ma l'approccio non è considerabile trustless [2].

Lightning e Raiden Network Entrambi i network si basano su un grafo di payment channel bidirezionali. Un pagamento avviene in maniera analoga all'instradamento di un pacchetto su internet. Una volta trovato il percorso ottimo esso deve essere completato con successo in ciascun hop intermedio. Se un solo hop fallisce il pagamento fallisce. Questo garantisce l'atomicità dei pagamenti [10] [17]. Sebbene Lightning Network e Raiden Network siano progettati per essere decentralizzati, la realtà economica fa tendere la topologia di rete alla centralizzazione; maggiore è il numero di hop, maggiori sono le commissioni e le probabilità di insuccesso. FulgurHub è stato disegnato con questo in mente e propone una topologia hub and spoke.

Capitolo 3

Analisi

Questo capitolo descrive il processo di analisi svolto. In particolare in sezione 3.1 si discutono gli obiettivi dell'analisi. In sezione 3.2 si descrive l'architettura generale di FulgurHub. Infine in sezione 3.3 si descrivono i principali casi d'uso e la gestione di eventuali eccezioni.

3.1 Obiettivi

Dimostrazione di fattibilità Un obiettivo di questa tesi è stato dimostrare la fattibilità del protocollo FulgurHub. In particolare ci si è concentrati sulle feature principali: apertura di un canale, pagamenti OnChain-OnChain, pagamenti OffChain-OffChain, pagamenti OffChain-OnChain, pagamenti OnChain-OffChain, prelievi a caldo, ricariche a caldo, chiusura di un canale e riscossione di pending token.

Dimostrare la scalabilità architetturale Come detto in capitolo 2, le motivazioni che hanno mosso la progettazione di FulgurHub riguardano i

limiti architetturali di scalabilità della blockchain. Obiettivo di questa tesi è stato anche dimostrare la scalabilità architetturale di FulgurHub.

3.2 Descrizione generale dell'architettura

Il sistema si basa su uno smart contract. Lo smart contract gestisce il balance on-chain e il payment channel di ciascun utente. In figura 4 si mostra l'architettura hub-and-spoke in cui 4 utenti (Alice, Berto, Cecilia e Dario) operano su FulgurHub [11].

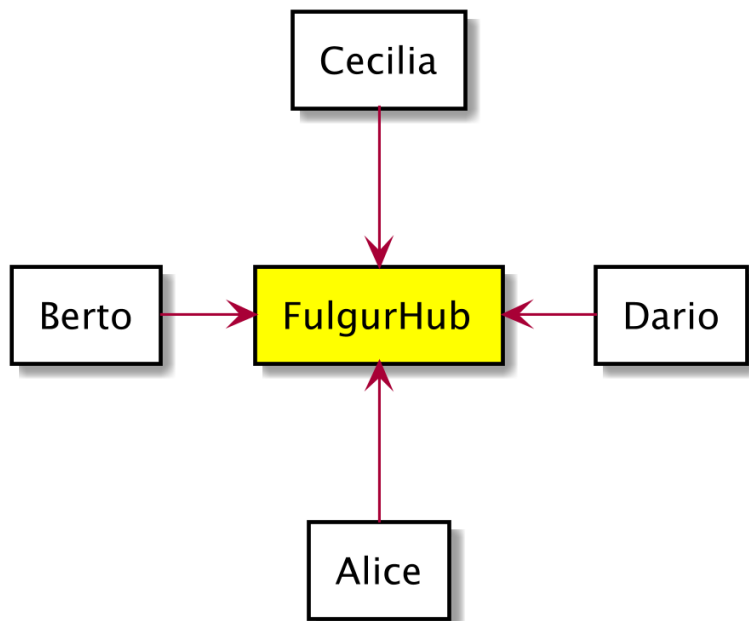


Figura 4: Architettura hub-and-spoke di FulgurHub

Hub L'hub è supportato da un modulo software che interagisce con lo smart contract. Il modulo è stateless, favorendo disponibilità e scalabilità. L'hub è passivo, questo significa che non contatta mai direttamente i client; solo i

client possono contattare l'hub. I client possono contattare l'hub mediante richieste http a uno dei suoi endpoint pubblici.

Client La relazione tra client e hub può essere descritta come una "registrazione trustless" del client al servizio di intermediazione offerto dall'hub [11]. Il client è supportato da un modulo software che interagisce con lo smart contract e l'hub. La registrazione del client coincide con l'instaurare un IPC esteso che permetta dei pagamenti ibridi, come descritto in 3.3. Un client può chiudere la registrazione dall'hub in ogni momento; in particolare deve chiudere la propria registrazione appena si verifica un comportamento anomalo dell'hub.

Smart contract Lo smart contract garantisce la relazione trustless tra i client e l'hub. In particolare lo smart contract deve essere utilizzato quando una delle parti non si comporta correttamente. Oltre a questo lo smart contract si occupa di aprire/chiedere la sottoscrizione del client e di gestire pagamenti ibridi che coinvolgano endpoint on-chain.

3.3 Casi d'uso

Strutture dati e simbolismo FulgurHub si fonda su due tipi di strutture dati, le propose e i token. Una propose ϕ_i descrive il balance off-chain di client (β_i^C) e hub (β_i^H). Le propose sono ordinate totalmente sulla base del numero di sequenza i . Un token τ_j può essere staccato (\mathbb{D}) o attaccato (\mathbb{A}) ad una propose. Inoltre una propose può essere firmata dal client ($\phi_i^{\sigma_C}$), dall'hub ($\phi_i^{\sigma_H}$) o da entrambi ($\phi_i^{\sigma_C, \sigma_H}$).

$$\phi_i^{\sigma_C, \sigma_H} = \langle \beta_i^C, \beta_i^H, \tau_j, \mathbb{D} || \mathbb{A} \rangle \quad (1)$$

Un token è identificato in maniera univoca dalla tupla (j, α_P) , dove j identifica il numero di sequenza del token e α_P l'indirizzo ethereum del pagato. Il client staccando un token può sottrarre una porzione ν_j del proprio bilancio. Un token può essere staccato dal bilancio on-chain od off-chain. Un token può essere recapitato al pagato. Il pagato per riscuotere un token deve attaccarlo off-chain (mediante una propose) od on-chain (mediante lo smart contract). Esistono due tipi di token; quelli riscuotibili on-chain (\mathbb{ON}) e quelli riscuotibili off-chain (\mathbb{OFF}). Inoltre un token può essere firmato dal client (τ^{σ_C}), dall'hub (τ^{σ_H}) o da entrambi ($\tau^{\sigma_C, \sigma_H}$). Un token può essere riscosso entro un tempo di scadenza exp .

$$\tau_{y, ID(P)}^{\sigma_C, \sigma_H} = \langle \nu_y, exp, \mathbb{ON} || \mathbb{OFF} \rangle \quad (2)$$

Una propose $\phi_i^{\sigma_C}$ con un token τ_y detached (\mathbb{D}) firmato rappresenta una ricevuta di pagamento. La ricevuta di pagamento è una prova incontrovertibile della riscossione di un token.

Per indicare il balance off-chain di un'entità k ad una propose con numero di sequenza pari a i si usa il simbolo β_i^k , mentre per indicare il balance on-chain $\overline{\beta^k}$.

L'indirizzo ethereum di un'entità k è indicato dal simbolo α_k . L'insieme di indirizzi ethereum che hanno una sottoscrizione attiva con il FulgurHub associato ad H è detto Π^H .

3.3.1 Sottoscrizione di un FulgurHub

Alice vuole sottoscrivere una registrazione su un FulgurHub. Questa attività coincide con l'apertura di un payment channel.

Precondizioni

- a) $\{\alpha^A\} \notin \Pi^H$
- b) L'hub ha deployato lo smart contract
- c) Il server dell'hub è in ascolto

Descrizione delle interazioni Un client per sottoscrivere un FulgurHub deve eseguire la funzione `subscribe` dello smart contract fornendo il proprio indirizzo ethereum α_C , il bilancio iniziale off-chain β_0^C e on-chain $\overline{\beta^C}$. Inoltre il client deve indicare il bilancio iniziale off-chain dell'hub β^H . Una volta eseguita la transazioni on-chain viene recapitata una notifica all'hub $\langle \beta_0^C, \overline{\beta^C}, \beta^H, \alpha_C \rangle$. In figura 5 viene fornito un diagramma di sequenza del caso d'uso.

