StorJ
Decentralized Cloud Storage Network

Tommaso Mandoloni
tommaso.mandoloni@studio.unibo.it

December 2020

Indice

Abstract

Questo progetto mira a discutere e comprendere i concetti principali di come funziona la tecnologia StorJ. In questi termini, lo scopo è fornire una documentazione chiara e dettagliata sulla sua architettura e sul suo modo di operare come sistema di rete decentralizzato di cloud storage. Il cloud storage decentralizzato rappresenta un'importante evoluzione nell' efficienza e nell'economia dello storage su larga scala. L'eliminazione del controllo centrale consente agli utenti di archiviare, condividere e gestire i dati senza fare affidamento su provider di archiviazione esterni. La decentralizzazione dello storing su cloud limita il rischio di guasti o interruzioni del flusso dati, aumentando contemporaneamente la sicurezza e la privacy dell'archiviazione di oggetti in rete.

Ovviamente ci sono molti modi diversi per costruire e implementare un sistema simile, ma sicuramente è necessario rispettare caratteristiche ben definite e certe responsabilità per poter funzionare correttamente e garantire un buon servizio. StorJ introduce un framework modulare per rispettare questi vincoli e per costruire un'efficiente rete di archiviazione e memorizzazione distribuita. StorJ è una piattaforma di cloud storage non censurata che non può avere tempi di inattività. È una piattaforma di archiviazione cloud end-to-end criptata e decentralizzata che utilizza la crittografia per garantire la sicurezza dei file. Pertanto, questa piattaforma consente agli utenti di archiviare i dati in modo sicuro e decentralizzato. Infatti i file vengono crittografati e scompattati in blocchi, chiamati *pieces*, e archiviati in una rete decentralizzata di computer e server intorno il mondo, denominati *host*. Nessuno tranne l'utente può accedere ai propri file, anche in forma crittografata.

Per questo motivo StorJ è famoso per le seguenti caratteristiche, che lo distinguono dagli altri cloud storage:

- **Veloce**: più di una macchina può servire contemporaneamente i pezzi dei file richiesti.
- **Economico**: gli utenti possono affittare lo spazio sul disco rigido di altre persone invece di pagare per un data center appositamente costruito per archiviare i file.
- **Sicuro**: i file sono sia crittografati che scompattati, per garantire una maggiore sicurezza.

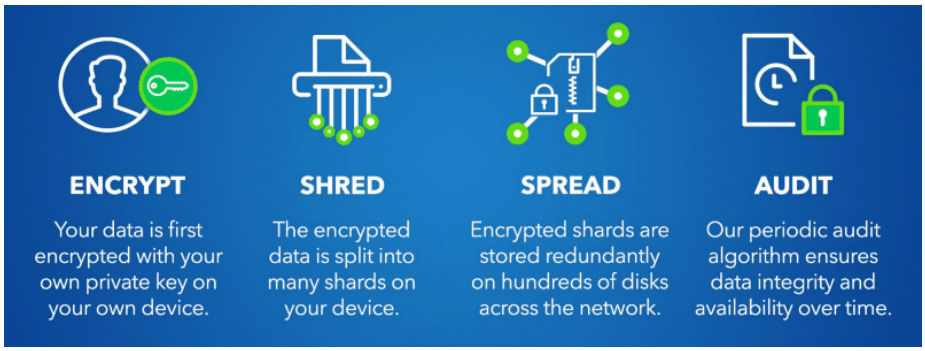


Figure 1: Caratteristiche principali di StorJ.

StorJ utilizza chiavi criptate e una funzione di hash crittografata per la sicurezza. I file vengono crittografati lato client sul dispositivo prima di essere caricati in rete, per proteggere al meglio i dati: ogni file è suddiviso in *chucks* che vengono prima crittografati e poi distribuiti attraverso la rete. La rete StorJ assume un ruolo fondamentale nella condivisione dei dati attraverso il cloud, perché ogni file è distribuito sui nodi StorJ gestiti da utenti di tutto il mondo, che affittano lo spazio inutilizzato del disco rigido in cambio di token, come criptovalute.

1 Obiettivi/Requisiti

L'obiettivo principale del progetto è essenzialmente analizzare l'intera architettura della tecnologia proposta da StorJ, spiegandone le principali caratteristiche, i principali componenti che ne fanno parte e come gli utenti possono interagire con essa.

L'obiettivo, inoltre, è quello di evidenziare sia i punti di forza che quelli di debolezza di tale tecnologia, sfruttando ambienti di simulazione reali, anche scomodi, su cui implementarne e testarne le funzionalità.

1.1 Scenari

Poiché questo caso di studio deve essere il più dettagliato e chiaro possibile, prima di tutto è necessario definire scenari ben delineati, al fine di suddividere il lavoro in diverse fasi per comprendere meglio il flusso di ricerca, analizzando ogni fase passo dopo passo.

In particolare si utilizzerà la seguente suddivisione:

- All'inizio verranno presi in considerazione tutti gli aspetti riguardanti la progettazione dell'architettura e dell'infrastruttura di sistema, quindi le sue principali caratteristiche e funzioni.
- Il passo successivo è analizzare e comprendere il framework utilizzato da questa tecnologia, ovvero fornire una prima visione di come i vari processi funzionano e interagiscono tra loro.
- Una volta compresa la teoria alla base di tale framework, verranno analizzate in modo più dettagliato le funzionalità che questa tecnologia consente, fornendo un'implementazione concreta.
- Successivamente verrà presentata una sezione pratica in cui verranno testate le principali funzionalità, per comprendere le potenzialità che StorJ mette a disposizione degli utenti.
- Infine, verranno presi in considerazione i possibili sviluppi futuri, e verranno evidenziate sia i punti di forza, che contraddistinguono StorJ dalle altre reti di cloud storage, sia le criticità, che tale tecnologia può incontrare.

2 Analisi Architetturale

Prima di progettare un sistema, è importante definirne vincoli e requisiti. Esistono molti modi diversi per progettare un sistema di storage decentralizzato, ma è fondamentale rispettare questi vincoli. Il primo compito di uno storage è garantire l'**universalità** del framework nonostante la presenza di vincoli molto ristretti.

I dati archiviati sulla rete Internet sono centralizzati e salvati in enormi data center, di proprietà di industrie tecnologiche, in grado di gestire un'enorme quantità di informazioni. La centralizzazione dell'archiviazione dei dati coinvolge alcuni argomenti importanti da gestire nel miglior modo possibile come **violazioni dei dati**, periodi di **indisponibilità**, **costi** per l'archiviazione fisica ed **estensibilità** dei dati per garantire un aggiornamento più rapido dei formati dei file rispetto alla richiesta degli utenti di dati più veloci.

Lo storage decentralizzato nasce come risposta per fornire una soluzione di cloud storage performante, sicura, privata ed economica.

2.1 Privacy e Sicurezza

L'obiettivo principale di una piattaforma di memorizzazione dati in rete è garantire sia la privacy che la sicurezza dei dati archiviati. In particolare, le piattaforme di storage decentralizzate devono gestire un ulteriore livello di complessità e rischio dovuto allo storage dei dati in nodi inaffidabili della rete, poiché gestiti da terze parti di cui non sempre si conosce il livello di affidabilità. Per questo motivo, lo storage decentralizzato deve essere costruito da zero, piuttosto che con un approccio basato su data center, per supportare sia la crittografia end-to-end che il vincolo di sicurezza della privacy dei dati a tutti i livelli del sistema.

In alcuni paesi i dati sono soggetti a standard di sicurezza ben definiti da rispettare, quindi è importante garantire trasparenza e resistenza agli attacchi.

2.2 Decentralizzazione

Per definizione, un'applicazione **decentralizzata** fornisce un servizio che non dipende esclusivamente da un singolo operatore. In questo modo, nessuna singola entità dovrebbe causare interruzioni dell'intero sistema o essere responsabile del costo delle operazioni gestite dal servizio.

Una delle principali motivazioni per preferire la decentralizzazione è la riduzione di costi delle infrastrutture per la **manutenzione**, **utenza** e **bandwidth**. La decentralizzazione nasce come modalità alternativa alle entità di archiviazione centralizzate che oggi dominano il mercato: è molto rischioso, infatti, consentire a singoli enti o società terze di conservare i dati personali, poiché potrebbe venderli a società pubblicitarie e rendere dati privati pubblici. Grazie a un'infrastruttura decentralizzata che offre sicurezza dei dati salvati, StorJ potrebbe anche smettere di funzionare che i dati sarebbero ancora disponibili per gli utenti che ne fanno richiesta.

La decentralizzazione risponde meglio alle esigenze di cloud storage e risolve molti limiti, rischi e fattori di costo fondamentali che provengono invece dalla centralizzazione di un sistema. In questo contesto, la decentralizzazione si traduce in una **rete distribuita** globale che può servire un'ampia gamma di casi d'uso di archiviazione; tuttavia, i sistemi di storage centralizzato richiedono architetture, implementazioni e infrastrutture diverse per affrontare ciascuno di questi stessi casi d'uso.

2.3 Durabilità e Device Failure

Per un sistema di archiviazione, uno degli obiettivi principali da perseguire e su cui prestare maggiore energia è quello di fare attenzione a non perdere i dati una volta archiviati, anche se si possono verificare possibili guasti del sistema. StorJ può memorizzare i dati con un'elevata **durabilità** e promette un **basso rischio** di perdita di dati.

È inevitabile che i componenti hardware di un dispositivo causino guasti al sistema come la rottura dei collegamenti di rete, interruzioni dell'alimentazione o inaffidabilità dei dati dopo un certo periodo di tempo. In questi termini è importante per un sistema decentralizzato garantire la **ridondanza** dei dati utilizzati come backup per il ripristino in caso di guasto del sistema.

I nodi di archiviazione della rete possono andare offline dopo un guasto hardware o software, connettività Internet intermittente, perdita di alimentazione, guasto completo del disco, arresto o rimozione del software. Per prevenire la perdita di dati, è necessario agire in base alla complessità del sistema e ai requisiti che deve fornire: più guasti di rete esistono, maggiore è la ridondanza necessaria per limitare il tasso di perdita dei dati; maggiore è la ridondanza, maggiore è la larghezza di banda necessaria per il corretto funzionamento del sistema. In effetti, esiste una relazione tra possibili guasti di rete e disponibilità di larghezza di banda.

2.4 Latenza e Larghezza di Banda

La **latenza** è il tempo di risposta di un dispositivo sulla rete a fronte di un input ricevuto. I sistemi di archiviazione decentralizzati possono sfruttare il *parallelismo* per:

- aumento dei tassi di trasferimento
- migliori capacità di elaborazione
- throughput complessivo anche quando i singoli collegamenti di rete sono lenti

Ma il parallelismo non può migliorare la latenza perché se un singolo collegamento di rete viene utilizzato come componente di un'operazione, la sua latenza sarà il limite inferiore per l'intera operazione. Quindi, se un sistema distribuito è costruito per applicazioni ad alte prestazioni, deve ottimizzare la sua intera architettura di sistema per una bassa latenza in modo continuo e aggressivo.

La **larghezza di banda** è la velocità di trasferimento dati, bit rate o throughput attraverso una rete. Con il guasto del dispositivo e possibili eventi imprevisti, qualsiasi sistema decentralizzato ha una quantità di traffico di riparazione per il ripristino: di conseguenza, è importante tenere conto della larghezza di banda richiesta non solo per l'archiviazione e il recupero dei dati, ma anche per la riparazione e la manutenzione dei dati. Problemi con la progettazione di un sistema di archiviazione che non considera attentamente la larghezza di banda:

- dare la priorità ai nodi di archiviazione con accesso illimitato alla larghezza di banda ad alta velocità
- centralizzazione del sistema a causa di un enorme bisogno di larghezza di banda

Quindi, per mantenere il sistema di archiviazione il più decentralizzato possibile e in grado di funzionare in ambienti diversi, l'utilizzo della larghezza di banda deve essere ridotto al minimo in modo aggressivo dato che la larghezza di banda **limita** lo spazio utilizzabile.

In effetti, la riparazione dei dati influisce sulla partecipazione alla rete dei nodi di archiviazione attraverso il loro utilizzo della larghezza di banda, limitando anche la quantità di spazio su disco utilizzabile. In altri termini, se un nodo di archiviazione ha 1 TB di spazio disponibile, con un limite di larghezza di banda mensile di 500 GB, e questo nodo di archiviazione può aspettarsi di riparare il 50% dei suoi dati in un mese, supponendo che ogni oggetto archiviato sia richiesto almeno una volta, non possiamo memorizzare più di 333 GB su questo nodo, perché qualsiasi cosa in più causa la saturazione della larghezza di banda, utilizzando più di quanto consentito (il 50% di 333 GB è 166,5 GB di capacità di riparazione dei dati $\rightarrow$, 333 GB + 166,5 GB = 499,5 GB, poco meno del limite di larghezza di banda di 500 GB).

$$PaidBandwidth + RepairBandwidth \leq BandwidthLimit$$

In sintesi, tassi di riparazione più elevati equivalgono a una dimensione di archiviazione effettiva inferiore, effetto che riguarda principalmente i nodi che servono dati a pagamento più frequentemente. In pratica, la larghezza di banda pagata varia con il tipo di dati che vengono archiviati su ciascun nodo. Per selezionare i giusti limiti di spazio utilizzabile, questi rapporti devono essere monitorati da vicino e costantemente.

2.5 Spazio di Archiviazione

Possiamo classificare i sistemi di archiviazione in due gruppi in base alla dimensione dei file che possono essere archiviati in essi:

- **Database** è la soluzione preferita per archiviare tante piccole informazioni
- **Object Store**, o un file system, è la scelta giusta per archiviare molti file di grandi dimensioni

La soluzione offerta da StorJ è un archivio in rete di oggetti decentralizzato per file di grandi dimensioni, poiché si presume che la dimensione degli oggetti archiviati sia di 4 MB o superiore. I file più piccoli sono ovviamente supportati, ma può essere più costoso memorizzarli siccome vengono archiviati come metadati e non distribuiti ai nodi di rete.

È importante notare che questa soluzione non svantaggerà negativamente i casi d'uso che richiedono la lettura di file di piccole dimensioni: gli utenti infatti possono aggirare questo problema con una strategia di impacchettamento aggregando e archiviando molti file piccoli come un unico file di grandi dimensioni.

Inoltre, il protocollo utilizzato per recuperare l'oggetto archiviato, consente agli utenti di scaricare file di piccole dimensioni senza richiedere il recupero completo dell'oggetto compattato.

