Hypercore Protocol

Jacopo Corina - jacopo.corina@studio.unibo.it

Luglio 2023

Indice

Capitolo 1

Introduzione

Le reti peer-to-peer (P2P) sono reti di comunicazione in cui le entità coinvolte non presentano un ruolo predefinito e nelle quali si prevedono contatti diretti senza intermediari, in modo opposto a quanto avviene nell'architettura client-server. Esse hanno iniziato il loro processo di diffusione ed evoluzione sulla fine degli anni 90 del secolo scorso, nascendo per facilitare la condivisione di file e porre un'alternativa ai sistemi centralizzati. Questi fattori hanno portato ad un interesse crescente da parte della ricerca scientifica, dato che molte reti sono nate appositamente per divulgare ricerche, essere indipendenti da terzi ed evitare possibili censure. Rimanendo nell'ambito del file sharing, molti datasets utilizzati sono condivisi in rete ma per caratteristiche legate ai protocolli HTTP e FTP, sono oggetto di possibili corruzioni di dati in quanto non sono presenti nativamente sistemi di supporto al *versioning* o al *content addressing*. Per sopperire a questa mancanza, il mercato dei servizi informatici al consumo si è arricchito di prodotti sempre più dotati di funzionalità ed interfacce user-friendly, come Dropbox, Amazon S3 o Google Drive. Questi prodotti, sfruttando economie di scala, possono portare a notevoli risparmi economici e riduzioni di investimenti iniziali. Inoltre, danno supporto al controllo del versionamento ma sono basati su architetture e sistemi proprietari, che possono implicare costi aggiuntivi di migrazione, limitazioni di banda per utente, rischi di sicurezza legati a problemi di privacy, perdita di possesso dei dati, rischio di *lock-in*.

L'utilizzo di sistemi di file sharing distribuito porta ad un miglioramento di *trust* e di utilizzo della banda disponibile evitando la saturazione di un unico host. I peer coinvolti compongono una nuova rete, detta *overlay network*, nella quale possono compiere scelte locali in autonomia. Per fornire un'operatività efficiente ed efficace, queste reti devono essere gestite attraverso sistemi di peer discovering, sia per ottimizzare le policy di download, sia perchè alcuni peer,

ad un certo momento, potrebbero non essere più disponibili o affidabili e quindi andranno esclusi.

Capitolo 2

Evoluzione dei sistemi P2P

La tecnologia P2P è divenuta popolare grazie ai sistemi di file sharing come Napster, al quale sono poi seguiti Gnutella e Bittorrent. Bittorrent è un protocollo che permette la diffusione di file tramite la condivisione di file *torrent*, i quali rappresentano una specifica *overlay network* e permettono di entrare in contatto con i peer che possiedono, anche parzialmente, un insieme di blocchi necessario a comporre il file richiesto. Esso ha la peculiarità di aver introdotto meccanismi di ottimizzazione come *rarest piece first*, col quale si predilige la ricerca e il download di blocchi meno diffusi nella *overlay network*, ed il concetto di incentivazione *tit-for-tat*, in modo da ottimizzare l'affidabilità della rete e limitare l'utilizzo di banda a quei peer che tentano solo di scaricare blocchi, senza condividere anche ciò che è in loro possesso.

Seguono ulteriori evoluzioni legate alla nascita di Kademlia e alla diffusione di *content-centric networks*, che evolvono il concetto di network layer: un contenuto può essere richiesto senza conoscere a priori dove sia e qualunque nodo può fornirlo, se ne è in possesso. Ciò implica l'uso esteso di *caching* e la necessità di adottare funzioni crittografiche di hashing, note come *self-certifying names*. Queste funzioni di hashing, se caratterizzate di resistenza alla pre-immagine e alla collisione, impediscono che a parità di output, detto *digest*, corrispondano diversi input di partenza ma non danno certezze sull'origine del dato, ad esempio informazioni sul suo creatore. Associando un sistema di firme crittografiche, è possibile porre una soluzione a ciò, ma si introducono ulteriori problematiche legate alla validità delle chiavi ed alla loro diffusione.

Di seguito vengono brevemente descritti alcuni tra i più diffusi file system P2P:

- IPFS (Interplanetary File System): è un insieme di moduli che permettono di gestire un oggetto, suddividerlo in *chunk* identificati da un codice univoco detto *CID*, e memorizzarli in una struttura dati a grafo. Attraverso *tabelle hash distribuite* è in grado di individuare quali peer sono in possesso di un determinato *chunk* per poi avviarne lo scambio. Sono previsti meccanismi per incentivare la persistenza del dato da parte dei peer.

- Swarm: è una piattaforma, basata su *Ethereum*, che permette scambio di oggetti. Anch'essa è basata sull'indirizzamento di contenuti, che in questo caso viene utilizzato in modo da individuare dove memorizzare i *chunks*, determinando così delle aree di responsabilità che incrementano la disponibilità del dato. Anche in questo caso il peer discovering è basato su *tabelle hash distribuite* e sono presenti meccanismi di incentivazione;

- SAFE (Secure Access For Everyone): è una piattaforma che si concentra principalmente sul fornire la garanzia dell'anonimato: l'autenticazione di un peer è effettuata tramite *self-authentication*, senza contattare componenti centralizzate, ed i file sono criptati attraverso un algoritmo di *self-encryption*. Per decriptarli, viene creata una struttura dati ausiliaria che può essere resa disponibile ai membri della rete in base alla riservatezza che si è scelto di utilizzare. Anche in questo caso vengono utilizzate *tabelle hash distribuite* e meccanismi di incentivazione;

- Storj: è una rete di storage peer-to-peer progettata per essere utilizzata in ambiente cloud. Possiede un'interfaccia compatibile con Amazon S3, la quale permette di manipolare i dati in massima libertà.

  Si compone di 3 entità:

  - nodo *satellite*: rappresenta una partizione della rete e funge da coordinatore per la scrittura e lettura di un dato;
  - nodo *uplink*: rappresenta l'utente finale che chiede di memorizzare e recuperare dati;
  - nodo *storage*: rappresenta l'unità in cui vengono memorizzati i dati. Viene utilizzata una tecnica di *erasure encoding* per ridurre la latenza di lettura, a costo di introdurre un overhead in fase di scrittura.

Capitolo 3

Hypercore Protocol

Hypercore Protocol è un insieme di moduli, volto a creare reti peer-to-peer per lo scambio di file, e, in generale, di dati. La sua architettura è altamente modulare ed in continua evoluzione: a partire dal modulo base denominato **Hypercore**, è prevista la disponibilità di altri aggiuntivi, come **Corestore**, **Hyperbee**, **Hyperdrive**, volti ad una manipolazione più ad alto livello, e **Hyperswarm**, adibito al peer discovering.