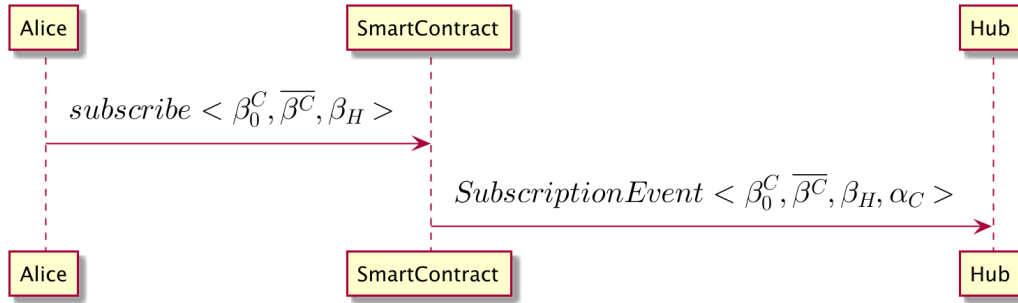


Figura 5: Sottoscrizione di un FulgurHub

3.3.2 Pagamento OnChain-OnChain

Un pagamento OnChain-OnChain sposta ν fondi dal balance on-chain di Alice $\overline{\beta^A}$ al balance on-chain di Berto $\overline{\beta^B}$. Questo pagamento viene totalmente gestito dallo smart contract e non richiede alcuna interazione con i server dei client o dell'hub.

Precondizioni

- a) $\{\alpha^A, \alpha^B\} \subseteq \Pi^H$
- b) Il balance on-chain di Alice e Berto è rispettivamente pari $\overline{\beta^A}$ e $\overline{\beta^B}$

Descrizione delle interazioni Alice esegue il metodo `transfer` dello smart contract. L'esecuzione del metodo richiede il quantitativo ν di fondi che si intende spostare e l'indirizzo ethereum α^B di Berto. Terminata l'esecuzione del metodo lo smart contract aggiorna il balance on-chain di Alice in $\overline{\beta^A} - \nu$ e quello di Berto in $\overline{\beta^B} + \nu$. Un diagramma di sequenza è disponibile in figura 6.

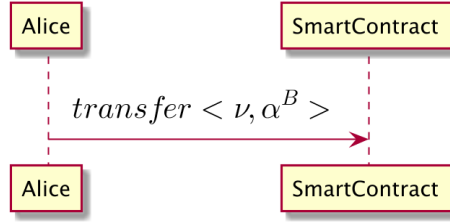


Figura 6: Pagamento OnChain-OnChain in FulgurHub.

3.3.3 Pagamento OffChain-OffChain

Un pagamento OffChain-OffChain sposta fondi dal balance off-chain di Alice β_i^A a quello di Berto β_i^B . Questo tipo di pagamento non richiede interazioni con la catena, il che lo rende economico e istantaneo.

Precondizioni

- a) $\{\alpha^A, \alpha^B\} \subseteq \Pi^H$
- b) Le ultime proposte confermate nei canali di Alice e Berto sono ϕ_i^A e ϕ_j^B .

Descrizione delle interazioni Alice costruisce, firma e invia $\phi_{i+1}^{\sigma_A}$ all'hub. L'hub risponde con la propose $\phi_{i+1}^{\sigma_A, \sigma_H}$ e il token $\tau_{y, \alpha_B}^{\sigma_A, \sigma_H}$ controfirmati.

$$\begin{aligned}\tau_{y,\alpha_B}^{\sigma_A} &= \langle \nu_y, \text{exp}, \text{OFF} \rangle \\ \phi_{i+1}^{\sigma_A} &= \langle \beta_i^A - \nu_y, \beta_i^H, \tau_{y,\alpha_B}^{\sigma_A}, \mathbb{D} \rangle\end{aligned}\tag{3}$$

$\tau_{y,\alpha_B}^{\sigma_A, \sigma_H}$ rappresenta una PoD (Proof of Detachment). Alice invia la PoD a Berto. Berto costruisce $\phi_{j+1}^{\sigma_B}$ effettuando l'attach della PoD.

$$\phi_{j+1}^{\sigma_B} = \langle \beta_i^B + \nu_y, \beta_i^H - \nu_y, \tau_{y,\alpha_B}^{\sigma_A}, \mathbb{A} \rangle\tag{4}$$

Berto invia la ricevuta di pagamento $\phi_{j+1}^{\sigma_B}$ ad Alice. Alice ora ha in mano una prova incontrovertibile del fatto che il suo token sia stato riscosso. In questa fase l'hub si è esposto di ν_i fondi sul canale di Berto; Alice deve ribilanciare questa situazione e lo fa costruendo $\phi_{i+2}^{\sigma_A}$, una nuova propose in cui attacca la PoD ricevuta da Berto.

$$\phi_{i+2}^{\sigma_A} = \langle \beta_i^B + \nu_y, \beta_i^H - \nu_y, \tau_y^{\sigma_B}, \mathbb{A} \rangle_{(\sigma_B)}\tag{5}$$

Il pagamento OffChain-OffChain è considerato concluso. In figura 7 viene fornito uno diagramma di sequenza delle interazioni.

B non invia la ricevuta di pagamento ad A Il collegamento tra Alice e Berto è opzionale. Alice infatti può contattare l'hub e richiedere la ricevuta di pagamento.

L'hub non permette di staccare un token Se l'hub non è collaborativo, Alice chiude il canale.

L'hub non permette di attaccare un token Se l'hub non è collaborativo, Berto ha la facoltà di chiudere il canale e successivamente riscuotere il pending token on-chain.

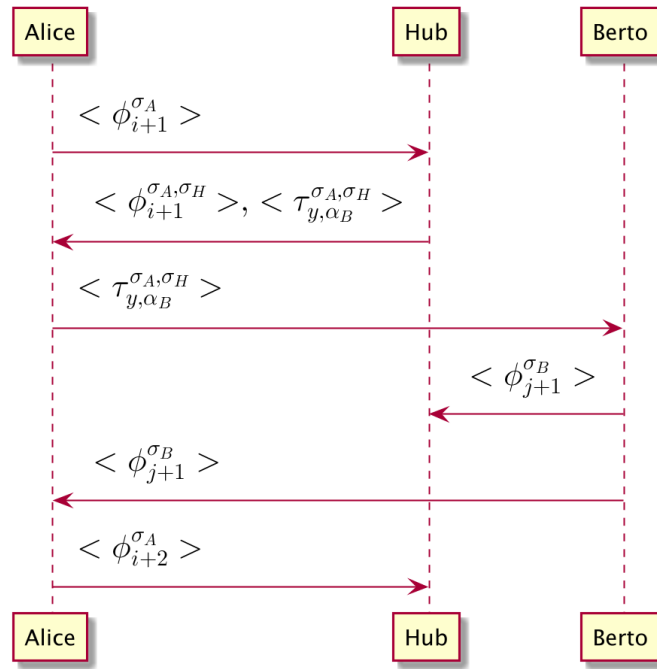


Figura 7: Pagamento OffChain-OffChain in FulgurHub.

Mancanza di cooperazione nel ricevere un pagamento Il client può cancellare il pagamento al termine della sua scadenza, ritirandolo off-chain.

3.3.4 Pagamento OffChain-OnChain

Un pagamento OffChain-OnChain consiste nel spostare fondi dal balance off-chain di Alice β_i^A al balance on-chain di Berto $\overline{\beta^B}$.

Precondizioni

- a) $\{\alpha^A, \alpha^B\} \subseteq \Pi^H$
- b) L'ultima propose confermata nel canale di Alice è ϕ_i^A .

Descrizione delle interazioni Alice costruisce, firma e invia $\phi_{i+1}^{\sigma_A}$ all'hub. L'hub risponde con la propose $\phi_{i+1}^{\sigma_A, \sigma_H}$ e il token $\tau_{y, \alpha_B}^{\sigma_A, \sigma_H}$ controfirmati.

$$\begin{aligned} \tau_{y, \alpha_B}^{\sigma_A} &= \langle \nu_y, exp, \mathbb{ON} \rangle \\ \phi_{i+1}^{\sigma_A} &= \langle \beta_i^A - \nu_y, \beta_i^H, \tau_{y, \alpha_B}^{\sigma_A}, \mathbb{D} \rangle_{(\sigma_A)} \end{aligned} \tag{6}$$

$\tau_{y, \alpha_B}^{\sigma_A, \sigma_H}$ rappresenta una PoD (Proof of Detachment). Alice invia la PoD a Berto. Berto effettua l'attach on-chain del token mediante la funzione attach dello smart contract. Lo smart contract aggiorna il balance on-chain di Berto in $\overline{\beta^B} + \nu_y$. Il pagamento è considerato concluso. In figura 8 viene fornito uno diagramma di sequenza delle interazioni.