2.6 Consenso Distribuito

La **Byzantine Fault-Tolerance** è una tecnica per costruire sistemi, generalmente distribuiti, che possono continuare a funzionare correttamente a fronte di guasti e attacchi ostili. Un sistema BFT ne garantisce il funzionamento, anche nel caso in cui una parte dei componenti dovesse deviare dal suo corretto funzionamento.

StorJ funziona in un ambiente **untrusted** in cui i singoli fornitori di memoria non sono necessariamente considerati affidabili. StorJ opera su Internet pubblico, consentendo a chiunque di diventare un fornitore di spazio online, e adottando il modello Byzantine, Altruistic, Rational (BAR) per identificare i partecipanti alla rete:

- I nodi *Byzantine* sono cattivi attori e possono deviare arbitrariamente dal loro protocollo di comportamento per qualsiasi motivo.
- I nodi *Altruistic* sono buoni attori e partecipano a un protocollo suggerito anche se la scelta è di deviare il comportamento.
- I nodi *Rational* sono attori neutrali e partecipano o deviano solo quando è nel loro interesse netto.

Alcuni sistemi di archiviazione di oggetti basati su data center distribuiti (cloud), come Amazon, operano in un ambiente in cui tutti i nodi sono considerati altruisti. Ad esempio, a parte i guasti hardware e le violazioni della sicurezza, i nodi di archiviazione di Amazon non faranno altro che ciò per cui sono programmati, perché Amazon li possiede e li esegue tutti.

Al contrario, i nodi StroJ operano in un ambiente in cui ognuno di essi è gestito dal proprio operatore **indipendente**, assumendo che la maggior parte dei nodi di archiviazione coinvolti nell'ambiente siano razionali, la minoranza bizantina e nessuno altruista. In questi termini, è importante assicurarsi che i nodi razionali sulla rete si comportino il più simile possibile al comportamento previsto dai nodi altruisti e gli effetti di quelli bizantini saranno minimizzati o eliminati.

La creazione di un sistema solido rispetto ai nodi che vanno in errore non richiede un protocollo di consenso bizantino (di tolleranza d'errore): StorJ evita l'utilizzo del **distributed byzantine consensus**, poiché si tratta di un sistema basato su *blockchain* dove alcuni nodi possono andare in fallimento.

2.7 Coordinazione tra Entità

I sistemi che evitano il coordinamento ove possibile hanno un rendimento migliore rispetto ai sistemi in cui i sotto componenti sono costretti a coordinarsi per garantire correttezza nel comportamento, dato che gli attori di sistema dovranno attenderne altri prima di eseguire i propri compiti e le operazioni di attesa possono avere un costo significativo.

I sistemi che riducono al **minimo** la coordinazione sono molto più **efficienti** nello scalare da piccoli a grandi carichi di lavoro. L'aggiunta di più risorse a un sistema che evita il coordinamento aumenterà la produttività e le prestazioni.

In questi termini, StorJ cerca di ridurre al minimo il coordinamento tra i nodi per raggiungere la scala degli exabyte, limitando i domini di coordinamento ai nodi che sono interamente controllabili da ciascun utente.

3 Framework

In questa sezione verrà analizzato il framework progettato, prestando particolare attenzione al fatto che StorJ cerca di garantire una forte flessibilità ed estensibilità dei componenti, in modo che siano indipendenti l'uno dall'altro in caso di aggiunta o rimozione di funzionalità future. Tutti questi componenti possono eseguire le seguenti operazioni sui dati:

- **Memorizzare**: quando un file viene archiviato in rete, un client lo cripta e lo scompone in più parti, che vengono distribuite ai peer attraverso la rete. Fatto ciò, viene generato un metadato che contiene informazioni su dove trovare i dati in rete, per riassemblare i pezzi e recuperare il file iniziale.

- **Recuperare**: quando i dati vengono recuperati dalla rete, il client segue i metadati per identificare le posizioni dei pezzi di file memorizzati, quindi, una volta recuperati i pezzi, il file originale verrà riassemblato sulla macchina del client che ne ha fatto richiesta.

- **Mantenere**: nel caso in cui la quantità di recupero dati sia inferiore ad una certa soglia, i dati necessari per i pezzi mancanti verranno rigenerati e sostituiti.

- **Retribuire**: viene concesso un compenso economico in cambio del servizio fornito.

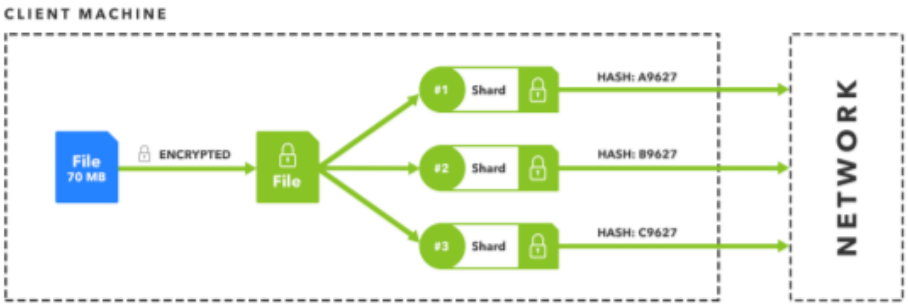


Figure 2: Processo di condivisione dei file tramite framework StorJ.

3.1 Stato dell'Arte

Per capire a fondo come è stata pensata e attuata l'infrastruttura proposta da StorJ è necessario comprendere le entità che ne fanno parte, capendone la terminologia e le loro funzionalità:

- **Client**: è un utente o un processo che caricherà o scaricherà dati dalla rete.
- **Peer Class**: è una raccolta di servizi di rete operanti nel sistema, come *storage nodes*, *satellites* e *uplinks*.

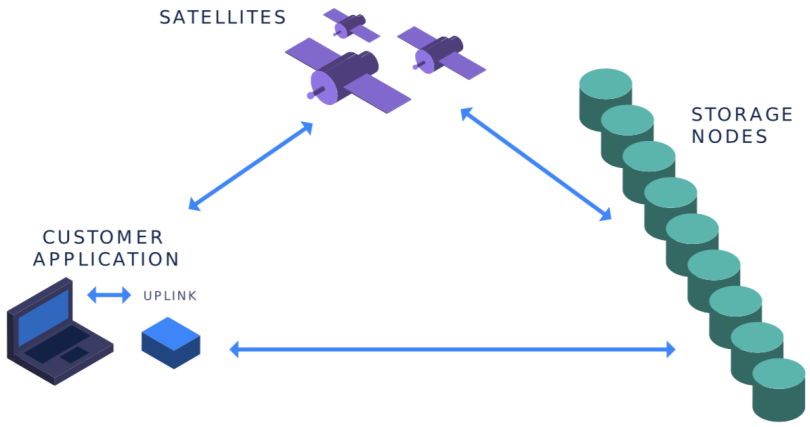


Figure 3: Classi dei peer che partecipano al framework StorJ.

- **Storage Node**: è un'entità che può scoprire altri nodi presenti nella rete, memorizzare dati, viene pagata per l'archiviazione e la larghezza di banda.

- **Uplink**: rappresenta qualsiasi applicazione o servizio che voglia archiviare e/o recuperare dati in rete, implementando crittografia ed erasure codes encoding.
- **Satellite**: è la classe del framework che partecipa al sistema di rilevamento dei nodi, memorizza nella cache le informazioni sull'indirizzo del nodo, memorizza i metadati degli oggetti, mantiene la reputation (affidabilità) del nodo di archiviazione, aggrega i dati di fatturazione, paga i nodi di archiviazione, esegue auditing e riparazioni e gestisce l'autorizzazione e gli account utente (che hanno fiducia nel framework StorJ).
- **Bucket**: è una raccolta illimitata, ma denominata, di file identificati da percorsi (ogni file ha un percorso univoco all'interno di un bucket).
- **Path**: è un identificatore (stringa di byte per controllo degli accessi) univoco per un file all'interno di un bucket. StorJ crittografa i percorsi prima che lascino il computer del client.
- **File/Oggetto**: sono i dati principali del sistema. Un file è indicato da un percorso, contiene una quantità arbitraria di byte e non ha una dimensione minima o massima. Un file è rappresentato da una raccolta ordinata di uno o più *segmenti*. I segmenti hanno una dimensione massima fissa. Un file supporta anche una quantità limitata di campi chiave/valore definiti dall'utente chiamati *attributi estesi*. Come i percorsi, i dati contenuti in un file vengono crittografati prima che lascino il computer del client per viaggiare in rete.
- **Attributo Esteso**: è un campo chiave/valore definito dall'utente che è associato a un file che, come i metadati di altri file, sono archiviati in modo crittografato.
- **Segmento**: rappresenta un singolo array di byte, compreso tra 0 e una dimensione massima configurabile dall'utente.
- **Segmento Remoto**: è un segmento che verrà codificato e distribuito attraverso la rete. Include informazioni come gli ID dei nodi su cui sono archiviati i dati.
- **Segmento Inline**: è un segmento sufficientemente piccolo in cui i dati occupano meno spazio rispetto ai dati corrispondenti di cui un segmento remoto avrà bisogno per tenere traccia di quali nodi avevano i dati. In questi casi, i dati vengono archiviati "in linea" anziché essere archiviati sui nodi.
- **Stripe**: è una suddivisione di un segmento. Una *striscia* è una quantità fissa di byte utilizzata come dimensione limite di crittografia e cancellazione (solo se il segmento è remoto) e su cui vengono eseguiti i controlli.

- **Erasure Share**: quando una stripe è codificata per la cancellazione, genera più parti chiamate *erasure share*. È necessario solo un sottoinsieme delle erasure share per recuperare la stripe originale. Ogni erasure share ha un indice che la identifica.
- **Piece**: quando le stripes di un segmento remoto vengono codificate in erasure share, le erasure share per quel segmento remoto con lo stesso indice vengono concatenate insieme, e quel gruppo concatenato di erasure share è chiamato *piece*. Se ci sono n erasure share dopo la cancellazione che codifica una striscia, allora ci sono n pezzi dopo l'elaborazione di un segmento remoto. L'i-esimo pezzo è la concatenazione di tutte le i-esime erasure share dalle strisce di quel segmento.
- **Puntatore**: è una struttura dati che contiene i dati del segmento in linea o tiene traccia di in quali nodi di archiviazione sono stati memorizzati i pezzi di un segmento remoto.

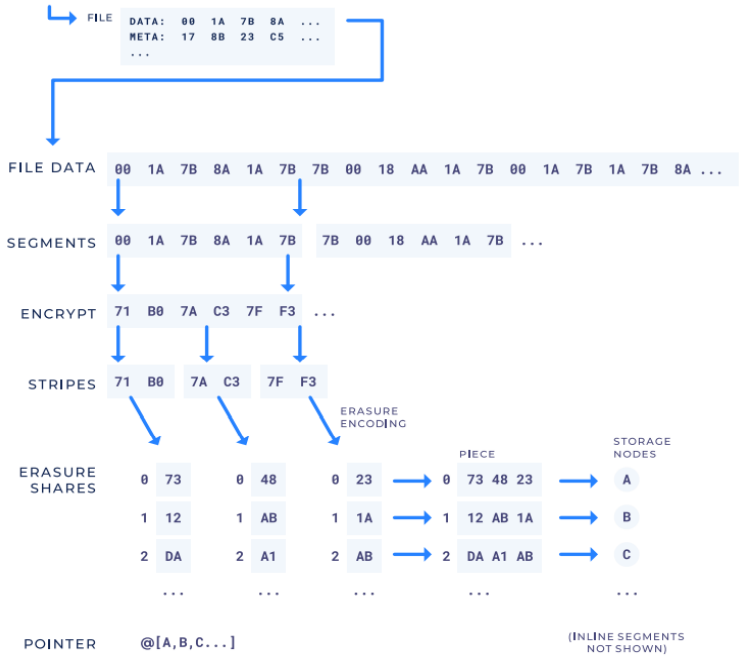


Figure 4: Decomposizione dei file.

3.2 Nodo di Archiviazione

Il ruolo del nodo di archiviazione è di **memorizzare** e **ritornare** dati. Oltre a garantire l'affidabilità dell'archiviazione dati, i nodi dovrebbero anche tutelare larghezza di banda e reattività della rete. Un operatore è un'entità che

vuole affittare il proprio spazio su disco inutilizzato in cambio di una qualche retribuzione; in sostanza mette a disposizione la memoria del proprio computer per memorizzare ed archiviare i file di altri utenti. I nodi di archiviazione vengono selezionati per archiviare i dati in base a vari criteri espliciti (come tempo di ping, latenza, velocità effettiva, capacità di larghezza di banda, spazio su disco sufficiente, posizione geografica) e, per ogni caricamento di file sulla rete, una quantità di metadati tiene traccia dei nodi selezionati per contenere i pezzi dell'oggetto memorizzato. Per semplificare la gestione di file non essenziali, viene associato ad ogni pezzo un TTL, ossia un "time-to-live", cioè un tempo massimo di permanenza del file all'interno della memoria, dopo di che verrà automaticamente eliminato. Se al momento della creazione del nodo non viene specificato nessun TTL, allora significa che potrà tenere memorizzati pezzi di file per un tempo non determinato.

A livello di sicurezza, ogni nodo gestirà il proprio **certificato** di autenticità, il quale richiede una coppia di chiavi, pubblica e privata, e un certificato auto firmato. La chiave privata del certificato di autenticità sarà memorizzata in un cold storage per evitare che venga compromessa ed è essenziale che venga gestita con un buon livello di sicurezza poiché in caso contrario sarà necessario richiedere un nuovo ID per il nodo. La chiave pubblica del certificato di autenticità determina l'ID per il nodo.

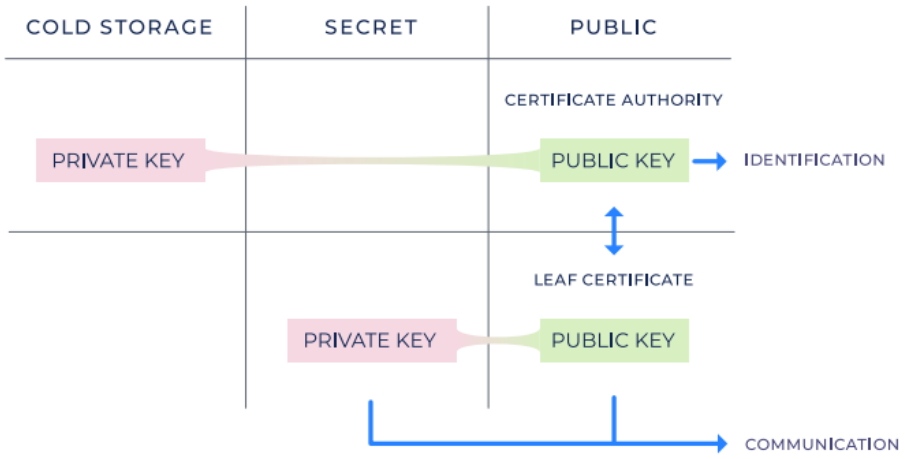


Figure 5: Certificati e chiavi del nodo di archiviazione.