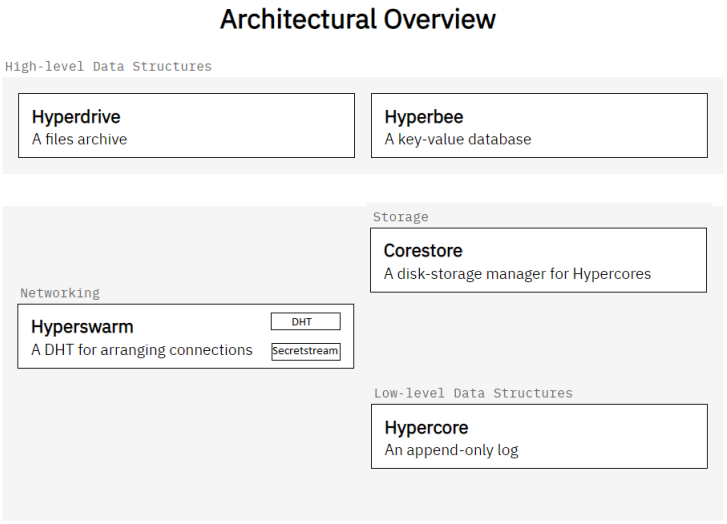


Figura 3.1: Principali moduli che compongono Hypercore Protocol

3.1 Hypercore

Hypercore è l'implementazione del concetto di *registro*, ossia un log binario append-only. Un peer può creare un'instanza di Hypercore, chiamata core, che inizialmente contiene solo metadati sulla propria struttura. Quando si aggiunge un dato, detto anche blocco, ad un core, si può pensare per analogia all'inserimento di un elemento in una lista di elementi concatenati, nella quale è possibile aggiungere e recuperare elementi, ma non modificarli, permettendo così di mantenere lo storico delle operazioni compiute. Tra i metadati è indicato quali blocchi sono presenti localmente nel peer, e quali no, ed essi possono essere richiesti ed inviati singolarmente *on-demand*, coinvolgendo possibilmente un numero elevato di peer in parallelo per permettere download e upload *sparsi* del contenuto. Successive modifiche possono essere attuate solamente dal "creatore" del core, in quanto unico possessore della *chiave privata* e possono essere lette da qualunque "lettore" in possesso della *chiave pubblica*, la quale assicura *read capability*. La caratteristica di *read capability* sott'intende che, qualora la *chiave pubblica* venisse utilizzata per l'individuazione di peer, si manifesterebbe il rischio di *capability leaks*. In virtù di ciò, a partire dalla *chiave pubblica* viene ricavata un'altra chiave, detta *discovery key*, la quale può essere utilizzata per annunciarsi ad altri peer che vogliono accedere al contenuto del core, senza rivelare la *chiave pubblica*.

3.1.1 Integrità e indirizzamento del contenuto

Partendo dal fatto che alcuni peer, indipendentemente dalla propria volontà, possano ad un certo punto adottare comportamenti indesiderati, è opportuno verificare che i dati ricevuti corrispondano esattamente ai dati attesi. La disponibilità di *digest* del contenuto desiderato permette di effettuare controlli di integrità ed evitare fenomeni di *content drift*, ossia scaricare dati diversi da quelli che ci si aspetta dopo averli richiesti tramite una connessione affidabile (*trustable*). Hypercore Protocol utilizza `BLAKE2b`[2] come algoritmo di hashing.

Un ulteriore vantaggio che si ottiene con l'utilizzo di tecniche di hashing, consiste nell'indirizzamento dei dati (*content addressing*), caratteristica essenziale per permettere il versionamento: ciascun cambiamento che si apporta ad un contenuto corrisponderà ad un nuovo hash, associabile ad una specifica versione.

3.1.2 Merkle Tree

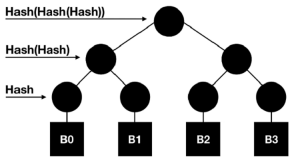
Per sfruttare al meglio le proprietà di integrità ed indirizzamento dei dati, essi sono organizzati utilizzando una struttura dati ad albero chiamata `Merkle Tree`[5]: basandosi su uno schema deterministico detto *binary in-order interval numbering ("bin")*, i nodi foglia (enumerati con indice pari) contengono informazioni effettive del dataset, utilizzabili per generare il proprio hash, mentre i nodi genitori (enumerati con indice dispari, radice compresa) contengono un hash derivabile dall'operazione di hashing dei due nodi figlio. Pertanto, la stringa hash associata ad una radice può essere riprodotta solo se tutti i dati a valle sono rimasti invariati, e di conseguenza due `Merkle Trees` con stesso contenuto avranno la stringa hash della radice coincidente. In figura 3.2 viene mostrato il processo di verifica di integrità di un blocco. La struttura dati consente di essere navigata ed è possibile richiedere solo un determinato sotto-albero, permettendo quindi accessi parziali e sparsi. Relativamente alla modifica dei nodi, viene utilizzata crittografia assimmetrica per firmare la radice, quindi solo il possessore della *chiave privata* può eseguire variazioni.

3.2 Hyperswarm

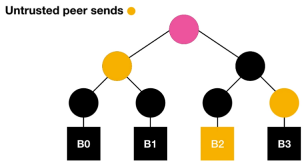
Hyperswarm ha il compito di mettere in comunicazione i peer, senza dover conoscere esattamente la loro locazione fisica. Esso si basa su un particolare tipo di *tabella hash distribuita* chiamata *Kademlia*[4], la quale usa il protocollo UDP ed è utilizzata in molti progetti famosi tra cui Bittorent.

Per poter funzionare, sono disponibili 3 nodi pubblici di default, che permettono di effettuare la fase di bootstrap iniziale.

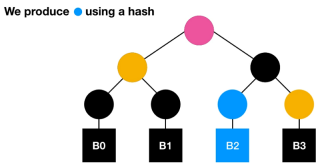
Un peer può utilizzare la *discovery key* di un core per unirsi ad un'istanza di Hyperswarm e renderlo disponibile ad altri. La caratteristica principale di Hyperswarm consiste nel fornire la *feature* di `holepunching`: nelle reti P2P ideali, i membri della rete dovrebbero avere la possibilità di connettersi "direttamente", in modo indipendente dalla relativa locazione. Nella realtà, l'utilizzo diffuso di firewall, che applicano di default il rifiuto di richieste in ingresso, unito alla diffusione di NAT, che mascherano gli IP, rendono necessaria l'introduzione di un componente aggiuntivo che sia conosciuto dai membri interessati e permetta di metterli in contatto tra loro. In Hyperswarm qualunque peer presente in DHT può collaborare a realizzare `holepunching` verso altri peer conosciuti, realizzando di fatto un `distributed holepunching`.



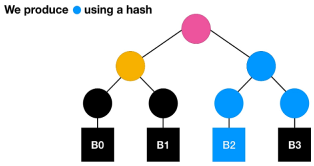
(a) A partire dai blocchi, i vari digest hash si costruiscono ricorsivamente



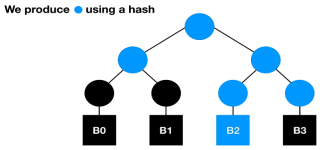
(b) Si riceve il blocco B2, assieme al hash del fratello $hash_a$(B3) e al hash del fratello del padre $hash_b$(hash(B0) combined with hash(B1))



(c) Si genera $hash_c$(B2)



(d) Si genera $hash_d$($hash_c$ combined with $hash_a$)



(e) Si verifica se hash(root) coincide con hash($hash_b$ combined with $hash_d$)

Figura 3.2: Flusso di verifica di integrità di un blocco in un Merkle tree

3.2.1 Kademlia: P2P overlay network

Per mettere in contatto i peer tra loro è possibile adottare un authority centralizzata; tuttavia, ciò rappresenta un *single point of failure* in caso di guasti di varia natura o attacchi informatici. Si potrebbe pensare di diffondere la topologia di rete conosciuta, propagandola a tutti i peer interessati, ma ciò comporterebbe un consumo crescente di memoria e un intenso traffico di rete. Una buona alternativa consiste nel fornire ad ogni peer una conoscenza parziale, ma che complessivamente diventerà globale.