3.3.5 Pagamento OnChain-OffChain

Un pagamento OnChain-OffChain consiste nel spostare fondi dal balance on-chain di Alice $\overline{\beta^A}$ al balance off-chain di Berto β_j^B .

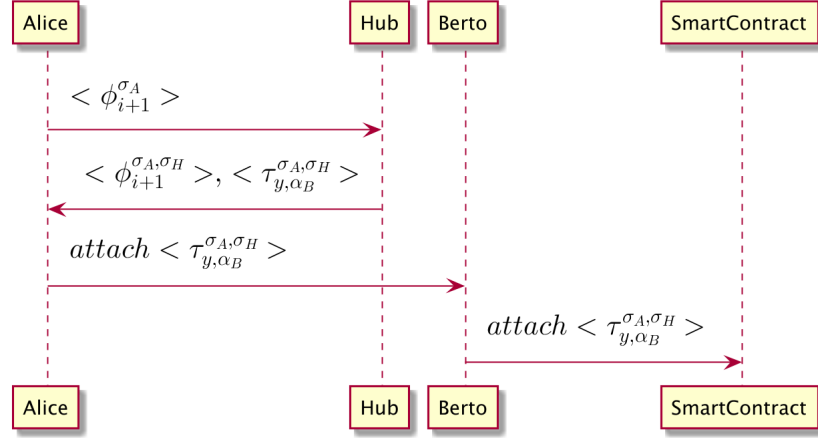


Figura 8: Pagamento OffChain-OnChain in FulgurHub.

Precondizioni

- a) $\{\alpha^A, \alpha^B\} \subseteq \Pi^H$
- b) L'ultima propose confermata nel canale di Berto è ϕ_j^B
- c) Il balance on-chain di Alice è $\overline{\beta_A}$

Descrizione delle interazioni Alice esegue la funzione detach dello smart contract fornendo l'indirizzo di Berto (α_B) e il quantitativo ν che si vuole staccare. Lo smart contract aggiorna il balance on-chain di Alice in $\overline{\beta_A} + \nu$. Terminata l'esecuzione della funzione, lo smart contract invia la relativa PoD a Berto. Berto costruisce, firma e invia $\phi_{j+1}^{\sigma_B}$ all'hub, attaccando la PoD. L'hub risponde con la propose firmata $\phi_{j+1}^{\sigma_B, \sigma_H}$. In figura 9 viene fornito uno diagramma di sequenza delle interazioni.

$$\begin{aligned} \tau_y^{\sigma_B} &= \langle \nu_y, \perp, \mathbb{ON} \rangle \\ \phi_{j+1}^{\sigma_B} &= \langle \beta_j^B - \nu_y, \beta_j^H, \tau_{y, \alpha_B}^{\sigma_B}, \mathbb{A} \rangle \end{aligned} \tag{7}$$

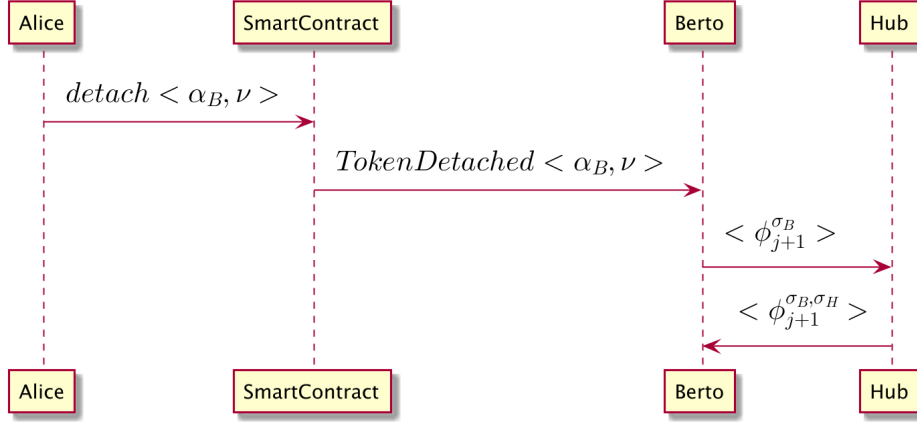


Figura 9: Pagamento OnChain-OffChain in FulgurHub.

3.3.6 Prelievo a caldo

Effettuare un prelievo a caldo significa spostare dei fondi dal balance off-chain di Alice β_i^A al balance on-chain di Alice $\overline{\beta}^A$.

Precondizioni

- a) $\{\alpha^A\} \subseteq \Pi^H$
- b) L'ultima propose confermata nel canale di Alice è ϕ_i^A
- c) Il balance on-chain di Alice è $\overline{\beta}_A$

Descrizione delle interazioni Alice costruisce, firma e invia $\phi_{i+1}^{\sigma_A}$ all'hub. L'hub risponde con la propose $\phi_{i+1}^{\sigma_A, \sigma_H}$ e il token $\tau_{y, \alpha_A}^{\sigma_A, \sigma_H}$ controfirmati.

$$\begin{aligned} \tau_{y, \alpha_B}^{\sigma_A} &= \langle \nu_y, exp, \text{OFF} \rangle \\ \phi_{i+1}^{\sigma_A} &= \langle \beta_i^A - \nu_y, \beta_i^H, \tau_{y, \alpha_A}^{\sigma_A}, \mathbb{D} \rangle \end{aligned} \tag{8}$$

Alice presenta $\tau_{y, \alpha_A}^{\sigma_A, \sigma_H}$ in catena eseguendo la funzione attach dello smart contract. Lo smart contract aggiorna il balance on-chain di Alice in $\overline{\beta}_A + \nu$.

3.3.7 Ricarica a caldo

Effettuare una ricarica a caldo significa spostare ν fondi dal balance on-chain di Alice $\overline{\beta^A}$ a quello off-chain β_i^A .

Precondizioni

- a) $\{\alpha^A\} \subseteq \Pi^H$
- b) L'ultima propose confermata nel canale di Alice è ϕ_i^A
- c) Il balance on-chain di Alice è $\overline{\beta_A}$

Descrizione delle interazioni Alice esegue la funzione detach dello smart contract passando come parametri α_A e ν . Lo smart contract aggiorna il balance on-chain di Alice in $\overline{\beta^A} + \nu$. Una volta terminata l'esecuzione della funzione, lo smart contract invia all'hub e ad Alice la relativa PoD. Alice costruisce, firma e invia $\phi_{i+1}^{\sigma_A}$ all'hub. L'hub risponde con la propose $\phi_{i+1}^{\sigma_A, \sigma_H}$ e il token $\tau_{y, \alpha_A}^{\sigma_A, \sigma_H}$ controfirmati.

$$\begin{aligned} \tau_{y, \alpha_B}^{\sigma_A} &= \langle \nu_y, \perp, \text{OFF} \rangle \\ \phi_{i+1}^{\sigma_A} &= \langle \beta_i^A + \nu_y, \beta_i^H, \tau_{y, \alpha_A}^{\sigma_A}, \mathbb{A} \rangle_{(\sigma_A)} \end{aligned} \tag{9}$$

3.3.8 Chiusura di un canale

Precondizioni

- a) $\{\alpha^A\} \subseteq \Pi^H$
- b) L'ultima propose confermata nel canale di Alice è ϕ_i^A

Descrizione delle interazioni Alice porta in catena l'ultima propose ϕ_i^A con la funzione close dello smart contract. Lo smart contract registra la richiesta di chiusura del canale e avvia un timer di durata pari a una costante G dello smart contract, detta grace period. Scaduto il timer, Alice può riti-

rare tutti i suoi fondi $\overline{\beta^A} + \beta_i^A$ eseguendo la funzione withdraw dello smart contract.

3.3.9 Riscossione di un pending token

Un client può riscuotere dei pending token, ovvero dei token non ancora scaduti o utilizzati, durante il grace period.

Precondizioni

- a) Alice ha avviato la chiusura del canale.
- b) Il timer G non è ancora scaduto.

Descrizione delle interazioni Alice presenta in catena un pending token utilizzando la funzione redeemToken dello smart contract. L'esecuzione di questa funzione non corrisponde con il prelievo immediato del token. Una notifica della presentazione del token corrente viene inviata all'hub. Una volta scaduto G , Alice può riscuotere il suo balance (incrementato del quantitativo del token).