Ogni nodo possiede un certificato *leaf* revocabile e una coppia di chiavi firmate dal certificato di autenticità del nodo. I nodi usano la coppia di chiavi per comunicare e ogni leaf possiede un **timestap** firmato di cui il satellite tiene traccia per ogni nodo. Nel caso in cui il certificato leaf venga compromesso, il nodo può emettere un nuovo certificato con un timestamp successivo. I peer interessati prenderanno nota del timestamp del certificato leaf appena creato e

rifiuteranno le connessioni da nodi con timestamp di certificati precedenti. Come meccanismo di ottimizzazione di gestione, i peer non dovranno prendere nota di quando sia certificato leaf che certificato di autenticità del nodo condividono lo stesso timestamp.

3.3 Comunicazione P2P

Tutti i peer presenti sulla rete comunicano tramite un **protocollo** standardizzato che fornisce:

- *reachability* dei peer in rete
- *authentication*, ovvero ogni partecipante alla comunicazione, dimostra l'identità del peer con cui sta parlando tramite crittografia, per evitare attacchi "man-in-the-middle".
- *privacy*, infatti la comunicazione tra il client e il nodo di archiviazione deve essere completamente privata.

Inoltre il framework è responsabile di mappare e dare un identificatore univoco ad ogni peer, in modo che ognuno di essi possa connettersi agli altri indirizzandosi a quello con cui vuole comunicare.

Esistono 3 diversi tipi di classi di peer presenti nella rete:

- Server di oggetti, che archiviano la maggior parte dei dati del sistema.
- Server di metadati, che tengono traccia dei metadati degli oggetti memorizzati in rete e in quali server sono localizzati.
- Clients, che accedono ai file memorizzati comunicando sia con il server degli oggetti sia con quello dei metadati per recuperare i dati corretti.

Per la comunicazione tra i peer presenti nella rete si utilizza un protocollo TLS (Transport Layer Security) che fornisce privacy e autenticità dei dati. Col tempo però, si è scelto di sostituire tale protocollo con uno più flessibile a livello di di sicurezza, il Noise Protocol Framework, che permette di ridurre gli scambi di handshake attraverso la rete nel caso in cui i dati siano già stati criptati e quindi è inutile inoltrarli in modo segreto. Quando si utilizza una comunicazione autenticata come TLS o Noise, ogni peer può accertare l'ID del nodo con cui sta parlando convalidando la catena di certificati e l'hashing della chiave pubblica del certificato di autenticità del suo peer. Si può stimare il carico di lavoro impiegato nella creazione del nodo considerando il numero di bit finali a 0 dell'ID. I satelliti possono impostare una soglia di lavoro minima per poter generare un *audit* che servirà poi nel tempo per ulteriori prove di correttezza.

3.3.1 Ricerca dei Nodi

Una volta compreso che esistono i nodi di rete e che possono comunicare tra di loro, bisogna considerare il fatto che i nodi sono un'astrazione dell'utente che

li mette a disposizione, e che quindi "vivono" grazie alla connessione internet e grazie ad un router che però cambia costantemente indirizzo IP all'interno della rete. Pertanto, l'obiettivo del sistema di individuazione dei nodi è fornire un mezzo per cercare l'ultimo indirizzo di un nodo in base all'ID del nodo, similmente a quanto accade con il DNS per la rete internet pubblica. La distributed hash table (DHT) **Kademlia** è un archivio chiave-valore con un protocollo integrato di ricerca del nodo. StorJ utilizza Kademlia come strumento primario per le azioni di lookup dei nodi nella rete, ignorando gli aspetti di memorizzazione chiave-valore. Utilizzando Kademlia soltanto per la scoperta dei nodi, si eliminano tutte quelle operazioni superflue che Kademlia mette a disposizione come ad esempio la ripubblicazione della chiave basata sul proprietario, la ripubblicazione della chiave basata sul vicino, e così via, eliminando inoltre la possibilità di eventuali attacchi al sistema. Sfortunatamente, le DHT come Kademlia richiedono più round trip di rete per molte operazioni, il che rende difficile ottenere tempi di risposta a livello di millisecondi. Per risolvere questo problema, è stato aggiunto un servizio di **cache** decentralizzato su Kademlia. Il servizio di caching vivrà in modo indipendente in ogni satellite e tenterà di comunicare con ogni storage node della rete su base continuativa, circa una volta all'ora. Il servizio di memorizzazione nella cache memorizzerà quindi l'ultimo indirizzo valido noto per ciascun nodo ed eliminerà i nodi con cui esso non ha parlato dopo un certo periodo di tempo. Anche se le operazioni di ping sono poco costose, ci si aspetta comunque che nel tempo venga trovata una soluzione implementativa migliore, nonostante appunto i requisiti di spazio siano trascurabili (la memorizzazione nella cache degli indirizzi per una rete di 80'000 nodi può essere realizzata con solo 5 MB di memoria).

A livello di failure, grazie ad una solida strategia di ridondanza, lo storage network sarà **resiliente** verso un certo grado di abbandono del nodo dalla rete o situazione di stallo di quest'ultimo. Pertanto, il sistema è robusto anche in caso di failure nelle operazioni di lookup nella cache, o se viene ritornato un indirizzo sbagliato. Inoltre, poichè il sistema di comunicazione peer-to-peer fornisce già un sistema di autenticazione, un'operazione di discovery del nodo in cache che fallisce o risponde in maniera errata causerà soltanto una perdita di performance ma non di correttezza.

Sebbene la cache del satellite non sia la fonte primaria di correttezza, poiché la strategia di riparazione dei nodi richiede di determinare se un nodo è online o offline in maniera rapida, il sistema smetterà di effettuare operazioni di lookup nella cache utilizzando Kademlia. Solo dopo casi di richieste di controllo non riuscite verrà eseguita una ricerca di riserva non simultanea in Kademlia per correggere le informazioni della cache potenzialmente obsolete. StorJ, oltre alle cache presenti all'interno dei satelliti, mette a disposizione alcune cache di rilevamento dei nodi gestite dalla comunità. Queste cache avranno il dovere di restituire rapidamente le informazioni sull'indirizzo per un determinato ID di un nodo e se il nodo è stato online recentemente. La cache per eseguire *node discovery* collezionerà inoltre informazioni sui nodi (metadati) attraverso lo scambio di messaggi tramite la rete, informazioni utilizzate poi per rendere più veloci e performanti le operazioni di lookup.

3.4 Ridondanza

Potrebbe succedere che a un certo momento, qualsiasi nodo di archiviazione possa andare offline in modo permanente. La strategia di **ridondanza** fornisce una tecnica che memorizza i dati in modo da garantirne l'accesso con alta probabilità, nonostante un numero qualsiasi di singoli nodi di archiviazione sia andato offline. Per ottenere un ottimo livello di durabilità (*probabilità che i dati rimangano disponibili anche in caso di guasto*), molti prodotti utilizzano semplici **repliche**. Sfortunatamente, la durabilità ha un impatto significativo sul fattore di espansione della rete (*overhead di archiviazione per l'archiviazione di dati affidabili*), aumentando il costo totale dei dati archiviati.

Ad esempio, supponiamo che un certo livello di durabilità richieda una strategia di replica che esegua cinque copie dei dati. Ciò causa un fattore di espansione del 500%. Questi dati devono essere archiviati sulla rete utilizzando la larghezza di banda, quindi una maggiore quantità di dati replicati si traduce in una maggiore quantità di larghezza di banda utilizzata per una certa quantità di dati. Come discusso in precedenza, l'utilizzo di una larghezza di banda elevata impedisce il ridimensionamento, quindi questa è una cattiva strategia per garantire un livello di durabilità elevato dei file.

Una tecnica alternativa alla semplice replica dei dati è l'uso di **codici di cancellazione** (erasure code), un modo più efficiente per garantire la ridondanza. Questa tecnica è ampiamente utilizzata sia per il sistema di archiviazione distribuito che P2P. I codici di cancellazione sono uno schema di codifica per manipolare la durabilità dei dati senza vincolarli all'utilizzo della larghezza di banda, con lo scopo di garantire la riparazione del traffico in un modo migliore rispetto alla semplice replica. È importante sottolineare che consentono variazioni di durabilità senza variazioni del fattore di dilatazione.

Per ogni oggetto vengono memorizzati anche 4 valori aggiuntivi k, m, o, n tali che $k <= m <= o <= n$. Di questi, k è il numero minimo di pezzi di file richiesti per la ricostruzione dell'oggetto e n è il numero totale di pezzi generati durante la creazione. m ed o sono rispettivamente il valore di sicurezza minimo e il valore ottimale, dove m è scelto in modo tale che se il satellite nota che il numero di pieces disponibili scende sotto m, viene subito triggerato il processo di riparazione nel tentativo di garantire sempre e comunque almeno k pieces, mentre o è scelto in modo tale che, durante i processi di upload e riparazione, appena o pieces hanno terminato l'upload, i restanti pezzi fino ad arrivare ad n vengono eliminati. Quindi possiamo dire che o è scelto in maniera tale che memorizzare un numero o di pieces è sufficiente per garantire la *durabilità* desiderata.

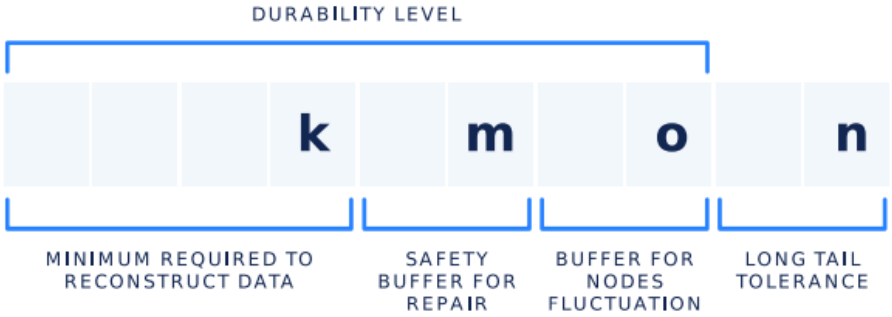


Figure 6: Suddivisione dell'oggetto memorizzato in cloud.

Secondo questo schema si può dire che:

- la quantità di uploads che possono essere tollerati senza perdere prestazioni risulta essere $n - o$
- la quantità di nodi che possono andare offline temporaneamente nello stesso momento senza avviare il processo di riparazione risulta essere $o - m$
- il buffer di appoggio utilizzato per evitare perdita di dati dal momento in cui si nota che i dati necessitano di un processo di riparazione e al momento in cui avviene effettivamente la riparazione, risulta essere $m - k$

3.4.1 Erasure Codes

Un codice di cancellazione (erasure code) è descritto come una coppia di due numeri, k e n. Quindi se un blocco di dati è codificato con un codice di cancellazione (k,n), significa che ci sono n *erasure share* totali generati, dove solo k di essi sono necessari per il recupero il blocco di dati originale. Quindi, se un blocco di dati è s byte, ciascuna degli n erasure shares è di circa s/k byte (in caso di $k = 1$, tutti gli erasure share sono unici).

Concentriamoci sul motivo per cui la durabilità degli erasure code è una risorsa importante dell'intero framework: ad esempio, la durata di un codice di cancellazione tipo ($k = 20$, $n = 40$), è migliore di un codice di cancellazione tipo ($k = 10$, $n = 20$), anche se il fattore di espansione (2X) è lo stesso per entrambi. Questo perché il rischio è distribuito su più nodi nel caso di ($k = 20$, $n = 40$).

Figure 7: Vari scenari del framework durante il caricamento e il download dei dati.

3.4.2 Riparazione Dati

La perdita di dati è un rischio sempre presente in qualsiasi sistema distribuito. Sebbene ci siano molte potenziali cause di perdita di file come danneggiamento fisico, comportamenti imprevisti, hardware difettoso, errore software, il motivo più importante per considerare la perdita di dati riguarda il **churn** (abbandono) del nodo di archiviazione, ovvero il nodo di rete va offline e lascia la rete. Il meccanismo di auditing viene utilizzato per verificare che i nodi memorizzino i dati correttamente, quindi ora l'obiettivo è rilevare quando un nodo di archiviazione smette di memorizzare i dati o va offline. Se un nodo va offline, è importante riparare i dati che aveva su nuovi nodi recuperando i dati originali tramite una **ricostruzione** degli erasure codes dai pezzi rimanenti, rigenerare i pezzi mancanti e memorizzarli di nuovo su nuovi nodi di archiviazione all'interno della rete.

Siccome un nodo potrebbe andare offline per un qualche motivo, insieme ai pieces dei file che memorizzava, sarà necessario ricostruire i pezzi mancanti dei file quando il numero di pieces di un segmento scende sotto una certa soglia m (quando un nodo va offline, il satellite provvederà a marcare i suoi pieces come mancanti). La cache del sistema di *node discovery* contiene informazioni accurate e aggiornate sullo stato dei nodi, in particolare se sono stati online recentemente. Quando un nodo cambia il suo stato da online a offline, questo evento provoca un processo di lookup inverso (indicizzato) nel database dei metadati dell'utente, in modo da identificare tutti i puntatori ai segmenti memorizzati in quel nodo. Per ogni segmento i cui pieces scendono sotto una certa soglia m, il segmento viene scaricato e ricostruito, i pieces vengono rigenerati

e memorizzati in nuovi nodi, e infine il pointer viene aggiornato con le nuove informazioni.

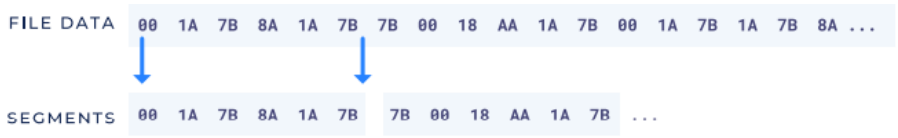
Una problematica importante da considerare è il fatto che il satellite deve stare costantemente in stato di running, poiché se il satellite dell'utente smettesse di funzionare, si stopperebbe anche il processo di riparazione dei dati, che quindi scomparirebbero dalla rete a seguito di un abbandono del nodo.