Ciascun peer è identificato da un valore a 160 bit, risultato dell'applicazione di una funzione hash sul proprio IP.

$$IP_{node} \rightarrow function_{hash}(IP) \rightarrow hash_{IP}$$

Avendo una propria identità, ciascun peer possiede una posizione ed è in grado di determinare quali sono i propri vicini. Si utilizza una metrica di distanza basata sul operatore XOR applicato bitwise, e successivamente interpretato come intero.

$$d(peer_X, peer_Y) = to\_int(hash_{peer_X} \oplus hash_{peer_Y})$$

Ne consegue che, leggendo la sequenza ottenuta, partendo dal bit più significativo a sinistra e procedendo verso destra, più due sequenze sono simili tra loro, minore sarà il valore ottenuto(*common the prefix, smaller the distance*). L'obiettivo è effettuare una richiesta ad un nodo conosciuto, e che possiede il dato di interesse, oppure ad un nodo che è "vicino" al nodo che lo possiede. Per evitare che ciascun peer debba conoscere la topologia completa, è sufficiente che ognuno abbia cognizione di un certo sottoinsieme degli altri presenti, adottando in contemporanea una strategia di routing che garantisca la convergenza. Per garantire la convergenza dell'algoritmo, ogni nodo non può avere una conoscenza "random" degli altri nodi: ognuno di essi deve conoscere almeno un altro nodo in ogni sottoalbero di cui non fa parte. Per chiarire meglio il concetto è possibile fare riferimento alla figura 3.3, in cui vengono utilizzati ID con lunghezza pari a 4 per semplicità: il nodo N1 è identificato dall'ID 0100 ed è in grado di identificare almeno un nodo in ognuno dei 3 sottoalberi di cui non fa parte, ossia quelli con radice 1, 00, 011 e 0101. In figura 3.4 viene mostrata l'esplorazione da parte di un nodo: il nodo N1 (verde) vuole contattare il nodo N2 (rosa) ma non ha una connessione diretta con esso, quindi contatta il nodo N3 (viola, di cui ha il link) nel sottoalbero 1, il quale ha un contatto nel sottolbero 11 e così via. La convergenza è garantita in tempo logaritmico e le richieste non avvengono in modo ricorsivo, ma iterativo.

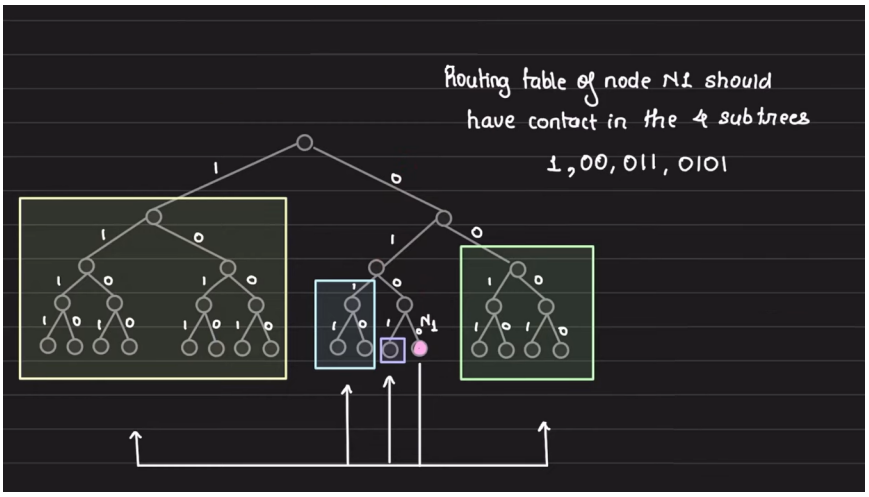


Figura 3.3: Sottolberi, nei quali un nodo, ha almeno un contatto diretto per ciascuno di essi

Infine, per garantire *fault tolerance*, un dato sottoalbero, invece di memorizzare un unico nodo di riferimento, può memorizzare una lista (*k-bucket*), con numero massimo k di nodi scelto a piacere, e tenerla aggiornata opportunamente grazie al monitoraggio dei messaggi ricevuti.

3.3 Corestore

Corestore permette di raccogliere un numero variabile di core in una sorta di repository, gestendoli in maniera centralizzata e ottimizzandone l'accesso. Viene utilizzato come building block da moduli più avanzati, come `Hyperdrive`.

Ciascuno store non ha un proprio identificativo, ma viene utilizzata la *chiave pubblica* di un core al suo interno, scelto come riferimento. Esso conterrà tutte le chiavi degli altri core esistenti nello store. É possibile associare dei nomi a ciascun core, visibili ed utilizzabili solo a livello locale, ed utilizzati per generare le chiavi in modo deterministico.

3.4 Hyperbee

Hyperbee fornisce l'implementazione di una struttura dati **B-tree** append-only, che sfrutta un core come unità di base ed utilizza una tecnica con indice

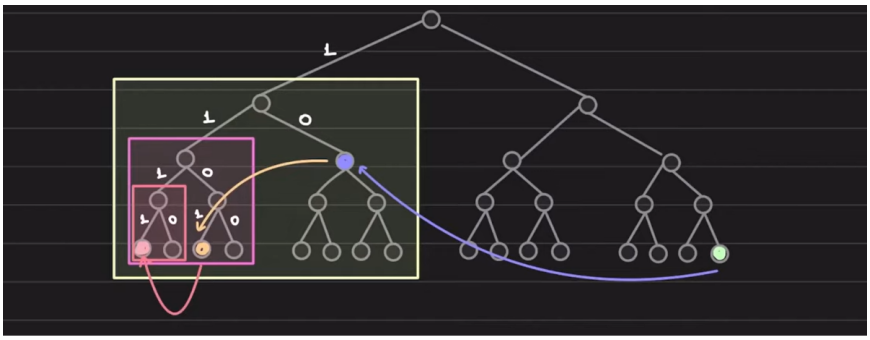


Figura 3.4: Ricerca di un nodo, svolta iterativamente

chiamata *embedded indexing*. È utilizzabile principalmente come un database chiave-valore e permette di effettuare operazioni massive. Poichè è basato su `Hypercore`, è disponibile il versionamento e il confronto tra varie versioni, oltre che garantire solamente al creatore la sua modificabilità.

3.5 Hyperdrive

Hyperdrive implementa un file system distribuito per reti P2P. Per il suo funzionamento sono necessari due sotto-componenti: un'istanza `Hyperbee`, usata per i metadati del contenuto, e un'istanza `Hypercore`, usata per memorizzare il contenuto effettivo. Anch'esso, come le sottostrutture che lo compongono, permette di usufruire di tecniche di versionamento.