Tentativo di ritirare un pending token già usato Alice presenta in catena un pending token già riscosso. Durante il grace period l'hub può portare in catena la relativa PoD del token utilizzando la funzione argueRedemptionToken. Alice viene punita per il suo comportamento malevolo; tutti i suoi fondi (on-chain e off-chain) vengono trasferiti all'hub.

Capitolo 4

Progettazione e sviluppo

Questo capitolo descrive responsabilità, requisiti, motivazioni tecnologiche e dettagli implementativi di FulgurHub. In particolare in sezione 4.1 si descrive lo smart contract, in sezione 4.2 si descrive il client e in sezione 4.3 si descrive l’hub.

4.1 Smart contract

4.1.1 Responsabilità e requisiti

Lo smart contract è il punto di contatto tra lo stato off-chain e quello on-chain. Lo stato on-chain si basa su una mappa ti dipo *indirizzo* \rightarrow *Wallet*. Un Wallet è una struttura dati che gestisce tutte le informazioni on-chain di un utente iscritto all’hub: balance on-chain, PoDs, PoAs, timestamp di chiusura di un canale e token riscossi. Un utente dell’hub che vuole interagire con il suo stato on-chain può farlo eseguendo una delle operazioni messe a disposi-

zione. Queste operazioni riguardano l'iscrizione all'hub, i pagamenti ibridi, la chiusura di un wallet e la riscossione di pending token. Oltre a questo lo smart contract mette a disposizione degli eventi. Gli eventi sono dei messaggi che possono essere pubblicati nel momento in cui una qualche funzionalità viene eseguita. Questi eventi sono pubblici e chiunque può mettersi in ascolto. Di seguito gli eventi messi a disposizione:

- **Subscribed** viene sollevato quando un nuovo utente si registra; l'hub si iscrive a questo evento per poterlo memorizzare sul suo database.
- **TokenDetached** notifica che un token è stato staccato on-chain.
- **TokenAttached** notifica che un token è stato attaccato on-chain.
- **WalletClosed** notifica che un utente sta chiudendo un canale; questa notifica serve all'hub per verificare che non sia stata presentata una proposta non valida.

4.1.2 Motivazioni tecnologiche

La blockchain presa come riferimento è Ethereum. Le motivazioni che hanno mosso la scelta di questa blockchain rispetto ad altre riguardano il supporto di smart contract e l'ambiente di sviluppo maturo. In particolare sono stati utilizzati i seguenti tool e linguaggi: Solidity, Ganache, Web3.

Linguaggio di programmazione dello smart contract Solidity è il linguaggio di programmazione C-like turing completo con il quale è possibile sviluppare gli smart contract in FulgurHub. Esso mette a disposizione un compilatore e un debugger. Il compilatore trasforma il linguaggio in codice macchina compatibile con la EVM (Ethereum Virtual Machine). Il debugger di Solidity permette di conoscere lo stato intermedio di uno smart contract durante la sua esecuzione.

Rete blockchain di test Ganache è una blockchain di test locale, che semplifica la fase di test di uno smart contract; permette di deployare ed eseguire uno smart contract, senza utilizzare la rete principale di Ethereum, abbattendo costi e tempi di sviluppo.

Interfaccia smart contract Web3 è un'interfaccia in JavaScript che permette di eseguire le operazioni più comuni sulla blockchain di Ethereum (E.G. il deployment di uno smart contract, l'esecuzione di una funzione o un pagamento). Le interazioni con lo smart contract non avvengono direttamente con web3, ma sono wrappate da un'interfaccia di più alto livello. Si è deciso di utilizzare questa interfaccia per non legare il particolare tipo di blockchain adottata con l'implementazione in se. Sebbene infatti la scelta progettuale sia ricaduta su Ethereum, questo approccio consente di estendere le funzionalità implementate su diverse tipologie di blockchain. Il linguaggio di programmazione adottato per implementare l'interfaccia di livello più alto è TypeScript; è stato utilizzato TypeScript rispetto a JavaScript dato il supporto della tipizzazione forte. Questo ha permesso di definire interfacce stabili e di intercettare eventuali bug già in fase di compilazione.

Altre soluzioni tecnologiche Esistono altre interessanti soluzioni alternative a Ethereum. Una in particolare è Tezos. Tezos come Ethereum mette a disposizione la possibilità di deployare smart contract con un linguaggio di programmazione turing-completo. Il linguaggio di riferimento è Michelson, un subset di Ocaml che semplifica la verifica formale di correttezza di uno smart contract. Sebbene Tezos non sia stato utilizzato in fase di sviluppo, un suo futuro impiego potrebbe essere facilmente integrabile grazie alla definizione dell'interfaccia di alto livello dello smart contract.

4.1.3 Dettagli implementativi

Interfaccia in TypeScript Di seguito viene esposta l'interfaccia di alto livello dello smart contract in TypeScript. Il funzionamento delle singole operazioni è descritto in dettaglio nel capitolo 3.

```
interface SmartContract {  
    subscribe(wallet: Wallet);  
    detachToken(token: Token);  
    attachToken(token: Token);  
    transfer(payeeAddress: string, amount: BigNumber);  
    close(propose: Propose);  
    redeemToken(token: Token);  
    argueRedemptionToken(token: Token);  
    withdraw();  
    argueClosure(propose: Propose);  
}
```

Il tipo Wallet Il tipo Wallet rappresenta la registrazione di un utente su FulgurHub. Esso contiene l'indirizzo pubblico del client e dello smart contract, il bilancio on-chain/off-chain iniziale del client e il bilancio off-chain dell'hub.

Il tipo Propose Il tipo Propose autocontiene tutte le informazioni che descrivono una proposta: il nonce, l'indirizzo pubblico dell'utente, l'indirizzo dello smart contract, il bilancio off-chain corrente del client e dell'hub, il relativo token che si è deciso di attaccare o staccare e la firma della propose.

Il tipo Token Rappresenta un token. In particolare contiene: nonce, indirizzo dello smart contract, indirizzo pubblico del pagato, il quantitativo spostato, il tipo di catena dove può essere attaccato (off-chain o on-chain), la data di scadenza e la relativa firma.

4.2 Client

4.2.1 Responsabilità e requisiti

Il client è il modulo che permette a un utente di interagire con l'hub, gli altri client e lo smart contract; deve rimanere attivo per il tempo di vita del canale di pagamento instaurato con l'hub. Le sue responsabilità riguardano: esecuzione di comandi privati/pubblici, gestione di eventi asincroni e registrazione dei messaggi off-chain scambiati.

Comandi privati/pubblici Un comando privato può essere eseguito solamente dall'utente associato al canale di pagamento. Questi comandi permettono di registrarsi all'hub, effettuare dei pagamenti, chiudere un canale e riscuotere pending token. Un comando pubblico è accessibile a qualunque utente associato a un certo FulgurHub; questi permettono di ricevere pagamenti off-chain e ricevute di pagamento.

Messaggi asincroni Lo smart contract genera delle notifiche; le notifiche sono dei messaggi asincroni. Il client deve poter ricevere e gestire questi messaggi asincroni. Queste notifiche riguardano il detach di un token on-chain e la ricezione di una proof of detachment.

Registrazione messaggi off-chain Tutti i messaggi scambiati off-chain devono poter essere memorizzati in maniera permanente dal client.

4.2.2 Motivazioni tecnologiche

RPC privata / endpoint pubblici L'RPC privata e gli endpoint pubblici permettono di eseguire rispettivamente i comandi privati e pubblici. Entrambi sono stati implementati con un server http Node.js; questo ha permesso

di utilizzare TypeScript, mantenendo un unico linguaggio di programmazione per il backend. L'RPC è esposta su una porta privata (10101), mentre i comandi che devono essere accessibili a tutti sono esposti su una porta pubblica (80).

Il monitor La gestione di eventi asincroni come gli eventi generati dallo smart contract è fondamentale in FulgurHub; per questo motivo si è deciso di localizzare questa responsabilità in un modulo dedicato denominato monitor. Il monitor gestisce due eventi asincroni: *onChainDetachment* e *onProofOfDetachmentPushed*.

- *onChainDetachment* è un evento generato dallo smart contract quando qualcuno effettua il detach di un token on-chain a favore dell'utente corrente.
- *onProofOfDetachmentPushed* è un evento generato quando l'utente corrente riceve una nuova proof of detachment.

Il comportamento legato a un evento non è contenuto all'interno del monitor; il monitor infatti permette solo di agganciare o sganciare a un evento un certo insieme di comportamenti, ovvero di funzioni. Questo approccio consente di estendere facilmente le funzionalità del modulo e quindi migliora la modificabilità del progetto.