L'intera operazione di riparazione dati però rischia di essere molto onerosa per il sistema, perciò si è scelto di adottare degli *hash* per ogni piece, che verranno memorizzati all'interno dei pieces stessi in ogni storage node, mentre verrà memorizzato un hash di validazione (che convalida la correttezza degli hash dei pieces) nel puntatore. Durante il processo di riparazione quindi gli hash di ogni pezzo possono essere recuperati e convalidati per verificarne la correttezza rispetto al puntatore, consentendo così di convalidare ogni pezzo nella sua interezza. Ciò consente al sistema di data repair di valutare correttamente se la riparazione è stata completata o meno con successo, e senza utilizzare ridondanza extra (overhead).

3.5 Struttura Files

Come si è precedentemente discusso, all'interno della tecnologia StorJ i file vengono suddivisi in sotto porzioni per essere poi distribuiti ai nodi della rete, perciò, in seguito, verrà analizzata la procedura di decomposizione di un oggetto che entra nella rete StorJ.

3.5.1 File → Segments

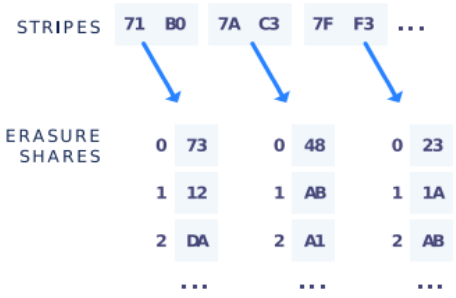


Il livello di suddivisione più alto di uno stream di dati di un file è chiamato **segment**. Nel caso in cui un file sia più piccolo dei suoi metadati (necessari per essere memorizzato in rete), i dati del file verranno memorizzati inline con i propri metadati, generando un *inline segment*. Nel caso invece di file di dimensione elevata, i dati verranno suddivisi in uno o più *remote segment*. Il fatto di dividere i grandi file in segmenti per essere distribuiti in rete comporta numerosi vantaggi in termini di sicurezza, privacy, prestazioni e disponibilità dei dati. Infatti, distribuendo i file all'interno dei nodi della rete si riduce l'impatto del *content delivery* dell'oggetto stesso, poiché anche la richiesta di bandwidth viene distribuita attraverso la rete. Inoltre in questo modo non ci sono standard per la dimensione dei file memorizzati in rete e quindi i tentativi di eventuali attaccanti per riuscire ad intercettare i dati non possono basarsi sulla lunghezza dei file, poiché i segmenti aiutano a tenere il traffico di rete nascosto.

3.5.2 Segments → Stripes

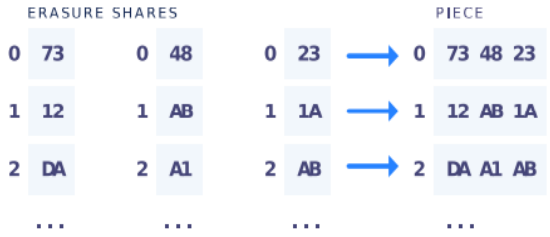
In alcune situazioni (ad esempio con determinati formati di file tipo video o audio) è importante avere accesso ad una sotto porzione di un pezzo del dato. A questo scopo, una **stripe** definisce un sottoinsieme di un segmento e non dovrebbe pesare più di 2 KB. La crittografia avviene su un piccolo nucleo di stripes, mentre la codifica della cancellazione (erasure encoding) avviene su una singola stripe alla volta.

3.5.3 Stripes → Erasure Shares



Come discusso precedentemente, gli **erasure codes** permettono di avere maggior controllo sulla durability della rete a fronte di storage node inaffidabili. Considerando un erasure code (k, n), vengono generati n erasure shares per ogni stripe. Ad esempio una stripe viene suddivisa in 40 erasure shares ($n = 40$), dove soltanto 20 ($k = 20$) sono utilizzati per la ricostruzione della stripe. Quindi ciascuno dei 40 erasure shares sarà $\frac{1}{20}$ della dimensione originale della stripe. L'erasure encoding di una singola striscia alla volta consente di leggere piccole porzioni di un grande segmento senza recuperare prima l'intero segmento.

3.5.4 Erasure Shares → Pieces



Poiché le stripes sono già molto piccole di dimensioni, gli erasure shares spesso sono ancora più piccoli, e i metadati associati per tenere traccia di ognuno di loro separatamente saranno di dimensioni molto maggiori rispetto alla loro. Tutti gli n erasure shares hanno un indice associato ben definito, in particolare per una stripes fissata e qualunque n, l'i-esimo share di un erasure code sarà sempre lo stesso. Quindi invece di tenere traccia di tutti gli erasure shares separatamente, tutti gli erasure shares con lo stesso indice vengono aggregati all'interno di un **piece**. In un erasure code (k, n) ci sono n pieces dove ogni pezzo i è la concatenazione ordinata di tutti gli erasure shares con indice i. Di conseguenza, dove ogni erasure shares è $\frac{1}{k}$ di una stripe, ogni piece è $\frac{1}{k}$ di un segment, e soltanto k pezzi sono necessari per il recupero dell'intero segmento. Un **piece** è ciò che viene memorizzato all'interno del nodo di rete.

I satelliti generano un *root piece ID*, scelto casualmente ogni volta che inizia un nuovo upload. L'uplink manterrà segreto l'ID e invierà un piece ID specifico a ciascun nodo di archiviazione, formato prendendo il codice di autenticazione del messaggio basato su hash (HMAC) dell'ID root (piece) e dell'ID del nodo. Questo serve a oscurare quali pieces appartengono insieme ai nodi di archiviazione. Il root piece ID viene memorizzato nel *puntatore*. I nodi di archiviazione denominano i pieces secondo l'ID del satellite. Se l'ID di un pezzo di file utilizzato da un satellite è riutilizzato da un altro satellite, ogni satellite può tranquillamente assumere che tale ID si riferisce ad un piece diverso di un altro satellite, con contenuto e ciclo di vita diversi.

3.5.5 Puntatori

Il proprietario dei dati vorrà conoscere in qualche modo come un segmento remoto viene suddiviso e dove sono collocati i pezzi per recuperarlo all'interno della rete. Tutto ciò è contenuto all'interno di una struttura dati chiamata **pointer**. Il puntatore quindi tiene traccia dei nodi nei quali sono memorizzati i pieces, delle informazioni di encryption, dei dettagli riguardo gli erasure codes, della soglia di riparazione che determina quanta ridondanza un segmento deve perdere prima di avviare il processo di riparazione, della quantità di pieces che devono essere memorizzati per considerare di successo il processo di riparazione (se il segmento è di tipo *inline*, il pointer conterrà i dati dell'intero segmento

piuttosto che i nodi nei quali sono stati suddivisi i suoi pieces).

3.6 Metadati

Una volta che un oggetto è diviso in pezzi più piccoli con *erasure codes*, è necessario tenere traccia di quali nodi di archiviazione sono selezionati per memorizzarli, in una visione di recupero dell'intero oggetto riassemblando i pezzi lungo il percorso. StorJ consente agli utenti di scegliere lo storage in base alla posizione geografica, alle prestazioni o allo spazio disponibile, utilizzando uno schema di selezione dei nodi esplicito (ricerche basate su directory). Inoltre, l'utente deve scegliere una **chiave** arbitraria per identificare la mappatura dei dati rispetto al nodo. Tutte queste caratteristiche e informazioni sono archiviate in un sistema di archiviazione basato su **metadati**.

Ogni volta che un oggetto viene aggiunto, modificato o rimosso, è necessario regolare o aggiornare una o più voci in questo sistema di archiviazione. Di conseguenza, il rischio di perdita dei metadati aumenta, e l'accumulo dei dati stessi potrebbe essere problematico in termini di dimensione della memoria, quindi è fondamentale considerare il fatto che i metadati variano con la dimensione dell'oggetto: maggiore è la dimensione media dell'oggetto all'interno dello storage , minore è il sovraccarico dei metadati da archiviare.

In StorJ, il sistema di metadati dell'infrastruttura memorizza principalmente i *pointers* (puntatori), in modo tale che gli altri componenti di rete comunicheranno con il database dei puntatori sia per memorizzare che recuperare i puntatori associati alle azioni da eseguire sui dati. Permettere agli utenti di gestire i metadati all'interno di un database tradizionale presenta i seguenti vantaggi:

- *Controllo Dati*: l'utente possiede il pieno controllo sui dati che lui stesso memorizza, eliminando possibili errori dovuti alla presenza di organizzazioni intermedie per la gestione dei dati memorizzati.

- *Semplicità*: mentre alcuni sistemi perdono grandi quantità di energia, tempo e risorse per implementare un'infrastruttura che sia Byzantine-fault tolerant, quindi con un certo grado di consenso nell'archiviazione dei metadati, in questo caso tutto questo lavoro viene risparmiato, traendone quindi benefici preziosi.

- *Risparmio Coordinazione*: gli utenti devono soltanto preoccuparsi di coordinarsi con gli altri utenti del proprio satellite, perciò se un utente deve eseguire un gran numero di richieste (causando saturazione di banda) può istanziare il proprio satellite senza preoccuparsi di coordinarsi con gli altri, riducendo così tutto l'overhead dovuto alle operazioni di coordinamento.

Lo scopo principale quindi è quello di permettere agli utenti di memorizzare i metadati in un database che possono scegliere e controllare loro stessi. D'altra parte però questo tipo di approccio può presentare i seguenti problemi:

- *Disponibilità*: la disponibilità dei dati dell'utente è strettamente legata alla disponibilità del server dei suoi metadati. La disponibilità del servizio

può essere resa buona da infrastrutture distribuite già esistenti e affidabili, assumendo che vengano impiegate risorse adeguate nel mantenimento delle operazioni per evitare ceh falliscano. Inoltre eventuali tempi di inattività del servizio di metadati non impattano sul resto della rete.

- *Durabilità*: nel caso il server dei metadati subisca una failure di notevoli dimensioni e non sia presente una policy di backup adeguata, tutti i dati dell'utente verranno persi. L'utilizzo di un database tradizionale aumenta questo tipo di rischio poiché fa uso di chiavi per la crittografia, ma StorJ memorizza comunque una copia di backup dei metadati periodicamente per evitare questo tipo di situazioni.

- *Fiducia*: l'utente deve fare totale affidamento sul server dei metadati.

3.6.1 Satellite

La raccolta dei servizi che memorizzano i metadati prende il nome di **Satellite**. Gli utenti della rete quindi avranno un loro account associato a una specifica istanza di un satellite, nel quale potranno memorizzare i metadati dei file, gestire l'access control sui dati, tenere traccia dell'affidabilità dei nodi di rete, riparare e mantenere i dati quando la ridondanza non è elevata. Bisogna specificare che un satellite non è necessariamente costituito da un singolo server, ma potrebbe anche essere eseguito come una collezione di server supportati da un database affidabile. Storj implementa un modello *thin client* che delega la fiducia nella gestione della posizione dei metadati dei file al servizio offerto dal satellite, che gestisce appunto la proprietà dei dati. Gli uplink sono così in grado di supportare la più ampia gamma possibile di applicazioni client, mentre i satelliti richiedono tempi di attività elevati e infrastruttura potenzialmente significativa.

Un satellite è un *server* applicativo standard che incorpora un database affidabile. Gli utenti accedono a un'istanza specifica di un satellite tramite le loro credenziali, e i dati disponibili in un'istanza di un certo satellite non saranno disponibili nell'istanza di un altro satellite, sebbene siano previsti vari livelli di importazione ed esportazione dati.

Nel rispetto della *sicurezza* dei dati dell'utente, il satellite non invia mai dati non criptati e non tiene memorizzate chiavi crittografiche. Le uniche caratteristiche di un oggetto che un satellite può condividere sono la sua esistenza, la dimensione, e altri metadati come le modalità di accesso. In questo modo viene completamente tutelata la privacy dei clienti e l'utente è in grado di avere pieno controllo dei dati che possiede, affidando la responsabilità di rendere i dati disponibili interamente al satellite.

Un'istanza di un Satellite è composta dai seguenti componenti:

- Una cache per effettuare node discovery

- Un database per i metadati indicizzato e crittografato

- Un sistema di gestione degli account e degli accessi ai dati

- Un sistema di auditing, statistiche e reputazione dei nodi di archiviazione
- Un sistema di riparazione dati
- Un servizio di pagamento per gli storage nodes

3.7 Crittografia

Il design di StorJ garantisce la totale sicurezza e privacy dell'intero sistema, quindi tutti i dati o metadati all'interno dello storage di rete vengono crittografati. I dati devono essere crittografati il prima possibile rispetto al flusso di archiviazione, idealmente prima che i dati lascino il computer di origine.

Il meccanismo di **crittografia** consente agli utenti di scegliere uno schema di crittografia specifico da adottare durante la memorizzazione dei dati. Si dovrebbero memorizzare anche i metadati di questo schema per consentire agli utenti di recuperare i propri dati utilizzando il meccanismo di decrittazione appropriato nel caso in cui le metodologie di crittografia vengano modificate o aggiornate. Per evitare la violazione dell'accesso di utenti indesiderati, la stessa chiave di crittografia non dovrebbe essere utilizzata per tutti i file, perché avere accesso a un singolo file significherebbe avere accesso a tutti i file utilizzando la stessa chiave di decrittazione. Quindi, ogni file dovrebbe essere crittografato con una **chiave univoca**, per consentire agli utenti di condividere l'accesso a determinati file senza fornire dettagli di crittografia per altri.

Poiché ogni file ha una chiave di crittografia diversa, i relativi metadati devono essere archiviati in modo sicuro e affidabile. Questi metadati sul file, incluso il suo percorso, verranno archiviati nel sistema di archiviazione dei metadati, a sua volta crittografato con uno schema di crittografia deterministico e gerarchico, al fine di consentire la condivisione soltanto di alcuni file senza condividerne altri.

I dati sono crittografati in blocchi di piccoli *batches* di strisce, di dimensione pari o inferiore a 4KB. Mentre la stessa chiave di crittografia viene utilizzata per ogni encryption batch in un segmento, i segmenti possono avere chiavi di crittografia diverse. Tuttavia, il *nonce* per ogni batch crittografato deve aumentare in modo monotono rispetto al batch precedente per tutto il segmento. Il nonce torna a 0 se il contatore raggiunge il valore massimo rappresentabile. Per prevenire attacchi di riordino dei dati, il nonce iniziale di ogni segmento è scelto in modo deterministico in base al numero del segmento. Quando più segmenti sono caricati in parallelo il il nonce iniziale per ogni segmento può essere calcolato dal nonce iniziale del file e il numero del segmento. Questo schema protegge il contenuto dei dati dal nodo di archiviazione che li ospita. Il proprietario dei dati mantiene il controllo completo sulla chiave di crittografia, e quindi sull'accesso ai dati. Nel caso in cui l'utente disattivi la crittografia sul percorso dati, questi saranno resi visibili soltanto al satellite scelto dall'utente, ma non ai nodi di archiviazione, che continueranno a non avere alcune informazione riguardo il percorso dei dati e i metadati dei pieces che memorizzano.