3.6 Utilizzo del framework Hypercore

Recentemente gli sviluppatori del framework Hypercore hanno ulteriormente esteso i moduli disponibili, facendo nascere l'ecosistema **Holepunch**. Le componenti descritte nei capitoli precedenti costituiscono, tuttavia, gli elementi principali di questa suite. In questo capitolo verranno discusse, per ogni modulo, le varie caratteristiche, evidenziandone quelle salienti. I moduli sono utilizzabili tramite la libreria disponibile in linguaggio `Javascript`, con il supporto dell'ambiente di esecuzione `Node.js`. Per ognuno di essi, l'autore ha realizzato script che mostrano le funzionalità, e nel capitolo ne viene data una breve descrizione. Per ulteriori dettagli si rimanda al codice sorgente.

3.6.1 Hypercore

Hypercore è l'elemento principe del framework, sul quale la maggior parte degli altri moduli vengono composti. Il contenuto di un core può essere riconducibile ad un array composto da un numero variabile di array di byte, in quanto sono presenti molteplici blocchi concatenati tra loro, e ciascuno ha contenuto indipendente dagli altri. Il creatore, generalmente l'owner del core, può apportare modifiche ad esso in qualunque momento; di conseguenza per consentire letture in modo asincrono, è utile introdurre un layer di memorizzazione, sfruttando il disco locale, la memoria RAM (attraverso un'astrazione fornita dalla libreria), o anche appoggiandosi ad un unità presente in rete. Un determinato core è un componente locale, che esiste in un determinato dispositivo, nel quale la presenza di una determinata *chiave privata* consente di apportare aggiunte al contenuto. Tali aggiunte possono essere apportate tramite il metodo `append`, che opera in modo atomico. È possibile operare in maniera concorrente su un'istanza locale, aprendo su di essa una nuova *sessione*. Le *sessioni* operano utilizzando la tecnica di *reference counting*, pertanto, finchè ci saranno *sessioni* locali aperte, continuerà ad essere aperta anche l'istanza locale. Per poter leggere il core localmente è possibile accedere ad uno specifico blocco, tramite indice, utilizzando il metodo `get`, oppure accedere a più blocchi sequenziali tramite il metodo `createReadStream`. Quest'ultimo metodo fornisce anche la possibilità, tramite un parametro booleano `snaphot`, di effettuare un lettura "live", in modo che eventuali variazioni sul core locale vengono catturate dallo stream e rese disponibili, altrimenti si legge lo stream fino ad un determinato istante temporale, ossia quello in cui viene aperto il flusso di streaming. Per quanto riguarda la lettura da dispositivi remoti, ad un core viene associata una chiave univoca, detta *discovery key* (derivata dalla propria *chiave pubblica*), la quale permette ad altri dispositivi in possesso di essa di cercare, mediante il componente `Hyperswarm`, il core, instaurarne una connessione e replicarne il contenuto. Una tecnica alternativa consiste nella comunicazione tra peer tramite l'uso di socket: in questo caso ogni peer è identificato da una chiave pubblica ed è possibile impostare un peer a fungere da server, mentre l'altro, fungendo da client, si connetterà al primo tramite la chiave e secondo la logica scelta avverrà lo scambio di dati. Quest'ultima tecnica è molto più complicata da gestire rispetto all'uso di `Hyperswarm`, che semplifica anche le politiche di replicazione di un core. Per alleggerire le comunicazioni di rete è previsto l'uso di default, per i vari metodi, della tecnica di *sparse downloading*: essa consente di scambiare tra vari peer solo i metadati che caratterizzano un certo core; nel momento in cui si richiede di leggere uno specifico blocco, se non lo si ha localmente, verrà richiesto e scarica-

to comunicando in rete. È possibile infine, tra le varie funzionalità, operare con snapshot oppure effettuare eliminazioni parziali. Il metodo `snapshot` consente di prelevare lo stato locale di un determinato core, ed associare tale stato ad una copia: successive modifiche apportate al core originale non saranno riflesse sul core copia, a meno di richiamare su di esso il metodo `update`, aggiornando quindi i metadati e riallineando le due versioni. Il metodo `truncate` consente di eliminare una parte dei blocchi di un core, troncando la lista ad un dato indice. L'operazione avviene *in-place* e viene assegnato un nuovo *forkId*, visibile nelle info del core.

3.6.2 Hypercore - script realizzati

Nel caso sia necessario garantire la persistenza dei dati presenti nel core, è possibile specificare il percorso del file system locale in cui andrà memorizzato il contenuto. Nel caso in cui non sia di interesse garantire la persistenza, è possibile utilizzare la libreria `random-access-memory`, che permette di memorizzare il contenuto all'interno della memoria RAM, per poi rimuoverlo al termine dell'esecuzione del programma. Questa funzionalità è disponibile anche per gli altri componenti che si basano su `Hypercore`.

- *hypercore-append-stampa-info*: mostra come si effettua l'inserimento di uno o più blocchi ad un core e come si stampa un log informativo sulla struttura del core in questione;
- *hypercore-sessione.js*: mostra come si possano inserire e leggere, sia puntualmente che tramite uno stream, blocchi creati da molteplici sessioni aperte sullo stesso core;
- *hypercore-readstream-snapshot-true* e *hypercore-readstream-snapshot-false*: mostrano un caso specifico nel quale, durante la lettura di un core per il tramite di uno stream, il core letto viene modificato aggiungendo un nuovo blocco; nel caso del primo file, tale modifica viene catturata ma non gestita durante la lettura dello stream, mentre nel caso del secondo file viene anche gestita;
- *hypercore-socket.js*: mostra come sia possibile generare una chiave ed utilizzarla per connettere un peer ad un altro, tramite l'utilizzo di socket, senza ricorrere a strutture esterne;
- *hypercore-snapshot-no-update* e *hypercore-snapshot-si-update*: mostrano un esempio di creazione di una copia one-shot a partire da un determi-

nato core; nel caso del primo file, il core di origine viene popolato di un nuovo contenuto ma non viene effettuato l'aggiornamento del core copia, pertanto non vengono riflesse le variazioni su di esso, mentre nel caso del secondo file si compie l'aggiornamento e la lettura del nuovo contenuto da parte del core copia viene eseguita con successo;

- *hypercore-truncate*: mostra come eliminare una parte del contenuto di un determinato core.

3.6.3 Corestore

Corestore è un modulo strettamente correlato al modulo `Hypercore` e funge da contenitore per molteplici core, astraendo dalle logiche di storage e replicazione. Partendo da uno store, è possibile istanziare direttamente un core al suo interno, fornendogli un nome arbitrario. Quest'ultimo è un concetto locale che viene utilizzato per generare dinamicamente le chiavi relative al core. In alternativa, gli accessi ad un core possono avvenire tramite la *chiave pubblica*, come descritto nel capitolo precedente. Per scambiare il contenuto di uno store tra i vari peer, è previsto l'uso di socket dirette tra essi, oppure l'uso di `Hyperswarm`: la differenza principale rispetto allo scambio presentato in `Hypercore`, consiste nel fatto che è necessario trasferire le chiavi di tutti i core presenti, utilizzando un core di appoggio, e che una volta partita la replicazione, si tenta di trasmettere tutti i core presenti nello store, assumendo che il peer ricevente abbia sufficiente memoria per gestirlo, in caso contrario la replica fallisce. Infine, poichè uno store può essere usato da molteplici utilizzatori, può essere utile adottare una suddivisione logica attraverso l'uso di `namespaces`, in modo da prevenire conflitti legati al nome associato al core.