Il database La registrazione dei messaggi off-chain è stata delegata a un database. Priorità assoluta di questo database è che non rappresenti un collo di bottiglia per il throughput dei pagamenti. La scelta è ricaduta su LevelDB, un database chiave-valore embedded, single process, multi thread basato sulle API linux POSIX. Le motivazioni che supportano questa scelta riguardano le ottime performance in scrittura di LevelDB [18].

4.2.3 Dettagli implementativi

Di seguito sono descritti i principali endpoint privati suddivisi per categoria di operazione.

Iscrizione di un wallet Un utente può registrare un wallet su Fulgu-rHub mediante questo endpoint. Il tipo *Wallet* contiene tutte le informazioni necessarie e viene passato come unico parametro http:

POST: `/subscribe`

```
{  
  wallet: Wallet  
}
```

Pagamento OnChain-OnChain Come visto in capitolo 3 i pagamenti OnChain-OnChain vengono gestiti dallo smart contract. Questo endpoint privato avvia il trasferimento eseguendo l'operazione *transfer* dello smart contract:

POST: `/transferOnChainOnChain`

I parametri necessari a eseguire questa operazione sono *recipientAddress*, ovvero l'indirizzo del pagato e *amount* ovvero l'importo che si vuole trasferire.

```
{  
  recipientAddress: string,  
  amount: BigNumber  
}
```

Pagamento OffChain-OffChain La prima fase di un pagamento OffChain-OffChain consiste nell'effettuare il detach di un token OffChain-OffChain, ovvero di un token che è stato staccato off-chain e che verrà attaccato off-

chain. I parametri necessari a eseguire il detach sono *addressPayee*, ovvero l'indirizzo pubblico del pagato, *uriPayee* l'indirizzo del server del pagato, *amount* il quantitativo che si intende trasferire e *tvl* il tempo di vita del token:

POST: `/detachOffChainTokenOffChain`

```
{  
    addressPayee: string,  
    uriPayee: string,  
    amount: BigNumber,  
    ttl: BigNumber  
}
```

Una volta ricevuta la proof of detachment dall'hub, essa può essere inviata al client mediante questo endpoint::

POST: `/sendProofOfDetachment`

```
{  
    proofOfDetachment: Token  
}
```

Il pagato recupera le proof of detachment ricevute mediante questo endpoint:

POST: `/popProofOfDetachment`

e invia al client pagante la ricevuta di pagamento:

POST: `/sendPaymentReceipt`

```
{  
    paymentReceipt: PaymentReceipt  
}
```

Questo endpoint permette al pagante di una transazione OffChain-OffChain di ribilanciare il canale. Con questo comando una transazione OffChain-OffChain viene considerata conclusa:

POST: `/settleOffChainOffChainTransfer`

```
{  
  propose: Propose  
}
```

Pagamento OnChain-OffChain Avvia un pagamento OnChain-OffChain.

I parametri necessari sono gli stessi dell'endpoint `/detachOffChainTokenOffChain`:

POST: `/detachOnChainTokenOffChain`

```
{  
  addressPayee: string,  
  uriPayee: string,  
  amount: BigNumber,  
  ttl: BigNumber  
}
```

Pagamento OffChain-OnChain Avvia un pagamento di tipo OffChain-OnChain:

POST: `/detachOffChainTokenOnChain`

```
{  
  addressPayee: string,  
  uriPayee: string,  
  amount: BigNumber  
}
```

Incassare un token Una volta ricevuto un off-chain token, questo può essere riscosso mediante questo endpoint:

POST: `/attachTokenOffChain`

```
{
  proofOfDetachment: Token
}
```

Incassare un pending token Un pending token può essere incassato durante il grace period del canale mediante questo endpoint:

POST: `/redeemToken`

```
{
  token: Token
}
```

POST: `/retrievePaymentReceipt`

Nel caso in cui il pagato non sia collaborativo un utente può richiedere una ricevuta di pagamento all'hub utilizzando questo endpoint:

```
{
  clientAddress: string
}
```

Chiudere un canale Avvia la chiusura del canale di pagamento. La chiusura avviene presentando in catena *latestPropose*, ovvero l'ultima propose concordata tra client e hub:

POST: `/close`


```
{  
    latestPropose: Propose  
}
```

Finalizzare chiusura Terminato il grace period, il client può effettuare il *withdraw*, finalizzando la chiusura del canale. Questo endpoint permette di eseguire questa operazione:

POST: */withdraw*

4.3 Hub

4.3.1 Responsabilità e requisiti

Chiunque abbia abbastanza fondi on-chain può inizializzare un FulgurHub. Per fare questo occorre deployare il relativo smart contract e mantenere costantemente attivo il modulo descritto in questa sezione. L'hub è un modulo software molto simile al client. Le sue responsabilità riguardano:

- **Esecuzione di comandi pubblici** Gli utente devono poter contattare l'hub eseguendo dei comandi pubblici.
- **Gestione di eventi asincroni** L'hub deve poter gestire degli eventi asincroni. Nell specifico le notifiche generate dallo smart contract.
- **Registrazione messaggi off-chain** Tutti i messaggi off-chain scambiati con gli utenti dell'hub devono poter essere memorizzati; essi infatti rappresentano delle prove di avvenuto pagamento che potrebbero dover essere presentate nel futuro in catena.

I principali requisiti architetturali dell'hub sono i seguenti:

- **Performance** L’hub deve eseguire le singole operazioni velocemente; questo è essenziale specialmente nel caso in cui occorre gestire frequenti micropagamenti.
- **Scalabilità** L’hub deve poter scalare orizzontalmente; questo significa che per far fronte a crescente numero di transazione basterà aggiungere dei nodi di calcolo.
- **Modificabilità** La base di codice deve poter essere facilmente modificabile ed estensibile.

4.3.2 Motivazioni tecnologiche

Gli endpoint pubblici L’hub è un modulo passivo; questo significa che non contatta mai deliberatamente un utente, ma è quest’ultimo che passivamente riceve dei comandi dall’hub. Questi comandi vengono impartiti mediante degli endpoint http pubblici. Come nel client, il server http è stato implementato mediante Node.js; questo ha permesso di mantenere TypeScript come unico linguaggio di backend.

Il monitor Come nel client anche nell’hub la gestione degli eventi asincroni è delegata a un modulo denominato monitor. Il modulo permette di agganciare a un evento un certo comportamento, senza cambiare il contenuto del monitor stesso. L’aggiunta o la rimozione degli eventi è rara, mentre invece la modifica del comportamento legato a un evento può cambiare frequentemente. Questo facilita l’estensione della gestione degli eventi, migliorando la modificabilità dell’architettura.

Database Come descritto in capitolo 3 l’hub riceve messaggi firmati dai client che deve memorizzare. Per la natura del protocollo di FulgurHub questi messaggi vengono frequentemente memorizzati e raramente letti. Il numero

delle scritture può essere anche ingente. Per questo motivo si è deciso di utilizzare un database chiave valore, in particolare Redis, dato il suo considerevole throughput in scrittura [19]. Altro motivo per cui è stato adottato Redis rispetto a un altro database chiave-valore è rappresentato dalla possibilità di effettuare tuning delle sue qualità architetturali. In particolare il teorema CAP dice che un'architettura può avere solo due tra queste caratteristiche contemporaneamente:

- Consistenza
- Disponibilità
- Partizionamento

Redis permette di scegliere quali di queste due caratteristiche avere. In una prima fase di un FulgurHub ha senso scegliere solamente la consistenza e la disponibilità. Sebbene un requisito essenziale dell'architettura sia la scalabilità, una singola istanza Redis su commodity hardware garantisce un throughput ampiamente sufficiente [19].

Nel caso in cui si debba aumentare il numero di transazioni al secondo si potrà scegliere tra scalare verticalmente l'hardware o abilitare lo sharding a sfavore della disponibilità.

4.3.3 Dettagli implementativi

Di seguito vengono descritti gli unici endpoint pubblici che mette a disposizione l'hub.