3.7.1 Autorizzazione

La crittografia protegge la privacy dei dati consentendo l'identificazione di una eventuale manomissione, ma l'**autorizzazione** consente di prevenire la manomissione impedendo ai client di apportare modifiche non autorizzate. Gli utenti autorizzati potranno aggiungere, rimuovere e modificare i file, mentre gli utenti non autorizzati non avranno queste capacità. Le operazioni sui metadati saranno autorizzate. Gli utenti si autenticheranno con il loro satellite, il che consentirà loro di accedere a varie operazioni in base alla loro configurazione di autorizzazione. Il sistema di autorizzazione utilizza i *macaroons*, ossia una specie di token che autorizza il possessore dei metadati ad avere accesso ad alcune risorse specifiche. I macaroons fanno in modo che chiunque possa aggiungere restrizioni che non potranno essere rimosse, senza doversi coordinare con un partito centrale. StorJ utilizza i macaroons per limitare e definire quali operazioni possono essere eseguite e a quali percorsi criptati possono essere associate. Quindi siccome è necessario limitare le operazioni sul satellite, e il satellite ha accesso soltanto ai percorsi dei file criptati, il meccanismo di autorizzazione deve lavorare sui percorsi dei file criptati, mentre l'accesso a specifici percorsi deve rimanere una limitazione fornita dalla crittografia.

Una volta che l'applicazione uplink è stata autorizzata, il satellite firmerà e approverà le operazioni ai nodi di archiviazione. L'uplink quindi dovrà recuperare una firma dal satellite prima di eseguire operazioni sui nodi. Tutte le operazioni sui nodi di archiviazione richiedono uno specifico ID del satellite e la firma associata. Un nodo quindi non permetterà le operazioni non firmate dal proprio satellite o con la giusta firma ma su oggetti appartenenti a un altro satellite.

3.8 Reputazione

Le *metriche di reputazione* nei sistemi decentralizzati sono una parte critica quanto fondamentale della cooperazione tra i nodi di rete. Sono utilizzate infatti per garantire che cattivi agenti all'interno della rete vengano eliminati, in modo da migliorare la sicurezza globale del sistema, la sua affidabilità e la durabilità. La *reputation* di un nodo di archiviazione può essere suddivisa in 4 sottosistemi, dove il primo serve come prova dell'identità di nodi appena entrati nel sistema per capire se sono adatti a contenere dati, il secondo è il processo di verifica iniziale per conoscere l'affidabilità di un nodo appena entrato nella rete, il terzo è un sistema di filtraggio che permette di bloccare la partecipazione in rete di nodi malevoli e infine il quarto è un sistema di preferenze basato sugli audits dei nodi che fanno ancora parte della rete dopo il processo di *filtering*.

In StorJ la reputazione del nodo di archiviazione preferenziale viene utilizzata solo per selezionare dove i nuovi dati verranno memorizzati, sia durante la riparazione che durante il caricamento di nuovi file, a discapito di eventi squalificanti. Se la reputazione preferenziale di un nodo di archiviazione diminuisce, i suoi pezzi di file non verranno spostati o riparati su altri nodi.

Quando un nuovo satellite si unisce alla rete, i nodi di archiviazione parte-

cipanti inizieranno il proprio processo di verifica. Questo processo limita la loro esposizione al nuovo e sconosciuto satellite, mentre costruisce una sua affidabilità nel tempo per evidenziare quale dei Satelliti ha la miglior capacità di pagamento. I nodi di archiviazione saranno in grado di configurare la quantità massima di dati che memorizzeranno per un satellite non attendibile e creeranno dati storici sul satellite che sarà ulteriormente affidabile in futuro.

3.8.1 Auditing

È essenziale convalidare e verificare che i nodi di archiviazione memorizzino ciò che è stato loro chiesto di archiviare nel modo più accurato possibile. Per questo motivo molti storage fanno affidamento su *prove di recuperabilità* probabilistiche chiamate **audits** per controllare e confermare con un grado di certezza e basso costo di overhead, che un nodo di archiviazione si stia comportando bene, mantenendo memorizzati i dati che dichiara, e che non sia vulnerabile a fallimenti. StorJ utilizza gli audit per determinare il grado di stabilità di un nodo: ad esempio, gli audit falliti comporteranno che un nodo di archiviazione venga contrassegnato come non valido, quindi i suoi dati verranno ridistribuiti a nuovi nodi evitando che quel nodo venga utilizzato per archiviare dati in futuro. Tuttavia, questo meccanismo di controllo non controlla tutti i byte in tutti i file. È in grado di rilevare falsi positivi, in cui la verifica di correttezza ritiene che il nodo di archiviazione stia memorizzando i dati correttamente ma in realtà i dati sono stati modificati o eliminati. Quando applicato a un nodo di archiviazione nel suo insieme, il rilevamento di dati mancanti o alterati diventa certo entro una soglia di errore nota e modificabile, quindi è necessario un sistema di **reputation** per mantenere la cronologia dei risultati degli audit per determinate l'identità di un nodo.

Perciò in una rete con nodi non attendibili, convalidare che tali nodi restituiscano dati in modo accurato o che si comportino diversamente da come previsto è fondamentale per garantire un corretto funzionamento di sistema. Gli audit sono un modo per confermare che i nodi dispongano dei dati che affermano di avere. Senza una periodica rigenerazione di queste challenge, un nodo di archiviazione potrebbe iniziare a superare gran parte degli audits senza memorizzare i dati richiesti ma tenendo traccia delle challenge precedenti e salvandone le risposte attese. Reed-Solomon erasure encoding è in grado di individuare errori ed eventualmente correggerli, tramite un meccanismo di correzione degli errori come l'algoritmo di Berlekamp-Welch, leggendo quindi una stripe alla volta come fosse una challenge e successivamente validando le risposte ottenute dagli erasure share. In questo modo si possono eseguire gli audits arbitrariamente senza challenge generate precedentemente.

Per eseguire un audit bisogna scegliere una stripe, della quale si richiedono gli erasure share da tutti i nodi di archiviazioni responsabili di quella stripe. Si lancia l'algoritmo di Berlekamp-Welch sugli erasure share e quando un buon numero di nodi ritorna l'informazione corretta, a quel punto diventa facile individuare errori ed anomalie. Tramite audit quindi si riesce a capire se un nodo di archiviazione è offline o non si comporta come dovrebbe restituendo

informazioni sbagliate: nel caso un nodo vada offline, il fallimento dell'audit potrebbe essere dovuto al fatto che la cache di discovery dell'indirizzo del nodo risulti obsoleta, e quindi verrà eseguita una nuova ricerca con Kademlia. Se il nodo risulta ancora offline, il satellite mette il nodo in modalità *containment*, in modo che venga eseguito l'audit finché non si ottiene una risposta corretta, altrimenti classificherà l'audit come fallito se non ha successo dopo una certa quantità di tempo in cui il nodo rimane offline. Nel caso si ottenga una risposta corretta, il nodo esce dalla modalità containment.

Tutti i fallimenti degli audit vengono memorizzati e salvati in un sistema di reputation del nodo, in modo che vengano utilizzati anche per testare la latenza del nodo, il throughput e la responsiveness.

3.9 Allocazione di Banda

E' fondamentale per il corretto funzionamento del sistema conoscere quanta larghezza di banda è utilizzata dai peer. Nella versione precedente del framework, veniva utilizzato un *exchange report* per collezionare le informazioni su cosa accade durante l'interazione tra due peer. Una volta terminata una normale operazione, entrambi i peer coinvolti nella comunicazione inviano un report a un servizio centrale che li colleziona per trovare un accordo. Se i peer sono in accordo si può calcolare la bandwidth utilizzata, mentre se non lo sono vengono utilizzati metodi di analisi dati e regressione per determinare quale peer ha influito negativamente sulla comunicazione. Con la nuova versione si vuole impedire un'influenza negativa a livello di protocollo di comunicazione. Per fare ciò si utilizza il protocollo *proxy-based authorization and accountabiliting for distributed systems* di Neuman che permette di misurare in maniera più accurata le risorse utilizzate e in un modo decentralizzato e utilizzando delegazione. Questo protocollo prevede che se un amministratore ha abbastanza risorse per coprire una certa operazione, un server allora creerà un *check* digitale firmato e lo trasferirà direttamente al proprietario dell'account. Si fa riferimento a questo check con il nome di *proxy*, ma può anche essere visto come *bandwidth allocation*. Questo controllo contiene informazioni che identificano il server, i due peer comunicanti, il numero massimo di risorse impiegabili in una operazione, un codice di controllo per prevenire duplicati e la data.

Nel contesto relativo alla tecnologia in questione il server è il satellite, mentre i peer sono rappresentati dai nodi di archiviazione, di cui si vogliono conoscere le risorse, ossia la bandwidth per la comunicazione. Il satellite allocherà banda soltanto se l'applicazione (uplink) è autorizzata alla richiesta. All'inizio di un'operazione di archiviazione, l'uplink trasferisce l'allocazione di banda allo storage node, che, una validata la firma del satellite da cui proviene, potrà effettuare un'operazione di richiesta dei limiti di larghezza di banda consentita. Invece di permettere nodi di archiviazione di imbrogliare e salvarsi l'allocazione di banda senza avere effettuato l'operazione di richiesta, ogni operazione viene suddivisa in sotto richieste in modo che anche se un nodo di archiviazione, o l'uplink stesso, smettessero di adottare questo protocollo, nessuna delle due peer class viene esposta a una perdita dati sostanziosa (protocollo *fair-exchange*).

Per supportare il protocollo di amministrazione di Neuman e l'overhead del satellite, l'allocazione di banda viene limitata, limitando quindi anche le capacità dei proxy. I proxy possono utilizzare chiavi pubbliche/private per la crittografia, il che significa che chiunque può validare il proxy, invece di un solo emittente originale. Siccome l'uplink possiede già una coppia di chiavi (come parte della sua identità), viene utilizzata la coppia di chiavi esistente invece che ricrearne una nuova per ogni richiesta. Le richieste di allocazione di larghezza di banda limitate, sono limitate dall'uplink per limitare il valore dell'allocazione della larghezza di banda solo a ciò che è stato trasferito finora. In questo modo, lo storage node manterrà solo l'allocazione di larghezza di banda più grande che ha ricevuto fino a quel momento, e l'uplink invierà solo allocazioni di larghezza di banda leggermente più grandi di quelle che ha ricevuto. Il nodo di archiviazione non ha alcun incentivo a mantenere più dell'allocazione più grande, in quanto condividono tutti lo stesso *check code*, che può essere incassato solo una volta.

3.9.1 Restrizioni di Allocazione

Nel caso di un'operazione GET, assumendo che l'allocazione di banda firmata dal satellite sia di x bytes totali, l'uplink inizierà a inviare una richiesta di allocazione ristretta per un piccolo numero numero di y bytes. Il nodo può quindi verificare l'autorizzazione dell'uplink, e se la richiesta di allocazione è firmata correttamente dal satellite, il nodo trasferirà dati di dimensione pari a y bytes prima di mettersi in attesa di una nuova richiesta. L'uplink invierà un'altra richiesta di allocazione dove y è maggiore, continuando a inviare richieste di allocazione dati finché y non diventa grande come x. Per ogni transizione, il nodo invia soltanto i dati che non sono stati inviati in precedenza, cioè invia soltanto un totale di x bytes.

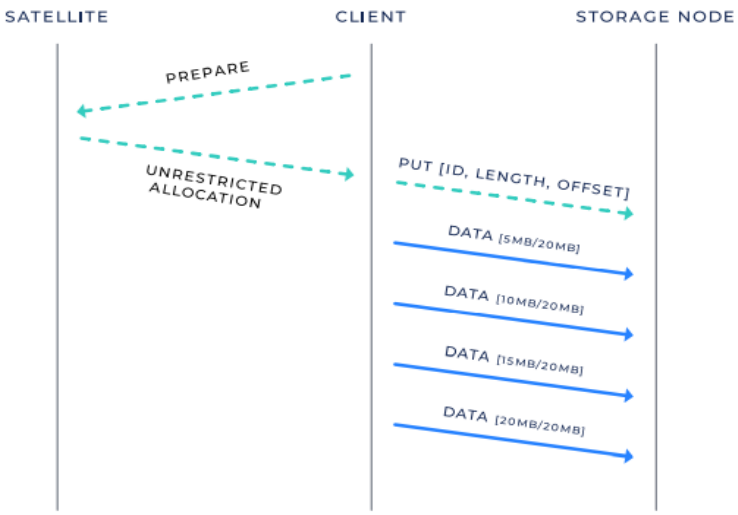


Figure 8: Operazione di PUT.

Se la richiesta viene terminata in un qualsiasi momento, si in modalità pianificata che non, il nodo conserverà il valore di allocazione ristretta più alto ricevuto fino a quel momento. Questo valore è la conferma firmata dall'uplink che l'uplink stesso è concorde sull'utilizzo della larghezza di banda fino a y bytes, insieme alla conferma del satellite sull'utilizzo della larghezza di banda di capacità x concessa all'uplink. Il nodo di archiviazione invierà periodicamente il più grande valore per l'allocazione di larghezza di banda limitata che ha ricevuto dai Satelliti, che lo pagheranno per la banda messa a disposizione. Se l'uplink non può regolare l'utilizzo della larghezza di banda, il satellite non firmerà una richiesta di allocazione di larghezza di banda, tutelando la reputazione del satellite. Allo stesso modo, se l'uplink tenta di utilizzare più larghezza di banda di quella allocata, il nodo di archiviazione rifiuterà la richiesta. Il nodo di archiviazione può essere pagato solo per l'importo massimo concordato dal cliente, poiché altrimenti non ha allocazioni di larghezza di banda valide da restituire per il pagamento. Questo sistema di misurazione del traffico della disponibilità di banda tiene traccia solo della larghezza di banda utilizzata durante le operazioni di archiviazione (archiviazione e recupero di pezzi).

Figure 9: Operazione di GET.