3.6.4 Corestore - script realizzati

- *corestore-creazione-sessioni.js*: mostra la creazione di un nuovo store, e di un nuovo core, specificando un nome arbitrario. Vengono create ulteriori sessioni, sia tramite nome che tramite *chiave pubblica*, e poi vengono eseguiti degli inserimenti;

- *corestore-replicazione-writer.js* e *corestore-replicazione-reader.js*: mostrano come replicare, attraverso l'ausilio di `Hyperswarm`, l'intero contenuto di uno store, compreso il core di appoggio. È possibile inserire un contenuto in un determinato core piuttosto che in un altro, secondo una certa logica.

Il peer che sarà destinatario della replicazione avrà accesso alle varie chiavi e potrà accedere al contenuto dei vari core secondo un'opportuna logica;

- *corestore-namespace.js*: mostra come creare differenti `namespaces` ed un nuovo core in ciascuno di essi. Si assegna, nei due `namespaces` distinti, lo stesso nome al relativo core ma si creeranno due core differenti.

3.6.5 Hyperswarm

Hyperswarm è il modulo che si occupa principalmente di facilitare la comunicazione tra peer e rendere trasparente la gestione di rete sottostante, inclusi tentativi di ri-connessione in caso di *failure* di rete. Esso si avvale di un componente che opera a più basso livello, chiamato `HyperDHT`, il quale permette di stabilire connessioni end-to-end (criptate con protocollo `Noise`) tramite tecniche di `holepunching`, senza dover esplicitare l'indirizzo IP del peer al quale ci si vuole connettere, ma tramite la *chiave pubblica* che lo identifica, detta anche *Noise public key*. La tecnica di `holepunching` permette di stabilire connessioni dirette tra due peer anche se questi sono mascherati da un processo di NAT, come nel caso di una rete domestica. Per poter operare, inizialmente si rende necessario contattare uno dei seguenti nodi di `bootstrap`:

- node1.hyperdht.org:49737
- node2.hyperdht.org:49737
- node3.hyperdht.org:49737

L'interfaccia di `HyperDHT` impone di definire quali peer devono operare come server (accettano connessioni in entrata), e quali come client (effettuano connessioni in uscita). Per rendere ancora più semplice il dialogo, invece di utilizzare come identificatore la chiave associata al peer (*Noise public key*), con `Hyperswarm` è possibile sfruttare un *topic*, cioè un identificativo di 32 byte, che rappresenta un "argomento/oggetto di interesse comune". Un *topic*, tipicamente corrispondente alla *discovery key* del core che si vuol rendere disponibile, viene annunciato da un determinato peer, che fungerà da "server", e viene aggiunta allo swarm un'associazione tra l'identificativo del core e chi lo possiede. Successivamente, altri peer "client" che vogliono richiedere il topic in questione, possono effettuare una query verso lo swarm, connettersi ad un peer "server" in possesso del topic e richiedere il core. Tuttavia, essendo i peer "pari" tra loro, la gerarchia nello swarm è elastica: ciascun peer è libero, al momento dell'operazione di join

di un topic sullo swarm, di specificare se unirsi ad esso solo come client, solo come server o entrambi.

3.6.6 Hyperswarm - script realizzati

- *hyperswarm-server-dht.js* e *hyperswarm-client-dht*: mostra come utilizzare `HyperDHT` per mettere in comunicazione due peer, secondo una logica di tipo client-server;
- *hyperswarm-server-topic-sender.js* e *hyperswarm-client-topic-receiver.js*: mostra come utilizzare un core come `topic` comune per mettere in contatto due peer tramite `Hyperswarm`, per poi ricevere le modifiche del core in entrata, tramite uno stream live.

3.6.7 Hyperbee

Hyperbee è il modulo che implementa un database chiave-valore distribuito, utilizzando come sottostrutture un'istanza `Hypercore` e la struttura dati **B-Tree**. Ad una determinata chiave può corrispondere un contenuto binario. Tale contenuto è memorizzato nei blocchi dell'istanza `Hypercore` e la struttura dati **B-Tree** memorizza i riferimenti ad essi. I progettisti di `Hypercore Protocol` hanno scelto di utilizzare questa struttura dati in quanto ha proprietà tali da effettuare ricerche in tempo logaritmico, caratteristica fondamentale per ridurre la latenza nell'uso di un database chiave-valore. Poichè si utilizza `Hypercore`, valgono le stesse regole di scrittura solo tramite possesso di *chiave privata*, replicazione tramite *discovery key*, gestione dei metadati e richiesta differita dell'effettivo contenuto. Inoltre è possibile mutare il contenuto di un'istanza `Hyperbee` tramite 3 operazioni principali, rappresentate dai metodi:

- `put` : inserimento di una coppia chiave-valore;
- `del` : eliminazione di una coppia chiave-valore;
- `get` : estrazione del valore di una coppia, data la chiave.

Per ottimizzare i tempi di inserimenti ed eliminazioni, è possibile utilizzare un ottimizzatore `batch` che, a differenza dell'uso delle primitive "base", opera in modo **atomico**. Il `lock` dell'istanza, attraverso l'ottimizzatore, può essere acquisito in modo esplicito, oppure tramite inserimenti o eliminazioni. Nello specifico, in questi ultimi due casi, viene acquisito alla prima operazione `put` o `del` che viene invocata dall'ottimizzatore. Il `lock` viene rilasciato solo al

momento dell'invocazione del metodo `flush` e finchè è acquisito, qualunque variazione apportata da un entità al di fuori del `batch` sarà impostata in uno stato di *pending* e sarà applicata all'istanza solo successivamente all'operazione di `flush`. I metodi `put` e `del` possono essere ulteriormente vincolati, anche utilizzando la modalità `batch`, passando ad essi una funzione detta `Compare and Swap (CAS)`. Essa prevede la ricezione di due argomenti:

- `prev` : rappresenta la entry attualmente presente data una determinata chiave;
- `next` : rappresenta la nuova entry che si vuole sostituire a quella rappresentata da `prev`.

Partendo da queste 2 entry, è possibile definire arbitrariamente una logica secondo la quale l'operazione di `put` o `del` debba fallire oppure no; ad esempio è possibile specificare che un inserimento può avvenire solo se il valore della nuova entry supera una certa soglia. Questa funzionalità permette all'utilizzatore della libreria di implementare le policy di gestione che ritiene più utili ai suoi scopi. Oltre a leggere il contenuto di un singolo blocco tramite il metodo `get`, è possibile leggere l'intero contenuto dell'istanza attraverso il metodo `createReadStream`. In entrambi casi, la entry letta viene rappresentata come una tupla composta da:

- `seq` : indice incrementale del core in cui è stata inserita la chiave;
- `key` : chiave specificata nella coppia chiave-valore;
- `value` : valore specificato nella coppia chiave-valore.