Ricezione di una propose L'hub aggiorna lo stato off-chain di un canale di pagamento mediante la ricezione da parte dell'utente di una propose. Gli utenti possono inviare una propose mediante l'uso del seguente endpoint:

POST: `/sendPropose`

```
{  
  clientSignedPropose: Propose  
}
```

Richiesta di ricevuta di pagamento Quando in un pagamento OffChain-OffChain il pagato non è collaborativo, il pagante può richiedere la ricevuta di pagamento all'hub mediante questo endpoint:

POST: `/retrievePaymentReceipt`

```
{  
  clientAddress: string  
}
```

Capitolo 5

Prove sperimentali

Questo capitolo discute le prove sperimentali condotte sull'implementazione di FulgurHub. In particolare in sezione 5.1 si discutono gli obiettivi, in sezione 5.2 l'approccio adottato, in sezione 5.3 si mostrano i risultati relativi al client, in sezione 5.4 si discutono i risultati dell'hub, in sezione 5.5 viene descritto il profiling dell'operazione di pagamento OffChain-OffChain e infine in sezione 5.6 si fanno delle considerazioni generali sui risultati ottenuti in termini di performance e scalabilità.

5.1 Gli obiettivi

Verifica performance Un obiettivo delle prove sperimentali è stato verificare le performance dell'architettura; in particolare l'analisi del throughput di client e server relativamente ai pagamenti OffChain-OffChain. Sebbene siano state implementate anche altre tipologie di pagamento (OffChain-OnChain, OnChain-OffChain e OnChain-OnChain) si è preferito non effettuare prove

di performance di tutte le operazioni che interagiscono con la catena. Il throughput delle operazioni che interagiscono con la catena sarebbe limitato superiormente dal throughput della blockchain di riferimento. In questo contesto con throughput si intende il numero di transazioni completate in un secondo.

Profiling Altro obiettivo delle prove sperimentali è stato la profiling dei pagamenti OffChain-OffChain; come visto in capitolo 3, un pagamento OffChain-OffChain è costituito da un insieme di sotto task; la durata di ciascun sotto task è stata profilata, con l'intento di trovare eventuali colli di bottiglia e di capire quale sia la distribuzione delle operazioni nel tempo.

5.2 L'approccio adottato

Benchmark server Eseguire un test delle performance di FulgurHub richiede il setup di un ambiente complesso e distribuito. In particolare ciascun client e hub dovrebbe risiedere su un nodo di calcolo dedicato. A tale scopo è stato realizzato un benchmark server. Il benchmark server permette di automatizzare il setup dell'ambiente di test e di eseguire dei performance test parametrizzati. L'esecuzione delle operazioni avviene mediante chiamate http. Di seguito si descrivono i comandi messi a disposizione dal server.

Questo endpoint permette di effettuare il setup dell'ambiente di test, in particolare vengono deployati i seguenti servizi:

- **Redis** Un'istanza di Redis viene deployata. Essa rappresenta il database dell'hub.
- **Hub** Con hub si intende il server dell'hub. Anch'esso viene deployato su un nodo dedicato.

- **Client** Ciascun client viene deployato su un nodo dedicato. Il numero di client da deployare viene specificato mediante il parametro *numberOfClients*.
- **Ganache** Una blockchain di test deve essere deployata per supportare le operazioni on-chain necessarie come la sottoscrizione dell'hub. A tale scopo è stato utilizzato ganache.

POST: `/environment`

```
{  
  numberOfClients: Number  
}
```

Per cancellare l'ambiente di test un comando dedicato viene messo a disposizione:

DELETE: `/environment`

Mediante il seguente comando è possibile eseguire un benchmark relativo ai pagamenti OffChain-OffChain; è possibile specificare due parametri: *concurrent* e *requests*. *concurrent* indica il numero di coppie di utenti che devono scambiare dei pagamenti in maniera concorrente. *requests* indica quanti pagamenti deve eseguire ciascuna coppia di utenti.

POST: `/benchmark/transfer/offchain/offchain`

```
{  
  concurrent: Number,  
  requests: Number  
}
```

Docker Ciascun nodo dell'ambiente di test è stato deployato su un container LXC. In particolare il benchmark server utilizza le API di Docker, per

costruire e distruggere i container di cui necessita. L'uso di nodi virtualizzati rispetto a nodi fisici reali ha vari vantaggi: tra cui l'abbattimento dei costi e dei tempi di sviluppo e la possibilità di aumentare o diminuire le risorse hardware dedicate di ciascun container modificandone la configurazione. D'altra parte la virtualizzazione non permette di tenere conto della latenza di rete.

Simulazione della latenza di rete Come detto nel precedente paragrafo, la virtualizzazione non permette di tenere conto della latenza di rete. Per questo motivo in fase di test è stata simulata ritardando di un fattore delta le comunicazioni remote.

5.3 Throughput del client

Con throughput del client si intende il numero di pagamenti al secondo confermati che un singolo utente può effettuare. Esso è stato verificato sia al variare della RAM che della latenza. Il client è realizzato in Node.js, questo significa che viene eseguito un unico processo senza alcun thread di supporto; per questo motivo non sono state effettuate delle prove sperimentali al variare del numero di core a disposizione.

Al variare della RAM Questo test è stato effettuato variando il quantitativo di RAM assegnato ai nodi del client e senza variare quella dedicata all'hub. Come è possibile verificare dai dati in tabella e dal grafico in figura 10 il client raggiunge il throughput massimo con un 1 o 2 GB di RAM.

Tabella 4: Throughput client al variare della RAM.

N°	RAM	Throughput (tx/s)
1	500MB	13

N°	RAM	Throughput (tx/s)
2	1GB	32
3	2GB	41

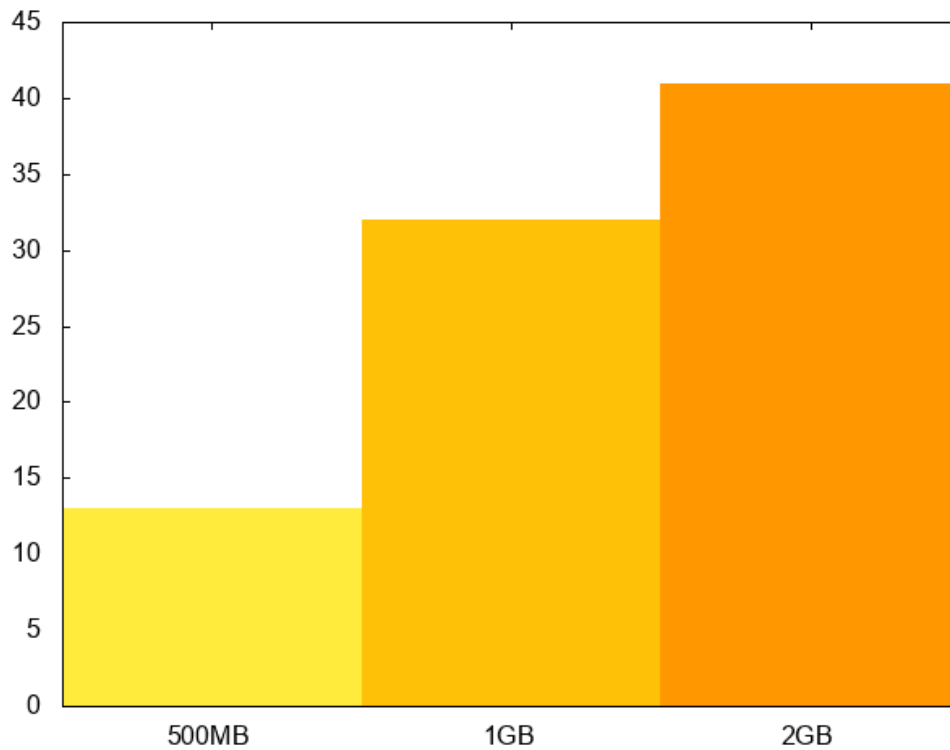


Figura 10: Throughput client al variare della RAM

Al variare della latenza I test sulla latenza sono stati eseguiti fissando la RAM di tutti i nodi a 2GB e simulando un ritardo tra tutte le connessioni remote instaurate. Come è possibile notare in figura 11 la latenza incide in maniera direttamente proporzionale sul throughput del client.

Tabella 5: Throughput client al variare della latenza.