3.9.2 Garbage Collection

Quando i client spostano, sostituiscono o eliminano dati, il satellite notificherà ai nodi di archiviazione che non sono più tenuti a memorizzare quei dati. Nelle configurazioni dove i messaggi di cancellazione sono emessi dal client, il sistema di metadati richiederà la prova che le cancellazioni sono state emesse a un numero minimo configurabile di nodi di archiviazione. Ciò significa che ogni volta che i dati sono eliminati, i nodi di archiviazione online e raggiungibili riceveranno subito le notifiche. I nodi di archiviazione a volte saranno temporaneamente non disponibili e quindi perderanno i messaggi di eliminazione. In questi casi, i dati non necessari sono considerati spazzatura (*garbage*). I satelliti pagano solo per i dati che si aspettano di essere archiviati. I nodi di archiviazione con molta spazzatura guadagneranno meno di quanto guadagnerebbero altrimenti, a meno che non venga impiegato un sistema di raccolta dei "rifiuti". Per questo motivo, si introduce un sistema di *garbage collection* per liberare spazio sui nodi di archiviazione, cioè liberare spazio occupato da risorse non utilizzate.

I nodi di archiviazione richiedono periodicamente una struttura dati apposita per identificare differenze con i dati spazzatura che aveva memorizzato precedentemente, che potrebbe essere un hash code delle chiavi (permette una identificazione efficiente dello stato out-of-sync di un nodo). Una volta individuato lo stato di out-of-sync del nodo, si utilizza un'altra struttura dati aggiuntiva, come un filtro Bloom, per trovare quali dati spazzatura non sono stati cancellati. Restituendo una struttura dati su misura per ogni nodo sulla base di una pianificazione periodica, un satellite può dare ad un nodo di archiviazione la capacità di ripulire i dati spazzatura secondo una tolleranza configurabile. I Satelliti rifiuteranno frequenti richieste di queste strutture dati per evitare che

diventino eccessive.

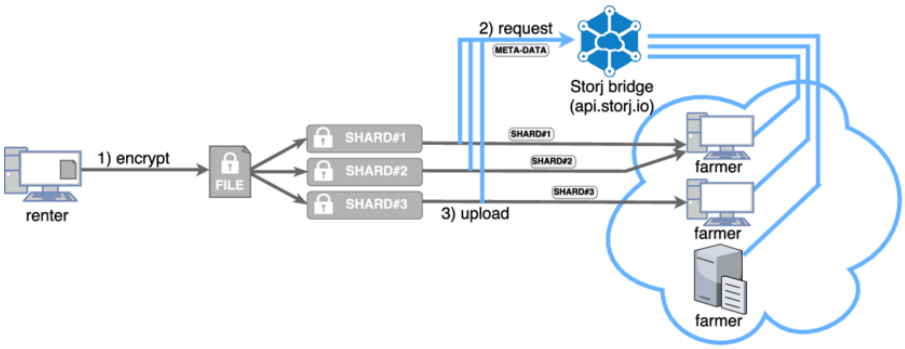


Figure 10: Workflow del sistema StorJ.

4 Sviluppo e Testing

Per quanto riguarda la fase di sviluppo e testing della tecnologia in questione, è stato utilizzato un terminale dove è stata eseguita un'istanza dell'applicazione di uplink, utilizzata appunto per facilitare l'interazione utente-sistema attraverso l'esecuzione di operazioni da riga di comando. In questa sezione verranno analizzate in prima battuta i vari step tramite i quali è stato possibile interagire con tale tecnologia, anche attraverso il supporto grafico di finestre di terminale, mentre in secondo luogo verrà effettuata una fase di testing seguita da un'analisi delle operazioni effettuate.

4.1 Preparazione dell'Ambiente

Per prima cosa è stato creato un progetto Storj tramite interfaccia web, inserendo:

- Nome, ossia un titolo del progetto, del quale poi potrà essere visualizzata una *dashboard* per osservarne l'andamento di memoria e larghezza di banda, monitorare il comportamento degli oggetti e della rete stessa, controllare i buckets e gli utenti connessi, etc.
- Satellite, ossia la piattaforma principale su cui si basa l'intero framework, selezionato tra 3 diverse opzioni, cioè Asia, Europa o America.

Una volta creato il progetto sul quale si intende lavorare, è stato necessario creare un **Access Grant**, ossia specificare con quali permessi un utente può interagire con il satellite del progetto. Questa operazione è molto delicata poiché StorJ, come già enunciato, fa della crittografia, della privacy e della sicurezza dei dati un punto di forza estremamente importante per il corretto utilizzo da

parte degli utenti. Quindi, dopo aver settato i permessi che si intende garantire tramite web form, il sistema ritorna 2 valori essenziali per il corretto settaggio dell'ambiente, ossia:

- **Satellite Address**: l'identificato del satellite sul quale si sta lavorando.
- **API Key**: il token con il quale StorJ garantisce sicurezza e privacy dei dati, anche nell'ambito di controllo degli accessi su un determinato oggetto.

A questo punto, si dispone di tutti i dati per effettuare il setup del sistema di testing.

4.2 Setup dell'Ambiente

Per utilizzare l'applicazione Uplink CLI, è stato necessario scaricare l'eseguibile on line per poterlo eseguire su un sistema operativo, quale Windows.

Una volta entrati nella cartella di download dell'eseguibile, per iniziare la configurazione dell'ambiente bisogna lanciare il seguente comando:

```
1  $ uplink.exe setup
```

A questo punto verranno richiesti all'utente dei settaggi da inserire durante la configurazione quali:

- `satellite`: selezionando quindi il satellite europeo *eu1*
- `access name`: inserendo un nome opzionale per poter effettuare l'accesso al satellite
- `API key`: inserendo il token prodotto durante la creazione dell'Access Grant, utilizzato poi da StorJ per la sicurezza delle operazioni sui dati
- `passphrase`: inserendo una parola segreta da richiedere per effettuare le operazioni e garantire quindi autenticità di chi le esegue

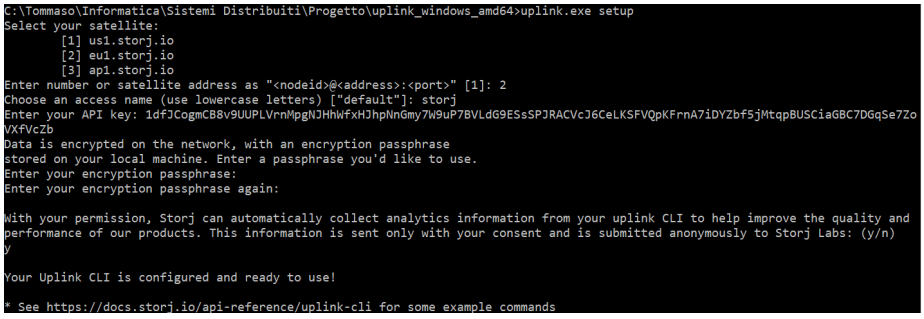


Figure 11: Setup dell'ambiente mediante l'applicazione Uplink CLI.

Una volta settati i campi per la configurazione dell'ambiente desiderata, lanciando il comando:

```
$ uplink.exe share --readonly=false --access=[name]
```

è stato generato e settato l'Access Grant, ossia i permessi, per quel determinato progetto che gira sul satellite definito in precedenza.

4.2.1 Controllo Accesso

E' possibile visualizzare in output le informazioni di accesso al satellite configurate e settate precedentemente tramite un'apposita suite di chiamate integrate al comando:

```
$ uplink.exe access [command]
```

dove [command] può essere sostituito da:

- `list`: stampa a video il nome e i satelliti associati agli accessi disponibili nella configurazione di sistema

```
C:\Tommaso\Informatica\Sistemi Distribuiti\Progetto\uplink_windows_amd64>uplink.exe access list
=========== ACCESSES LIST: name / satellite ================================
storj / 12L9ZFwhzVpuEKMUNUqkaTLGzwY9G24tbiigLiXpmZWKwmcNDDs@eu1.storj.io:7777
```

Figure 12: Output del comando `access list`.

- `inspect`: passando in input il nome dell'Access Grant configurato precedentemente, permette di visualizzare alcune informazioni utili

```
C:\Tommaso\Informatica\Sistemi Distribuiti\Progetto\uplink_windows_amd64>uplink.exe access inspect storj
{
  "satellite_addr": "12L9ZFwhzVpuEKMUNUqkaTLGzwY9G24tbiigLiXpmZWKwmcNDDs@eu1.storj.io:7777",
  "encryption_access": {
    "default_key": "gHwNnLb2F3UIMP7EgSOex4vfnL012Wkcgn5jTbd+XIk=",
    "default_path_cipher": "ENC_AESGCM"
  },
  "api_key": "1dfJCogmCB8v9UUPLVrnMpgNJHhWfxHJhpNnGmy7W9uP7BVLdG9ESsSPJRACVcJ6CeLKSFVQpKFrnA7iDYZbf5jMtqpBUSCiaGBC7DGqSe7ZoVXfVcZb",
  "macaroon": {
    "head": "YwlxpY72F5MBAHL5kJsmfDsDGSqqNvUFJpR0yjLo3EQ=",
    "caveats": [
      {
        "nonce": "PdLwyg=="
      }
    ],
    "tail": "siQEYUtxxxxi5IeA9YdYoUdo22S6Y17hREBY_oa_BMM="
  }
}
```

Figure 13: Output del comando `access inspect [name]`.

- `register`: utilizzato per registrare l'Access Grant in modo da poterlo utilizzare anche su un gateway hostato in rete, pubblico o privato

E' anche possibile rimuovere i permessi di accesso a determinati file mediante l'apposito comando `revoke` o importare una configurazione esistente tramite il comando `import`.

4.3 Interazione

In questa sezione si vuole riportare una un'implementazione delle operazioni mostrando e documentando come interagire con tale tecnologia. Come file campione per le dimostrazioni pratiche del funzionamento di StorJ, è stato preso in considerazione il logo ufficiale di StorJ sotto riportato.

Figure 14: Logo ufficiale StorJ.

4.3.1 Creazione Bucket

Per prima cosa occorre creare un **Bucket**, ossia il contenitore che permetterà di memorizzare i file che verranno archiviati al suo interno attraverso la rete. Per procedere con la generazione di un bucket quindi sarà sufficiente eseguire da terminale il seguente comando, specificando il nome del bucket che si intende istanziare:

```
$ uplink.exe mb sj://container
```

A questo punto, come si potrà osservare dall'output prodotto dal sistema, il bucket è stato creato ed è pronto a memorizzare i file che verranno aggiunti al suo interno. Inoltre è possibile controllare i file presenti all'interno di un bucket o la lista dei bucket presenti nel satellite attraverso il comando di listing:

```
$ uplink.exe ls
```

Ciò che avviene nella pratica è una richiesta della lista degli oggetti presenti nel bucket inviata dall'uplink. Quindi la richiesta, o meglio il percorso del bucket richiesto, viene criptata e inviata al satellite. A questo punto il satellite effettuerà una validazione in modo da verificare che la richiesta sia autentica e in caso di successo, risponderà con la lista degli oggetti presenti nel bucket. Alla fine l'uplink decripterà la risposta ricevuta e mostrerà il risultato all'utente richiedente.

4.3.2 Upload

Per eseguire l'upload di un file all'interno del bucket appena creato è sufficiente eseguire il comando:

```
$ uplink.exe cp [file] sj://[bucket]
```

specificando il path del file da caricare in rete e il nome del bucket all'interno del quale si vuole memorizzare.

E' possibile inoltre visualizzare la **distribuzione** geografica dell'oggetto tra i nodi della rete, poiché, come spiegato nella sezione riguardante l'architettura e il funzionamento del framework, StorJ divide ogni file in pezzetti di dimensione più piccola e li memorizza in nodi separati, per poi recuperarli una volta che l'utente fa richesta del file originale.

```
$ uplink.exe share --url sj://[bucket]/[file]
```

Con questo comando vengono visualizzate a video tutte le informazioni riguardanti il file specificato, e in particolare, viene rilasciato un URL che, se aperto tramite browser, mostrando la distribuzione geografica dei *pieces* del file tra i nodi che compongono la rete.

```
C:\Tommaso\Informatica\Sistemi Distribuiti\Progetto\uplink_windows_amd64>uplink.exe share --url sj://container/storj.png --not-after=none
Sharing access to satellite 12L9ZFwhzVpuEKMUNUqkaTLGzwY9G24tbiigLiXpmZWKwmcNDDs@eu1.storj.io:7777
========== ACCESS RESTRICTIONS ============================================================
Download  : Allowed
Upload    : Disallowed
Lists     : Allowed
Deletes   : Disallowed
NotBefore : No restriction
NotAfter  : No restriction
Paths     : sj://container/storj.png
========== SERIALIZED ACCESS WITH THE ABOVE RESTRICTIONS TO SHARE WITH OTHERS ===========
Access    : 1bwhhbnrfHx7bbv7VqteuTuvo6RuXzcRT9seRPmgSr4Bkki5hkUFJ2whkhpiTSR7KCUnK5p65eyXRPbp1MjqRWwci9TXE67JSZrXQL44qZE9xMYpXuLYfw3i4fRdSNg
evxmsg7vxX1L4q1n82zV6VBTNGnxxsQmQwHNXjzFuvRt9kVvxZg1P2gR62yi3uCLve9ouu4VYdcaBicAFA6yz7mYRubUFijR9tgovN4u2vLmbhjyyTpJaRq2a937DCBKMy3eBLHAzSn
gq7k2HVUp6cx2Q11sQqhAteF71Lu991Lwxiff1B3ncNBTNnCKBJEpkFPzcvf4PZKF6e7VezBs5qLyeb9D2GiERKDr6rqYjQkGbu6P6WafhxnoSgdWC3XDiuW1TPTcpG6HsKixJyuRz4
nZh2u7obss1rPMJX8dDqzEqpgsuauvssTkqQzafTx5BiyL
========== CREDENTIALS ===================================================================
Access Key ID: jxfcmw7rjnwmz5plozx7btv6qvpq
Secret Key   : j3dax4qkh5z6lmnzc6gisicohs4jwbxw5eajm67a2hfmqas5fm4fi
Endpoint     : https://gateway.us1.storjshare.io
Public Access:  true
========== BROWSER URL ====================================================================
REMINDER  : Object key must end in '/' when trying to share recursively
URL       : https://link.us1.storjshare.io/s/jxfcmw7rjnwmz5plozx7btv6qvpq/container/storj.png
```

Figure 15: Informazioni sul file presente in rete.