È possibile leggere lo stream di dati utilizzando altri 2 metodi che forniscono ulteriori informazioni di dettaglio, `createHistoryStream` e `createDiffStream`. Il primo permette di estrarre lo storico delle operazioni effettuate sull'istanza `Hyperbee`: ad ogni entry è associato, oltre a `seq`, `key` e `value`, anche il campo `type`. Esso può assumere il valore `put` o `del` in modo da evidenziare se nella sequenza di operazioni compiute, una entry sia stata eliminata. Tale entry non è comunque recuperabile attraverso il metodo `get`. Il secondo, invece, permette di confrontare la versione attuale ([ultimo valore di `seq` presente nell'istanza] + 1) dell'istanza con un'altra precedente, restituendo un elenco di entry che sono presenti nella versione più recente ma non in quella specificata, e viceversa. Quindi se due entry sono causalmente equivalenti (hanno lo stesso valore di `seq`), non verranno considerate nell'output.

Infine, tra le varie funzionalità offerte da `Hyperbee`, c'è la possibilità di creare più sotto-database logici attraverso il metodo `sub`, specificando un nome arbitrario detto `sub-prefix`. Alla chiave di ogni nuova entry inserita nel sotto-database, sarà preposto il `sub-prefix` di appartenenza. Questa funzionalità è utile per creare dei `namespaces` nello stesso `Hyperbee`.

3.6.8 Hyperbee - script realizzati

- *bee-writer-batch.js* e *bee-writer-nobatch.js*: mostra come si possano generare nuove entry, inserirle all'interno dell'istanza `Hyperbee`, e renderle disponibili ad altri peer tramite `Hyperswarm`. Vengono inserite 10000 entry con chiave del tipo *key1,...,k10000*. Nel caso del file in cui si utilizza il meccanismo dell'ottimizzatore `batch`, si tentano di inserire anche alcune entry, utilizzando direttamente l'istanza `Hyperbee` mentre il `batch` ha acquisito il `lock` su di essa. Tali entry saranno effettivamente inserite solo dopo che esso avrà rilasciato il `lock`;

- *test-bee-writer-batch.js* e *test-bee-writer-nobatch.js*: per verificare la differenza di performance, sono stati realizzati dei test in cui vengono creati ed inseriti vari gruppi di blocchi, ciascuno con cardinalità maggiore del precedente, tenendo contemporaneamente monitorati i relativi tempi di esecuzione. In figura 3.5 è riportata la configurazione del PC utilizzato, mentre in tabella 3.1 e figura 3.6 vengono mostrati i risultati ottenuti;

- *bee-reader.js*: mostra come si possano recuperare le varie chiavi, generate in precedenza da un bee-writer, contattabile tramite il *topic* pubblicato su `Hyperswarm`;

- *bee-sub-compareAndSwap*: mostra come creare dei sotto-database, ad esempio "pari" e "dispari", visualizzarne il contenuto, ed effettuare modifiche ad essi specificando funzioni di `compare-and-swap` definite arbitrariamente;

- *bee_put-get-del_stream-read.js*: mostra un flusso di inserimenti ed eliminazioni effettuati su un'istanza `Hyperbee`. Vengono utilizzati stream di lettura, sia del contenuto attuale che quello storico, e vengono effettuati confronti tra versione attuale e versioni precedenti dell'istanza, in modo da evidenziare le divergenze tra esse.

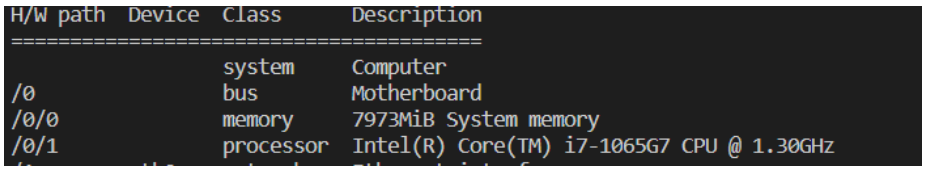


Figura 3.5: Caratteristiche tecniche del PC sul quale sono stati eseguiti i test

N. blocchi	t_batch (ms)	t (ms)
100000	110	296
200000	241	399
300000	383	956
400000	704	878
500000	615	2004
600000	844	3137
700000	1024	2332
800000	2929	5546
900000	7321	5295
1000000	5478	11183
1100000	6301	39455
1200000	5408	61090
1300000	9456	166121

Tabella 3.1: Tempi di esecuzione batch/no batch a confronto variando il numero di inserimenti.

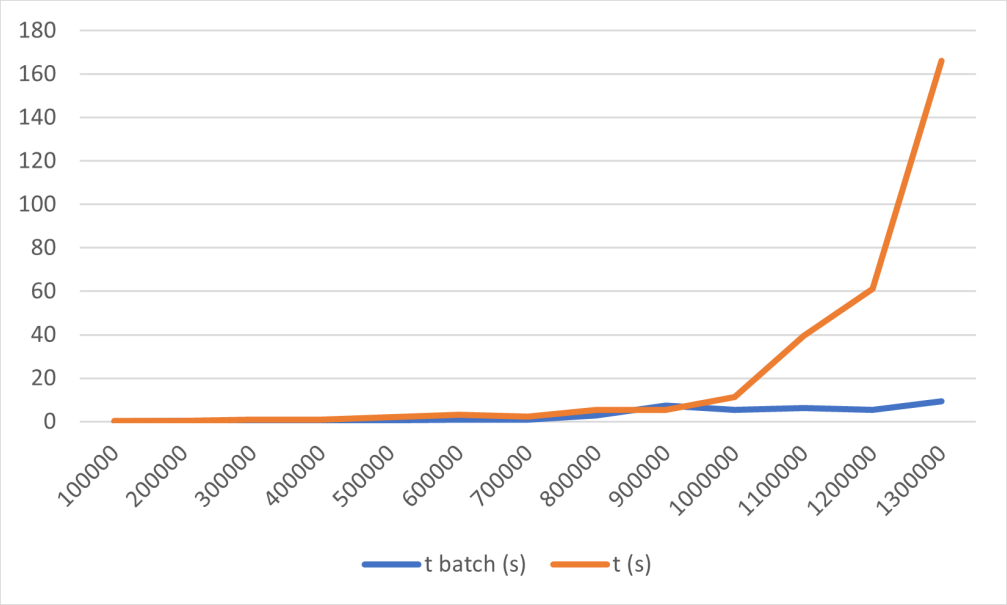


Figura 3.6: Andamento dei tempi di esecuzione al variare della cardinalità dei gruppi di blocchi

3.6.9 Hyperdrive

Hyperdrive è il modulo che realizza un file system distribuito, utilizzando un'istanza `Corestore` che racchiude una struttura `Hyperbee` e una `Hypercore`. La struttura `Hyperbee` permette di mantenere informazioni sull'organizzazione del contenuto e può essere facilmente trasferita, in quanto di dimensioni inferiori rispetto all'altra, la quale memorizza il contenuto effettivo.