N°	Ritardo simulato (s)	Throughput (tx/s)
1	0	43
2	0.5	22
3	1	13.5
4	2	7
5	10	1.4

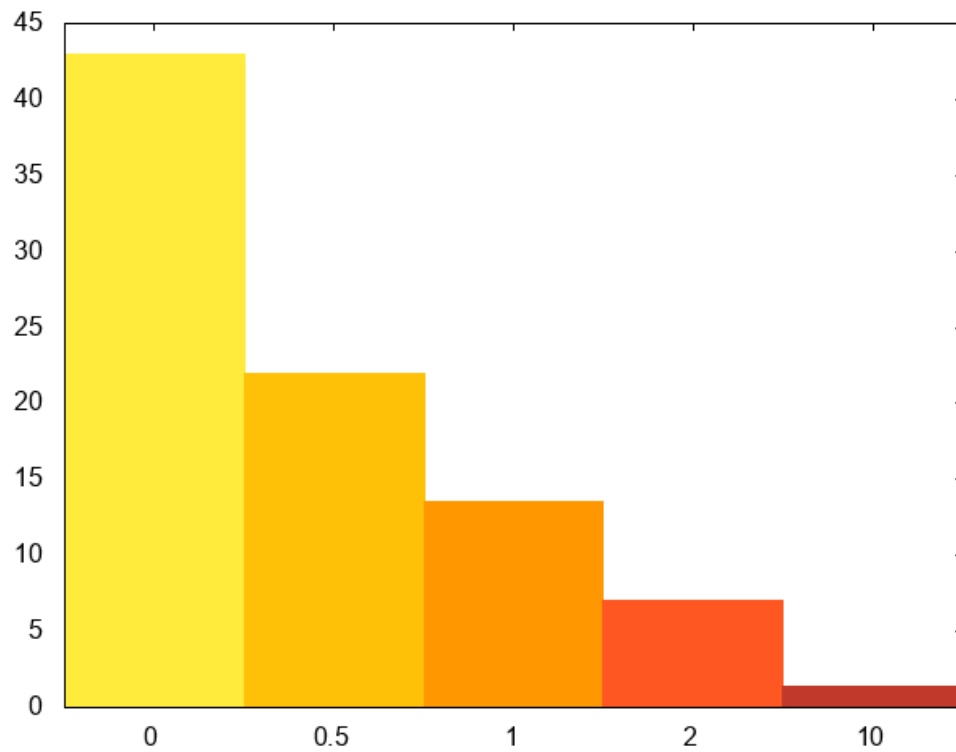


Figura 11: Barchart throughput client al variare della latenza.

5.4 Throughput dell'hub

Con throughput dell'hub si intende il numero di pagamenti concorrenti al secondo confermati che un singolo hub può gestire. Esso è stato verificato sia al variare della RAM che della latenza. Anche l'hub come il client è realizzato in Node.js, questo significa che viene eseguito un unico processo senza alcun thread di supporto; per questo motivo non sono state effettuate delle prove sperimentali al variare del numero di core a disposizione.

Al variare della RAM Il test è stato effettuato variando il quantitativo di RAM assegnata all'hub. Come è possibile verificare dal grafico in figura 12 l'hub raggiunge il throughput massimo con un 1 o 2 GB di RAM. In particolare su un nodo di calcolo con 500MB di RAM il numero di transazioni completate al secondo è pari a 45. Aumentando la RAM a 1GB il throughput quasi raddoppia, raggiungendo 84 transazioni al secondo.

Tabella 6: Throughput hub al variare della RAM.

N°	RAM	Throughput (tx/s)
1	500MB	45
2	1GB	84
3	2GB	84

Al variare della latenza I test sulla latenza relativi all'hub sono stati effettuati fissando a 2GB la RAM di tutti i nodi e simulando un ritardo tra tutte le connessioni remote instaurate. Come è possibile notare in figura 13, la latenza incide in maniera direttamente proporzionale sul throughput del client.

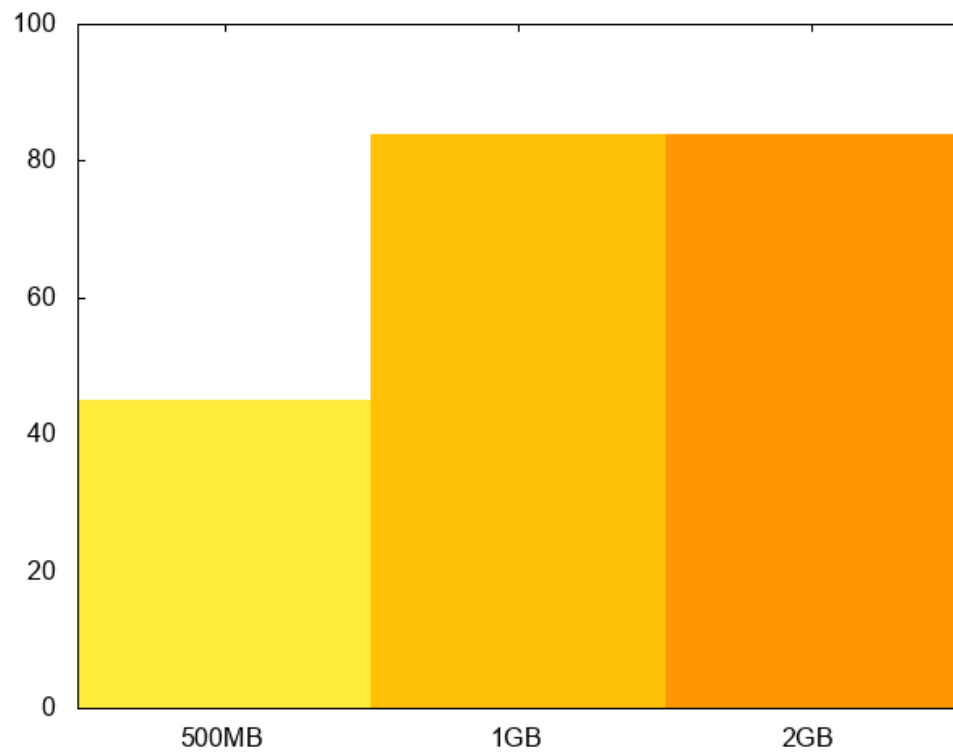


Figura 12: Barchart throughput hub al variare della RAM

Tabella 7: Throughput hub al variare della latenza.

N°	Latenza (s)	Throughput (tx/s)
1	0	84
2	0.5	43
3	1	21
4	2	13
5	10	2

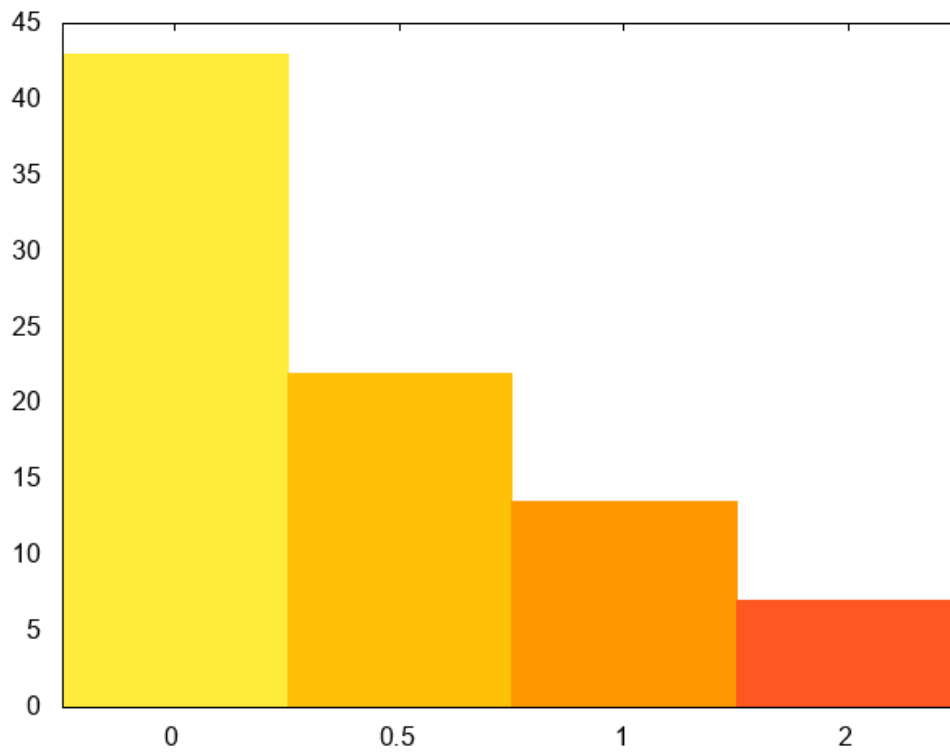


Figura 13: Barchart throughput hub al variare della latenza.

5.5 Profiling

Come visto nel capitolo 3 un pagamento OffChain-OffChain per essere considerato concluso richiede lo scambio di un certo quantitativo di messaggi, vedi figura 14. In sezione 5.3 si è visto che su una macchina con 2GB di RAM un client riesce a completare fino a 41 pagamenti al secondo; quindi un pagamento viene completato in circa 24ms.