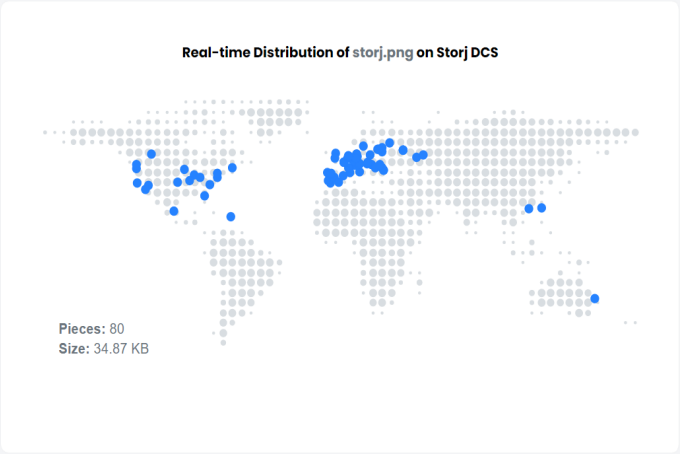


Figure 16: Distribuzione geografica dei pezzi di file tra i nodi della rete.

Ciò che accade a livello framework può essere riassunto nel seguente diagramma:

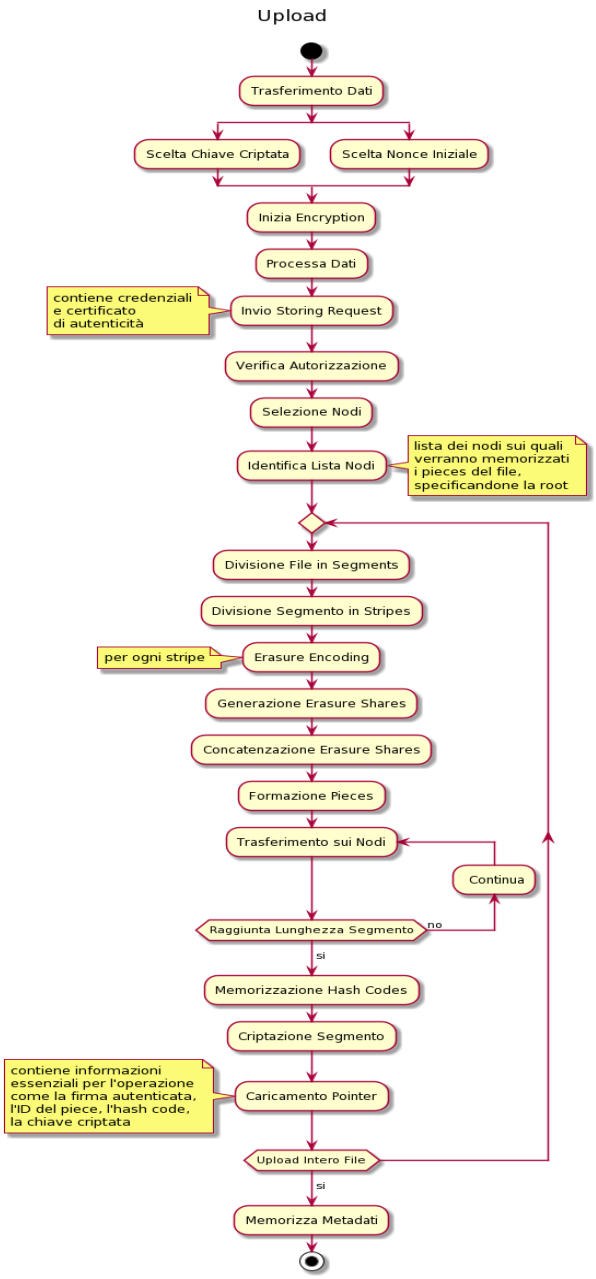


Figure 17: Diagramma di Attività dell'operazione di Upload.

4.3.3 Download

Per effettuare il download di un file presente all'interno di un bucket, una volta che ne è stata verifica l'effettiva presenza tramite comando di listing, è sufficiente eseguire:

```
$ uplink.exe cp sj:/[bucket]/[file] $env:USERPROFILE/Downloads/[file]
```

in modo che il file presente online venga recuperato tramite riassemblaggio dei suoi pieces e scaricato nella directory locale specificata.

Ciò che accade a livello framework può essere riassunto nel seguente diagramma:

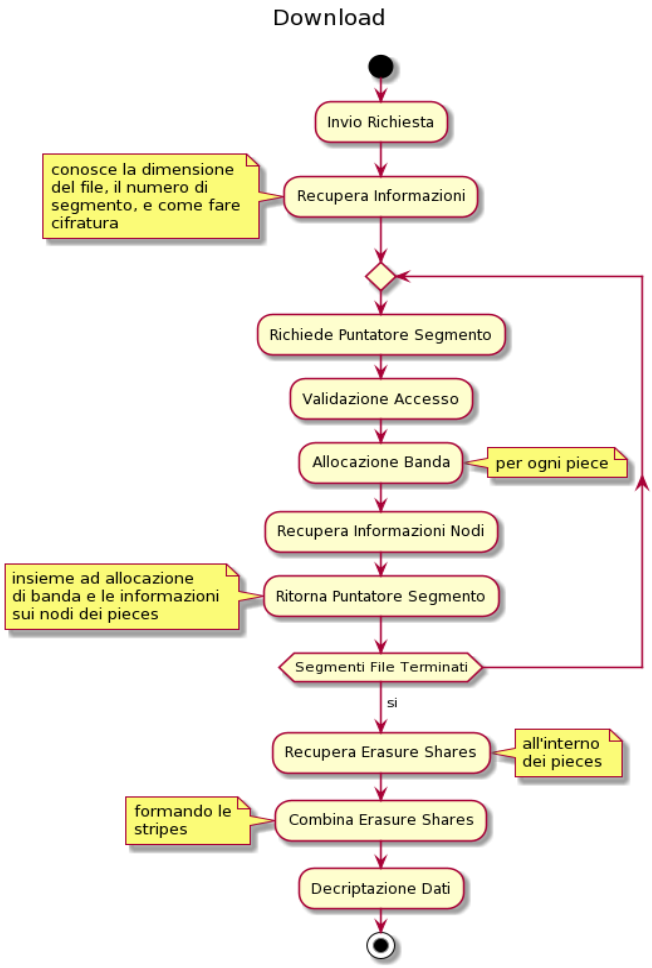


Figure 18: Diagramma di Attività dell'operazione di Download.

4.3.4 Delete

Se si intende rimuovere ed eliminare un oggetto presente all'interno di un bucket e i suoi relativi metadati, sarà sufficiente lanciare il comando:

```
$ uplink.exe rm sj:/[bucket]/[file]
```

per rimuovere ogni traccia del file dalla rete.

Ciò che accade a livello framework può essere riassunto nel seguente schema:

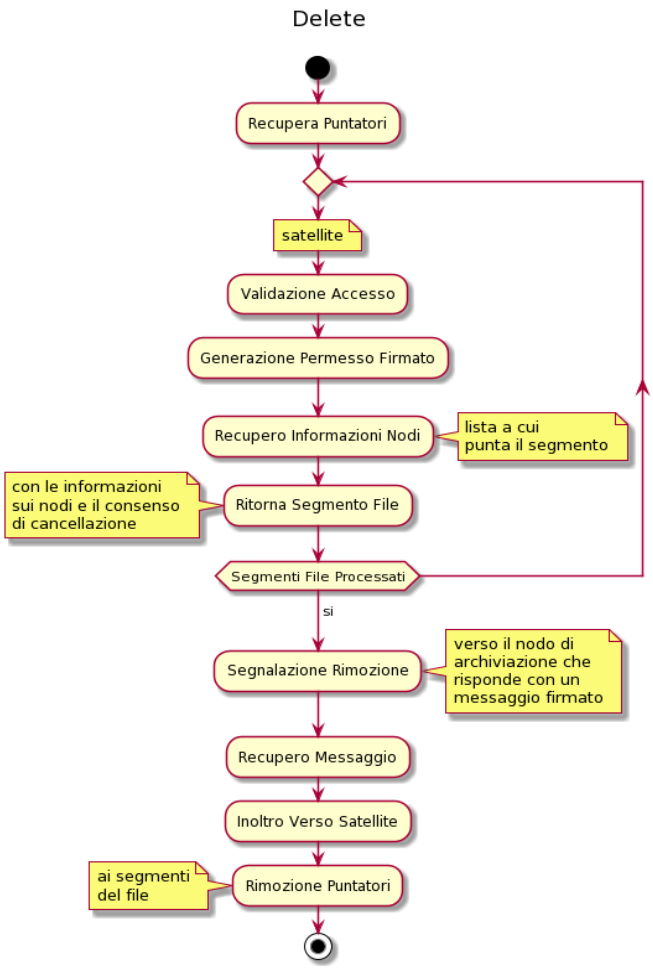


Figure 19: Diagramma di Attività dell'operazione di Delete.

4.3.5 Altre Operazioni

Ulteriori operazioni meno coinvolgenti a livello implementativo, ma comunque messe a disposizione dal framework, e delle quali se ne descrive il funzionamento, sono:

- Spostamento: quando un utente desidera spostare un file, per prima cosa invia una richiesta all'uplink specificando il nuovo path, che provvederà a richiedere tutti i puntatori ai segmenti dell'oggetto da spostare al satellite. Quindi una volta che il satellite ha verificato l'autorizzazione per l'accesso al file, restituisce i metadati dei segmenti. L'uplink, conoscendo a questo punto il percorso del file poiché il satellite ne ha concesso le informazioni, decripterà i metadati utilizzando la chiave derivata dal percorso dell'oggetto. Una volta settata la nuova destinazione del file, i metadata verranno crittografati nuovamente con un'altra chiave derivata questa volta dal nuovo percorso di destinazione. A questo punto si notifica il satellite che dovrà aggiornare i puntatori ai segmenti del file ed eliminare quelli vecchi.

- Copia: quando un utente desidera copiare un file, viene messo in atto lo stesso processo per l'operazione di spostamento, con la differenza che una volta specificato il nuovo percorso del file copiato, non si cambieranno i puntatori ai segmenti del file ma ogni nodo di archiviazione eseguirà una copia di un pezzo del file, alla quale viene associato un nuovo ID. A questo punto viene eseguita un'operazione di upload per fare in modo che la copia del file venga fisicamente creata all'interno del satellite.

- Auditing: il processo di auditing avviene tramite un meccanismo di polling in cui il satellite verifica e controlla che tutte le erasure shares dei pezzi di file che un nodo ospita siano corretti, o meglio non siano stati corrotti o eliminati inaspettatamente. Il satellite provvederà a contrassegnare i nodi di archiviazione che hanno riscontrato problemi nel recupero delle erasure shares.

- Riparazione Dati: il processo di riparazione dati avviene tramite due fasi distinte in cui prima di identificano i file difettosi e poi successivimaente si provvede alla loro riparazione. Tramite meccanismo di polling, il satellite controlla periodicamente ogni nodo di archiviazione, contrassegnando quelli che risultano difettosi, sia per motivi legati alla rete o per motivi dovuti al fallimento di audit. Una volta che sono stati individuati i nodi da riparare, si procede con il meccanismo di riparazione che prevede il download di pezzi di file per ricostruire i segmenti difettosi, o rimpiazzare quelli mancanti (ridondanza). Una volta generati i nuovi pezzi dei segmenti dei file, il satellite seleziona i nodi che durante la verifica iniziale ha riconosciuto come affidabili, e eseguirà un upload dei nuovi pezzi verso di essi, modificando il puntatore ai metadati del file, dato che il percorso di allocazione è stato cambiato.

4.3.6 Demo

Come primo step implementativo è stato creato un apposito script per poter effettuare il setup dell'ambiente di sviluppo in maniera rapida e intuitiva.

```
@echo off
echo.
echo StorJ Test Environment Setup
echo ================================================
echo.
pause
echo Setup Go Module
echo ================================================
echo.
set current_dir=%cd%
go mod init storj_demo
echo.
echo Add Uplink Dependency to Go Module
echo ================================================
echo.
go get storj.io/uplink
```

Le operazioni eseguite dallo script, seguendo un determinato ordine procedurale, sono essenzialmente 3:

1. viene settata come cartella di lavoro, contenente il *workspace*, la cartella in cui viene eseguito lo script di setup.

2. viene creato un **modulo** Go specifico in modo da rendere più agevole la gestione di eventuali librerie utilizzate. Un modulo è essenzialmente un progetto che contiene dei **pacchetti** Go. Infatti i moduli Go sono stati aggiunti in risposta a una crescente necessità di rendere più facile per gli sviluppatori mantenere varie versioni delle loro dipendenze, oltre ad aggiungere maggiore flessibilità nel modo in cui gli sviluppatori organizzano i loro progetti sul proprio computer.

3. import della libreria `storj.io/uplink` che fornisce le principali API per poter interagire con il framework.

Fondamentalmente si tratta di una piccola demo scritta in Go, grazie alla quale si riesce ad interagire con la tecnologia StorJ in maniera rapida, intuitiva e soprattutto molto flessibile. Infatti l'intero framework sembra supportare in maniera super efficiente questo linguaggio di programmazione, al contrario di linguaggi più orientati agli oggetti come Java che hanno bisogno di una fase di configurazione di progetto onerosa.

Sostanzialmente sono state eseguite le operazioni di base che un utente può eseguire utilizzando questo tipo di tecnologia. Il progetto consiste in 4 file principali di cui 3 sono delle classi OOP wrappate secondo sintassi di Go, dato che il linguaggio non è orientato agli oggetti, mentre il file rimasto è il punto di entrata dell'applicazione. Per cercare di mantenere una modularità di codice consistente e cercando di mantenere il codice più flessibile ed estendibile possibile, sono state create delle `type struct`. In pratica si tratta di generare una

classe come una struttura e definire un nuovo tipo di dato (nome della classe) per poterla utilizzare anche all'esterno. Inoltre, sempre all'interno del file dove viene definita la struttura con i suoi campi, sono stati creati i metodi appositi, seguendo la sintassi di Go, per poter eseguire funzioni specifiche sul nuovo tipo appena definito. In questo modo è stato possibile implementare costruttori e metodi di tipi di dato creati ad hoc e poterli utilizzare all'interno dell'app. In particolare le 3 "classi" implementate sono:

- **Account**: questa classe si occupa principalmente della parte di accesso dell'applicazione. Infatti, tramite chiamata al suo costruttore, vengono generate le stringhe di accesso utili per poter iniziare ad interagire con i satelliti di StorJ, le quali possono essere recuperate tramite *getter* appositamente creati. Inoltre dispone di un metodo `retrieveAccess` che permette di recuperare un oggetto Access attraverso le credenziali inserite, in modo da ottenere i permessi per effettuare operazioni verso il cloud network.