`Hyperdrive` rimane un'astrazione di file system a sé stante, logicamente separata dal file system locale del dispositivo su cui è in esecuzione. Attualmente è prevista la possibilità di utilizzo solo tramite la API fornita dalla libreria, non è disponibile il montaggio ed uso in modo stand-alone. Per poter interagire con il file system locale, è previsto l'utilizzo di `Localdrive`, un tool che permette di effettuare una mappatura e l'accesso al file system locale. Inoltre un tool aggiuntivo, chiamato `Mirrordrive`, permette di mettere in comunicazione l'istanza locale di `Localdrive` con quella locale di `Hyperdrive`, in modo che aggiornamenti sull'istanza di `Hyperdrive` vengano riflessi sull'altra e viceversa. Al fine di propagare ad altri peer le modifiche che si apportano ad un determinato file collegato ad `Hyperdrive`, è essenziale utilizzare il componente `Hyperswarm`.

Similarmente a quanto già visto per gli altri moduli, non è prevista la possibilità di accesso diretto ai drive appartenenti a diversi peer: solo il creatore può apportare modifiche e altri peer possono leggere solo se in possesso della *chiave pubblica*. Analogamente all'interfaccia presente in `Hypercore`, è possibile leggere e scrivere il contenuto di un file attraverso uno stream di dati; tuttavia, sono disponibili 4 operazioni principali che agiscono sul file system di `Hyperdrive`:

- `put` : creazione di un record;
- `get` : recupero di un record;
- `del` : eliminazione dei metadati di un record;
- `clear` : eliminazione del contenuto di un record.

Un record, anche noto come entry, è composto dai seguenti elementi:

- `seq` : elemento che rappresenta un indice incrementale;
- `key` : elemento che che rappresenta il path del file;
- `value` : elemento che contiene i riferimenti, sotto forma di indici, per accedere al contenuto del file descritto dal path di `key`.

Il contenuto è memorizzato, come detto in precedenza, all'interno di un core e, per agevolare l'accesso tramite indicizzazione, viene utilizzata una struttura dati ausiliaria detta **Hyperblob**, che funge da wrapper al core e permette di accedere ad un determinato contenuto tramite un riferimento, composto da una tupla di 4 elementi, detta anche `ID`:

- `byteOffset` : offset da utilizzare nel blocco iniziale;
- `blockOffset` : blocco dal quale iniziare la lettura;
- `blockLength` : numero di blocchi da leggere;
- `byteLength` : numero totale di byte da leggere.

Ad esempio, se si istanzia un nuovo blob e si inserisce la stringa *hello world*, essa avrà il riferimento composto come segue:

- `byteOffset` : 0
- `blockOffset` : 0
- `blockLength` : 1

- `byteLength` : 11

È possibile accedere all'intero record tramite il metodo `entry`, utilizzando la stringa `ID` del blob, oppure solo al contenuto tramite il metodo `get`, specificando il percorso del file (la key del record). Infine, per quanto riguarda il versionamento, è disponibile il metodo `checkout`, che permette di ottenere una determinata versione read-only del drive, ed il metodo `diff`, che permette di mettere a confronto versioni differenti del drive, in modo simile a quanto visto in `Hyperbee`.

3.6.10 Hyperdrive - script realizzati

- *drive-writer.js*: mostra come utilizzare `Localdrive` e `Mirrordrive` per replicare il contenuto di una cartella locale verso i peer connessi ad `Hyperswarm`;
- *drive-reader.js*: mostra come leggere gli aggiornamenti pubblicati da un writer e memorizzarli in una cartella locale;
- *bee-reader.js*: mostra come leggere gli aggiornamenti pubblicati da un writer, mostrando in questo caso solo i metadati;
- *drive-write-and-read.js*: mostra come leggere il contenuto da file system locale, memorizzarlo all'interno dell'istanza `Hyperdrive`, recuperarlo tramite l'uso di stream oppure tramite accesso con stringa del path o `ID` del blob;
- *drive-diff.js*: mostra come si possano recuperare versioni precedenti dell'istanza `Hyperdrive`, operare eliminazioni e fare confronti tra le versioni.

Capitolo 4

Confronto con altri file system P2P

Nel seguente capitolo viene illustrata una breve panoramica su come altri sistemi distribuiti P2P, nello specifico Bittorrent, Swarm, SAFE, Storj e IPFS affrontano i seguenti aspetti:

- Architettura di rete;
- Gestione dei file;
- Sicurezza delle informazioni (CIA);
- Incentivi.

4.1 Architettura di rete

Ogni file system distribuito ha necessità di costruire una rete logica di overlay per permettere la comunicazione tra i vari peer. Principalmente ci si basa su DHT come Kademlia, e si possono avere 2 tipi di rete overlay: una per il peer discovering, l'altra per lo scambio di dati. Ciascun peer deve rendere identificabile il proprio nodo tramite un particolare ID, quasi sempre tramite una chiave auto-generata, oppure la relativa versione hashed. Mentre Hypercore, al pari di Bittorrent, utilizza una DHT per il peer discovering e successivamente una rete di overlay non strutturata, nella quale non esiste il concetto di distanza e

organizzazione di "vicini" per lo scambio dati; IPFS, Swarm e SAFE utilizzano DHT per entrambi gli scopi. Ulteriori scelte strategiche sono applicate da ciascuna rete: SAFE, Swarm e Storj strutturano la rete in sotto-sezioni, ciascuna con propria autonomia organizzativa; IPFS utilizza una DHT ma, data la necessità, ogni peer mantiene una connessione con ciascun peer col quale viene a contatto e occorre applicare periodicamente un algoritmo di pruning per ripulire connessioni datate.

4.2 Gestione dei file

Il modo in cui file vengono manipolati e gestiti è chiaramente l'aspetto più variabile e caratterizzante di un file system distribuito. Ci sono due aspetti principali in ognuno di essi: come i file vengono cercati (look-up) e come questi vengono memorizzati (storage).

Hypercore utilizza una specifica DHT in cui si assume che ciascun vicino possegga almeno una parte di un file. IPFS può utilizzare una tecnica "opportunistica" con la quale si interrogano i vari peer senza avere una conoscenza pregressa sulle parti da essi possedute. Altri possono utilizzare componenti centralizzate, come i *satelliti* in Storj e i *tracker* in Bittorrent.

Molteplici sono le tecniche di memorizzazione dei file, tuttavia, si tende a privilegiare l'immutabilità e la non-cancellazione del contenuto. Bittorrent, IPFS e Hypercore memorizzano integralmente i dati di un file in uno specifico nodo e successivamente lo dividono in pezzi, detti *chunk*, durante lo scambio; Swarm, SAFE e Storj dividono direttamente il file in *chunk*, che vengono memorizzati in nodi potenzialmente diversi, influenzando di conseguenza il processo di look-up . In particolare Storj impiega una tecnica di *erasure encoding* tale da ricostruire il file intero senza avere necessità di recuperare tutti i *chunk*.

L'utilizzo di *chunk*, permette di sfruttare al meglio le capacità di storage di ogni peer evitando la saturazione di un nodo in particolare, tuttavia è necessario avere la "volontà" del peer, assieme all'aggiunta di metadati necessari alla ricostruzione e al maggior traffico di rete.

4.3 Sicurezza delle informazioni (CIA)

In un sistema distribuito i dati sono suddivisi su molti nodi autonomi che tipicamente si uniscono alla rete in modo non supervisionato, cioè senza controlli stringenti a monte sulla loro affidabilità. Pertanto è necessario valutare aspetti

che garantiscano l'accesso controllato, l'integrità e la disponibilità diffusa e a lungo termine dei dati gestiti (*long-term availability*).