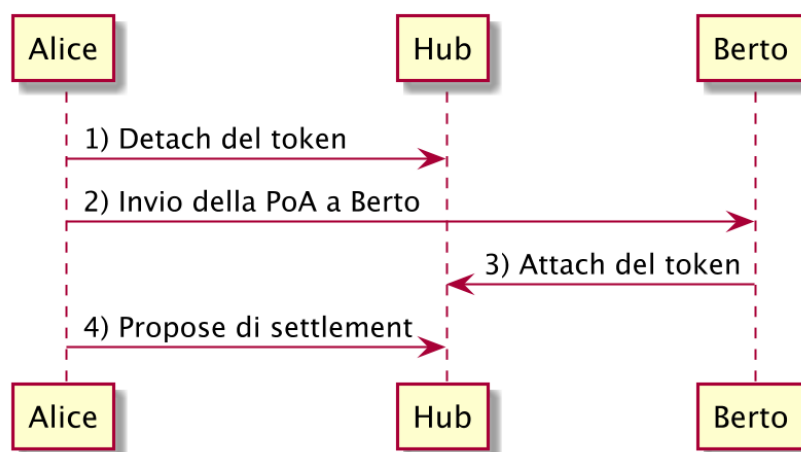


Figura 14: Pagamento OffChain-OffChain in FulgurHub.

Rispetto alle operazioni indicate in figura 14, la loro distribuzione nel tempo è indicata nel diagramma in figura 15.

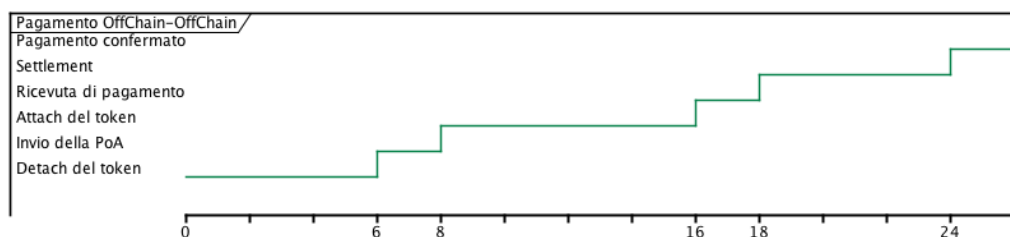


Figura 15: Profiling pagamento OffChain-OffChain

Come è possibile notare dalla figura 15 le operazioni più onerose risultano essere il detach del token, l'attach del token e l'invio della propose di settlement, ovvero tutte le operazioni che richiedono la firma di un messaggio. Questo risultato è quello aspettato, infatti la firma crittografica richiede un lavoro computazionale maggiore rispetto alla lettura e invio di un messaggio. Node.js dal punto di vista dell'ottimizzazione di operazioni computazionali onerose non è la tecnologia adatta, data la sua natura single thread; per questo motivo la firma dei messaggi è delegata a un modulo software dedicato e ottimizzato scritto in C basato sulle N-API di Node.js.

5.6 Considerazioni

Sulla base dei dati presentati nelle precedenti sezioni, si discutono i risultati in termini di performance e di scalabilità.

Performance Su un singolo nodo con hardware adeguato, l'implementazione di FulgurHub presentata in questa tesi può arrivare a gestire fino a 84 transazioni al secondo. In figura 16 si confronta il throughput di FulgurHub con quello delle principali blockchain. Come è possibile notare FulgurHub permette di ottenere un throughput 5 volte maggiore rispetto a quello della blockchain sottostante (Ethereum) e di decuplicare quello relativo all'attuale implementazione di BitCoin.

Tabella 8: Throughput FulgurHub a confronto con le principali blockchain.

N°	Tecnologia	Throughput (tx/s)
1	BitCoin	7
2	Ethereum	15

N°	Tecnologia	Throughput (tx/s)
3	FulgurHub	84

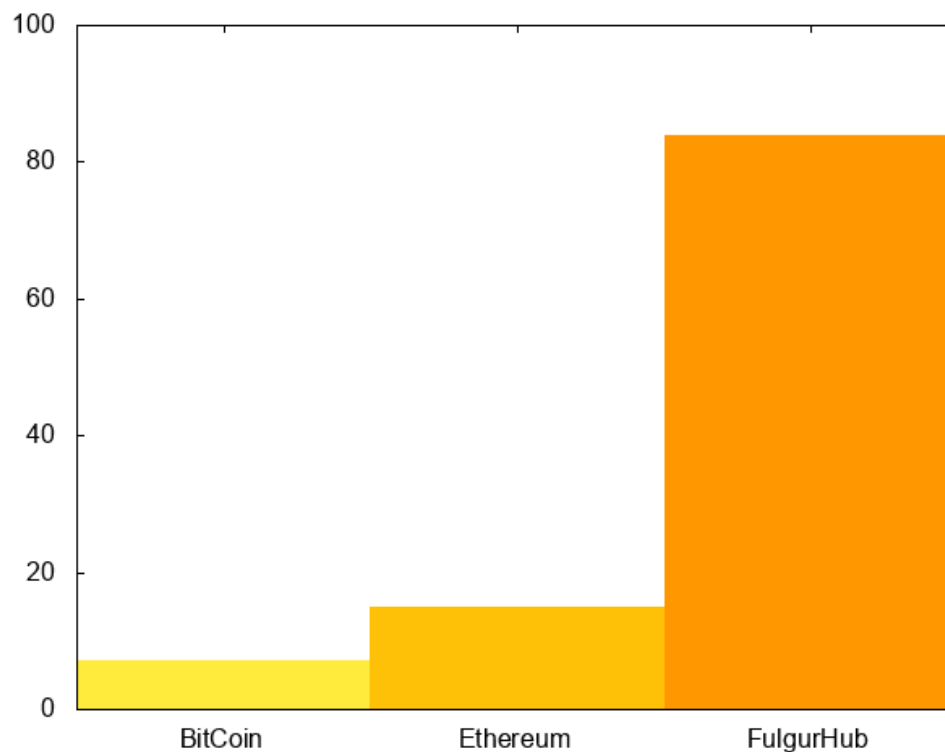


Figura 16: FulgurHub a confronto con tecnologie alternative

Scalabilità dell’hub Un singolo nodo di FulgurHub permette di ottenere un throughput sensibilmente maggiore rispetto a quello delle principali blockchain. Oltre a questo l’hub è un modulo stateless, ovvero il singolo nodo hardware non ha uno stato; lo stato dell’hub infatti è memorizzato su un database dedicato (Redis); questa caratteristica permette di replicarlo senza particolari difficoltà. Con una configurazione adeguata, basata su un load balancer che distribuisca le richieste su più istanze, il numero di transazioni gestite

dall'hub può scalare linearmente, in maniera direttamente proporzionale al numero di nodi.

Capitolo 6

Conclusioni e sviluppi futuri

Autogestione finanziaria

Endpoint denominati in maniera diversa

[1] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. 2016. Cryptocurrencies without proof of work. In *International conference on financial cryptography and data security*, 142–157.

[2] Thomas Bocek, Sina Rafati, Bruno Rodrigues, and Burkhard Stiller. 2017. Coinblesk—a real-time, bitcoin-based payment approach and app. *Blockchain Engineering* (2017), 14.

[3] Vitalik Buterin and others. 2014. A next-generation smart contract and decentralized application platform. *white paper* (2014).

[4] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. 2016. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 acm sigsac conference on computer and communications security*, 3–16.

- [5] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. 2017. TumbleBit: An untrusted bitcoin-compatible anonymous payment hub. In *Network and distributed system security symposium*.
- [6] Sunny King and Scott Nadal. 2012. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August 19*, (2012).
- [7] Jae Kwon. 2014. Tendermint: Consensus without mining. *Draft v. 0.6, fall* (2014).
- [8] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 acm sigsac conference on computer and communications security*, 17–30.
- [9] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).
- [10] Joseph Poon and Thaddeus Dryja. 2016. The bitcoin lightning network: Scalable off-chain instant payments. *draft version 0.5 9*, (2016), 14.
- [11] Federico Spini. 2018. Fulgur: Hybrid trustless wallet. (2018).
- [12] The bitcoin boom | the new yorker.
- [13] Average number of transactions per block.
- [14] Visa inc. Is a global payments technology company that connects consumers, businesses, financial institutions and governments in more than 200 countries and territories, enabling them to use electronic payments instead of cash and checks.
- [15] A proof of stake design philosophy | vitalik buterin.

[16] State channels - an explanation.

[17] Raiden network.

[18] Google/leveldb: LevelDB is a fast key-value storage library written at google that provides an ordered mapping from string keys to string values.

[19] How fast is redis? – redis.