- **Project**: questa classe rappresenta il core dell'applicazione. Infatti è stata implementata principalmente per generare, sulla base dei permessi ottenuti dall'Access, un oggetto Project, il quale rappresenta il punto di accesso per poter eseguire operazioni verso il cloud network.

- **Controller**: classe di utility utilizzata per incapsulare tutte le operazioni di interazione utente-storj. In questo modo, il codice relativo al main dell'applicazione risulta più pulito e definito.

Il `main` del progetto si trova nel file principale *storj-demo.go*, il quale si preoccupa soltanto di creare le istanze delle classi sopra citate ed utilizzarne i costrutti in modo da poter interagire con il cloud network di StorJ. E' possibile eseguire la demo spostandosi nella cartella di progetto ed esguendo il comando:

```
$ go run .
```

Inoltre è stato verificato che tutte le operazioni eseguite siano state effettuate correttamente controllando l'apposita dashboard via web, attraverso la quale è possibile verificare i bucket di cui si è in possesso, visualizzarne gli oggetti e i metadati.

4.4 Testing

In questa sezione viene analizzata la fase di testing alla quale è stata sottoposta la tecnologia in questione, cercando di capirne le funzionalità e i comportamenti, anche a fronte di situazioni non usuali. Per interagire con la piattaforma StorJ sono stati scelti **Go** come linguaggio di programmazione, poiché dotato di forte espressività, e la libreria `uplink` (https://pkg.go.dev/storj.io/uplink) come riferimento per le principali API.

4.4.1 Inizializzazione

Anche in questo è stato effettuato un setup dell'ambiente di testing come passo preliminare, in modo da impostare correttamente tutte le varie dipendenze di progetto, ma cambiando l'inizializzazione del modulo Go.

```
    @echo off
    echo.
    echo StorJ Test Environment Setup
    echo ================================================
    echo.
    pause
    echo Setup Go Module
    echo ================================================
    echo.
    set current_dir=%cd%
    go mod init storj_test
    echo.
    echo Add Uplink Dependency to Go Module
    echo ================================================
    echo.
    go get storj.io/uplink
```

4.4.2 Implementazione

Una volta eseguito lo script di inizializzazione dell'ambiente di sviluppo, la cartella di lavoro conterrà i file `go.sum`, contenente una lista delle varie dipendenze di progetto, e `go.mod`, contenente soltanto le dipendenze dirette del progetto, ossia quelle importate direttamente dal modulo. Una volta settato l'ambiente, utilizzando Visual Studio Code come IDE di sviluppo, è stato creato un file contenente un'intera *suite* con tutto il codice con il quale sono stati eseguiti le operazioni di testing. Per eseguire i test quindi, sarà sufficiente posizionarsi con un terminale nella cartella di lavoro contenente i file delle dipendenze di progetto e il file di test, ed eseguire il comando:

```
  $ go test -v
```

per lanciare tutti i test e verificarne la corretta esecuzione. Oltre all'aspetto puramente funzionale dei test eseguiti, è stata dedicata particolare attenzione anche alla loro espressività, in modo che a livello di scrittura risultino del tutto intuitivi e facili da capire.

5 Discussione

Assumendo una visione globale del sistema descritto nelle sezione precedenti, questa parte tratterà una discussione in chiave critica dell'intero framework, cercando di far emergere anche quali potrebbero essere le debolezze del sistema. Come precedentemente discusso, StorJ dispone di caratteristiche che lo rendono uno dei sistemi best-in-class nel mercato attuale. In particolare sono emersi i seguenti punti di forza, che permetto a tale tecnologia di di essere un motore di archiviazione veloce, economico e sicuro:

- E' un sistema cloud decentralizzato basato su *blockchain*, in particolare sfruttando una rete peer-to-peer, ovvero consente lo scambio diretto di dati tra diverse parti all'interno di una rete, senza la necessità di intermediari.

- Mantiene la privacy dei dati memorizzati, ai quali può accedervi soltanto il proprietario. Infatti sistemi centralizzati come Dropbox o Google Drive hanno delle limitazioni notevoli, visto che i dati aggiornati potrebbero risultare ridondanti, la larghezza di banda a disposizione per effettuare tanti accessi al data center potrebbe saturarsi causando problemi di recupero dati, e i dati memorizzati all' interno dei server sono visibili agli admin di sistema.

- Utilizza *file sharding*, ossia la divisione di un file in più pezzi prima di essere inviato in rete. In questo modo si evidenziano due vantaggi principali, poiché da un lato il trasferimento è molto rapido dato che è possibile parallelizzare l'invio dei vari pezzi del file, e dall'altra parte non ci sarà mai nessuno che possiederà l'intero file ma soltanto il proprietario conoscerà la collocazione dei pezzi in cui il file è stato suddiviso.

- E' supportato da processi di verifica di corretto funzionamento del sistema. Infatti se non è possibile recuperare un pezzo di un file dalla rete, poiché è stato distrutto per sbaglio, oppure il nodo è offline, StorJ sfrutta un meccanismo di ridondanza per recuperare comunque una copia del pezzo smarrito, e ripristinarlo. Inoltre per evitare di incorrere in un overhead dovuto al sovra utilizzo di questo meccanismo, è stato implementato un processo di verifica tramite *audits* che permette di verificare se un nodo si sta comportando come ci si aspetta e che tutti i pezzi di file siano consistenti.

- Implementa un meccanismo di file *encryption* grazie al quale riesce mantenere costante e garantire la privacy dei dati memorizzati e dei pezzi dei file archiviati sui nodi di rete. Inoltre permette di gestire in modo accurato il controllo degli accessi sui file stessi.

5.1 Proprietà Fondamentali

Riassumendo è possibile identificare i punti di forza di StorJ in 3 caratteristiche fondamentali:

1. **Disponibilità**: Sebbene la maggior parte dei provider di servizi cloud centralizzati utilizzi varie strategie per fornire protezione contro i guasti delle singole unità, non sono tuttavia immuni da eventi a livello di sistema che possono provocare delle *failure*. Fenomeni atmosferici, interruzioni di corrente, errori dell'operatore, difetti di progettazione, sovraccarico della rete o attacchi possono compromettere interi data center. Infatti, in un data center centralizzato, la possibilità che una singola unità si guasti è altamente correlata alla possibilità che un'altra unità si guasti. In un sistema decentralizzato, al contrario, ogni nodo è gestito da un'entità diversa, in

una posizione diversa, con alimentazione e accesso alla rete separati. Pertanto, la possibilità che un singolo nodo si guasti è quasi del tutto non correlata con la possibilità che si guastino altre unità. Infatti anche se la possibilità che una singola unità si guasti nella rete Storj è maggiore rispetto a un cloud centralizzato, la possibilità di un guasto collettivo è estremamente ridotta.

2. **Sicurezza**: Tutti i dati vengono crittografati lato client prima di raggiungere il sistema dove vengono suddivisi e distribuiti su un gran numero di unità gestite in modo indipendente che fanno parte di una rete molto più ampia di nodi di archiviazione gestita in modo indipendente. Con questa struttura un eventuale attaccante non riuscirebbe ad ottenere alcun risultato in maniera conveniente in termini di tempistiche e prestazioni: ad esempio supponendo che ogni file sia distribuito attraverso 50 drive su disco rigido differenti appartenenti a una rete di 100'000 nodi indipendenti, per compromettere un singolo file, un malintenzionato dovrebbe localizzare e compromettere tutti e 50 i drive di memorizzazione, ciascuno gestito da un provider differente, e appartenenti a un rete con 100'000 potenziali drive su cui il file potrebbe esser stato distribuito, e quindi difficili da individuare. Anche se l'attaccante dovesse riuscire a compromettere tutti i 50 drive, per ricostruire il file dovrebbe decriptare i dati con le chiavi che soltanto l'utente che lo ha memorizzato possiede. Inoltre l'attaccante dovrebbe ripetere la stessa procedura per altri drive differenti per riuscire a compromettere un altro singolo file, il che diventa troppo oneroso.

3. **Efficienza**: Le prestazioni dell'intero sistema suono supportate dalla capacità di StorJ di sfruttare il parallelismo per eseguire le operazioni, dato che i nodi di archiviazione sono indipendenti, riducendo in questo modo la latenza che invece si verifica quando vengono richiesti dati da locazioni lontane rispetto al data center che li ospita. Inoltre il particolare schema di codifica di cancellazione (erasure coding) che utilizza garantisce che le unità lente, le reti lente o con un carico temporaneamente elevato non limitino il throughput.

Forte di queste caratteristiche, presenta ovviamente anche numerose problematiche che verranno successivamente analizzate.

5.2 Vulnerabilità

Utilizzando Kademlia come protocollo di rete peer-to-peer che regola la comunicazione tra i nodi e le modalità di scambio delle informazioni, StorJ è potenzialmente vulnerabile ad un attacco *Spartacus*, conosciuto anche come *dirottamento di identità* (identity hijacking). Qualunque nodo infatti può assumere l'identità di un altro nodo di rete semplicemente copiando il suo ID, e ricevere quindi messaggi e informazioni destinate al nodo di cui si sta rubando l'identità. In questo modo è possibile effettuare attacchi mirati a nodi specifici della rete, che potrebbero potenzialmente entrare in contatto con informazioni

preziose che quindi rischiano di essere compromesse. Per mitigare un attacco di questo genere, è necessario implementare gli ID dei nodi come hash code di chiavi pubbliche e richiedendo che i messaggi siano *signed*, ovvero firmati in modo autentico. Così facendo un eventuale attaccante non sarebbe in grado di generare la chiave privata corrispondente, e perciò non sarebbe in grado di firmare i messaggi e di conseguenza non è abilitato a partecipare alla rete.

Una tipologia di attacco molto frequente nei sistemi distribuiti decentralizzati è l' attacco *Sybil*, che comporta la creazione di una grande quantità di nodi di rete nel tentativo di interrompere le operazioni di rete, dirottando (hijacking) o cancellando (dropping) i messaggi di comunicazione, e quindi disperdendo l'informazione trasmessa. StorJ tenta di mitigare questo tipo di vulnerabilità introducendo un sistema di *reputazione* del nodo, che prevede un periodo di controllo iniziale prolungato. I nodi infatti, prima di potersi considerare attendibili, devono essere popolati da una quantità di dati significativa o diventare membri partecipanti delle tabelle di routing di Kademlia. Questo sistema di controllo, quindi, impedisce a una grande afflusso di nuovi nodi di prendere possesso dei dati provenienti da nodi di archiviazione esistenti senza aver prima dimostrato la loro longevità. Ad esempio dati due nodi di archiviazione, A e B, il nodo di archiviazione B non può entrare nella tabella di instradamento (routing) del nodo di archiviazione A finché il nodo di archiviazione B non può presentare un messaggio firmato da un satellite C di cui il nodo A si fida, affermando che B ha superato un numero sufficiente di controlli (audits) verificati da C.

Se un nodo risulta dannoso, rimane molto difficile da individuare poiché si comporta normalmente ma riesce ad oscurare messaggi ed informazioni che passano attrverso di esso, senza appunto nessuno che se ne accorga (attacco *eclipse*). Storj utilizza hash di chiave pubblica come l'ID del nodo, firme basate su tali chiavi pubbliche e più ricerche di rete distinte (per fare discovery dei nodi partecipanti) per evitare che tutte le connessioni in uscita non raggiungano nodi dannosi. Più grande è la rete, più difficile sarà impedire a un nodo di trovare una parte della rete non controllata da un utente malintenzionato. Dal momento che un nodo di archiviazione o un satellite sono stati introdotti in una parte della rete che non è controllata dall'attaccante in nessun momento, gli hash e le firme della chiave pubblica assicurano che gli attacchi man-in-the-middle siano impossibili. L'unica cosa da fare quindi è assicurarsi che i nuovi nodi vengano opportunamente introdotti in almeno un nodo affidabile della rete durante il processo di bootstrap.

StorJ potrebbe essere vulnerabile a un attacco di tipo *Honest Geppetto* in cui l' attaccante gestisce un gran numero di nodi di archiviazione fittizi sulla rete, accumulando nel tempo sia dati che reputazione. Una volta raggiunta una certa quantità di informazioni, l'attaccante richiama a sé i nodi fittizi, in modo da entrare in possesso dei dati memorizzati, o semplicemente potrebbe cancellare il nodo dalla rete causando perdita di informazione. Per impedire un azione di questo genere è fondamentale analizzare la *correlazione* tra i nodi di archiviazione della rete attraverso tempi di inattività, latenza, analisi di routing, in modo da calcolare la probabilità che due nodi siano gestiti o meno dalla stessa organizzazione. I satelliti perciò devono cercare di distribuire i pieces dei file

sul maggior numero possibile di nodi di archiviazione non correlati. Nonostante la codifica Reed-Solomon utilizzata da StroJ permetta al client di scaricare il numero necessario di pezzi da altri nodi, se più nodi dannosi collaborino per ottenere il controllo di molti pezzi dello stesso file, esso non sarà più recuperabile. Infatti, questo tipo di attacco chiamato *Bytes Hostage*, è specifico per l'archiviazione distribuita, in cui i nodi dannosi si rifiutano di trasferire pezzi di file al fine di estorcere pagamenti aggiuntivi ai clienti.

La misurazione della larghezza di banda con le firme di autenticazione riduce al minimo il rischio per i nodi di archiviazione di diventare inaffidabili, introducendo un processo di verifica che ne limiterà l'esposizione ad eventuali attacchi e permettendogli di rifiutare richieste da satelliti che risultano non affidabili. Tuttavia StorJ non è stato progettato per proteggere i dati da client compromessi. Infatti sebbene i nodi di archiviazione e i satelliti siano progettati per richiedere l'autenticazione tramite firme prima di servire le richieste sui dati, è ragionevole immaginare una modifica del nodo di archiviazione o del satellite che permetterà di eseguire le operazioni a qualsiasi richiedente pagante. Nonostante questo anche in una rete con un satellite senza fiducia, la privacy dei dati non è significativamente compromessa dato che la crittografia avanzata lato client protegge il contenuto del file da letture malevole non autorizzate.

Senza una rigenerazione periodica di verifiche di affidabilità, un nodo di archiviazione può iniziare a superare la maggior parte degli audit senza archiviare tutti i dati richiesti. Invece, è necessaria una striscia casuale di erasure shares da tutti i nodi di archiviazione, perciò, quando un numero sufficiente di nodi di archiviazione restituisce informazioni corrette, è possibile identificare facilmente eventuali risposte errate o mancanti. I nuovi nodi di archiviazione verranno inseriti in un processo di verifica fino a quando non saranno stati superati un numero sufficiente di audits.