4.3.1 Confidentiality (Accesso controllato al dato)

La crittografia è la tecnica più diffusa per proteggere i dati da accessi non autorizzati, in quanto non si hanno informazioni certe di *trust* (affidabilità) che caratterizza ciascun peer. La tecnica di *storage chunking* fornisce un ulteriore grado di protezione in quanto è necessario recuperare molte parti da differenti peer per identificare correttamente il contenuto. Come detto nel capitolo precedente, questa tecnica non viene applicata in Hypercore. Hypercore si basa su crittografia assimmetrica, nella quale la chiave pubblica (diffusa in maniera esterna rispetto alla rete di Hypercore) di una determinato file viene usata per ricavare la chiave di discovery. Quest'ultima è utilizzata per trovare i peer che possiedono il dato cercato e successivamente recuperarlo e decriptarlo. Bittorrent e IPFS non impiegano alcuna tecnica di controllo degli accessi, mentre Storj, può far affidamento sull'entità dei nodi *satelliti*, ciascuno dei quali può regolare gli accessi ai file di propria competenza.

4.3.2 Integrity (Integrità del dato)

Per assicurare che il contenuto di un file non sia stato manipolato, è possibile applicare un algoritmo di hashing. Dato un hash, tipicamente fornito tramite canali esterni, e l'algoritmo di hashing utilizzato, l'integrità del contenuto può essere verificata applicando l'algoritmo su di esso e confrontando l'hash ottenuto con quello atteso. Hypercore e Bittorrent memorizzano il digest hash in appositi file di metadata. IPFS, Swarm e SAFE utilizzano tecniche di hashing per realizzare il content-addressing, quindi l'indirizzo è determinato direttamente dal contenuto e il controllo di integrità avviene di conseguenza. Storj utilizza i nodi *satellite* per effettuare audit di integrità sui vari nodi di storage e gestisce meccanismi di *reputation* in base agli esiti rilevati.

4.3.3 Availability (Disponibilità del dato)

Ciascun peer può essere soggetto a guasti di natura tecnica oppure ad attacchi informatici che possono compromettere, temporaneamente o permanentemente, la disponibilità dei dati ivi memorizzati. Pertanto è fondamentale replicare i contenuti su più nodi. La replicazione può essere effettuata in modo attivo,

passivo o attraverso caching. Queste ultime due modalità non richiedono alcuna tecnica di coordinazione. SAFE, Swarm, IPFS sfruttano principalmente caching, grazie ad appositi manager che memorizzano i contenuti in modo ridondante man mano che vengono soddisfatte le varie richieste. Questi tre possono applicare, in alternativa, meccanismi attivi di incentivazione per assicurare *long-term availability*, mentre Storj, sfruttando la tecnica di *erasure encoding*, evita di dover recuperare tutti i *chunks* altrimenti richiesti. Hypercore e Bittorrent si differenziano da tutti i precedenti, in quanto la replicazione è fatta in modo passivo ed il dato è disponibile solo se un peer decide di mantenerlo tale.

4.4 Incentivi

È possibile utilizzare meccanismi di incentivazione per stimolare i peer a tenere un comportamento "corretto" nei confronti degli altri membri della rete. In Hypercore non è stato previsto alcun meccanismo del genere, tuttavia altri file-system distribuiti li adottano. Al fine di promuovere è possibile monitorare i dati scambiati con quelli ricevuti, per poi ridurre, come fa Bittorrent, la capacità di reperire dati ai peer che sono meno disponibili (*free-riding*). Per migliorare la disponibilità del dato a lungo termine, è possibile prevedere che sia fornito un'ulteriore incentivo (cripto-valuta oppure aumenti di banda) a quei peer che durante la partecipazione nello scambio di file, si rendono disponibili anche a memorizzarli in locale per poi renderli consultabili nel tempo.

4.5 Sfide aperte nel mondo dei file system distribuiti

4.5.1 Gestione degli accessi e confidenzialità

Le architetture generalmente adottate nei file system distribuiti forniscono meccanismi di gestione degli accessi ma sono più adatte alla diffusione di dati pubblici, come articoli e ricerche piuttosto che dati personali. Inoltre, bisogna considerare che non esistono metodi efficaci per eliminare rapidamente dati personali che vengono diffusi in maniera non autorizzata.

4.5.2 Anonimato

Dato un contenuto, il creatore, il nodo che lo ospita o l'utente che lo richiede possono essere mantenuti anonimi. Il principio di anonimato è stata una chiave di successo delle prime reti P2P, come FreeNet, ma i meccanismi di incentivazione, che appunto necessitano di avere un riferimento all'entità da compensare, obbligano ad utilizzare almeno uno pseudonimo, non garantendo una minimizzazione del rischio di una possibile censura da parte di terzi.

Capitolo 5

Conclusioni

In questo report è stata esposta una panoramica sulla nascita delle reti P2P e la loro rapida evoluzione che in poco più di 10 anni ha apportato notevoli migliorie in termini di affidabilità e prestazioni. Successivamente si è provveduto a fornire una descrizione dell'architettura Hypercore, divisa in moduli utilizzabili in base alle necessità. È stata presentata una descrizione di due componenti essenziali al suo funzionamento: la struttura dati Merkle Tree e la hash table distribuita Kademlia. Proseguendo, si è provveduto ad esaminare le funzionalità e gli utilizzi dei vari moduli, realizzando contestualmente numerosi esempi utilizzando la API Javascript per mostrarne il funzionamento. Infine, l'ultima parte, di carattere più teorico, ha posto l'accento sul confronto tra Hypercore e altri file system distribuiti. Nonostante il vasto ecosistema e le promettenti aspettative, Hypercore è da ritenersi un progetto in via di sviluppo, non ancora idoneo all'utilizzo nel mondo industriale poichè l'interfaccia è suscettibile ad evoluzioni, anche radicali, come già successo nel 2022. Inoltre, la documentazione reperibile su Internet sia sul funzionamento che sulle librerie disponibili è scarsa e non eccessivamente approfondita, ma questo sembra essere comune anche agli altri file system, poichè buona parte della letteratura scientifica è focalizzata su IPFS. Questa condizione rende queste reti, nonostante il grande potenziale, poco developer-friendly, rendendo più ripida la curva di apprendimento. In conclusione, le reti peer-to-peer possono essere la chiave per ottenere enormi benefici in termini di scalabilità e latenza ma si rendono necessarie ulteriori protopazioni da applicare a casi d'uso reali, che una volta validate possono portare ad un miglioramento di esperienza tale per cui la documentazione ed il materiale a supporto possano essere stabili nel tempo.

Bibliografia

[1] Documentazione hypercore, 2023. https://docs.holepunch.to/.

[2] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: simpler, smaller, fast as md5. Cryptology ePrint Archive, Paper 2013/322, 2013. https://eprint.iacr.org/2013/322.

[3] Erik Daniel and Florian Tschorsch. IPFS and friends: A qualitative comparison of next generation peer-to-peer data networks. *CoRR*, abs/2102.12737, 2021.

[4] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[5] E Mykletun, M Narasimha, and G Tsudik. Providing authentication and integrity in outsourced databases using merkle hash trees. uci-sconce technical report [r/ol](2003).