Technology Know-How: SeaweedFS

Asia Lucchi
asia.lucchi@studio.unibo.it

7 Dicembre 2021

Indice

Introduzione

SeaweedFS è un sistema di storage distribuito particolarmente efficace nel mantenere molti file di dimensioni ridotte. I suoi punti di forza sono l'alta disponibilità, lo sfruttamento ottimizzato dello spazio a disposizione e la semplicità.

SeaweedFS nasce con l'obiettivo di essere:

- Un metodo di mappatura dei file con rapida ricerca sul disco, ovvero con tempo costante O(1).
- Un sistema personalizzabile, a più livelli, su cui si possono inserire i dati su richiesta.
- Un sistema di storage flessibile che mantiene i dati meno richiesti su un cloud.
- Un File System scalabile in grado di sostituire HDFS.
- Un archivio ad alte prestazioni internamente compatibile con S3.

Svariati utilizzatori di SeaweedFS lo definiscono come una versione di AWS S3 self-hosted, gratuita e meno complessa.

L'analisi effettuata in questo documento si riferisce alla versione 2.78 del sistema[1]. Le release si susseguono velocemente, a volte nel giro di pochi giorni, perciò si è deciso di sceglierne una in modo da non dover aggiornare in continuazione il sistema utilizzato e non verificare la coerenza con le modifiche alla documentazione.

[1] https://github.com/chrislusf/seaweedfs/releases/tag/2.78

Capitolo 1

Architettura

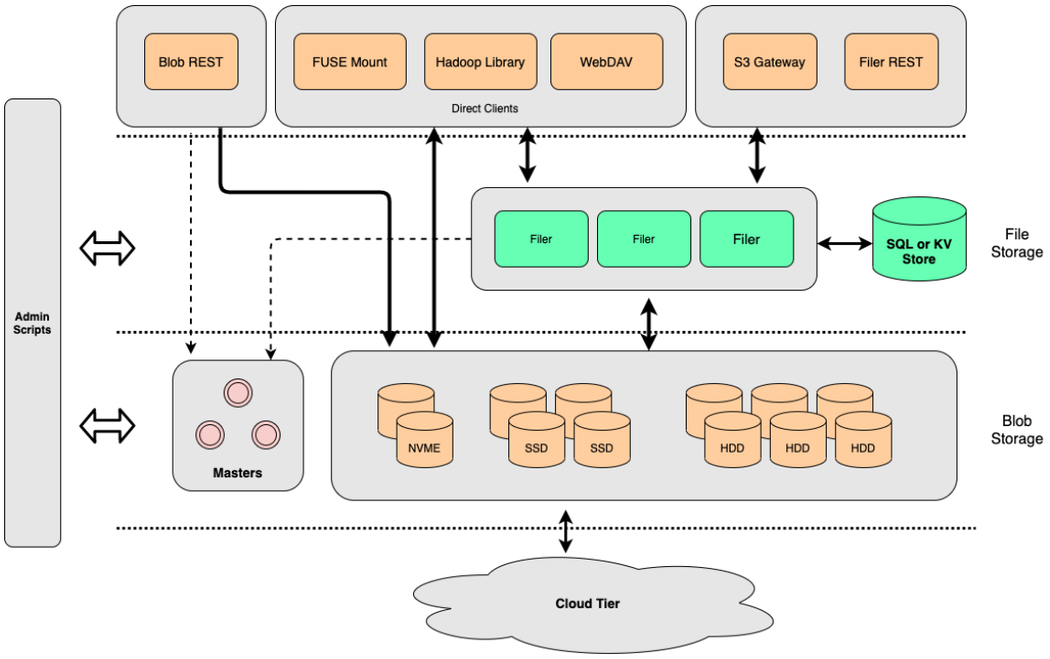


Figura 1.1: Architettura del sistema fornita dagli sviluppatori.[1]

L'architettura di SeaweedFS è costruita su più livelli:

- **Blob Storage** - comprende i server di tipo Master e Volume e il Cloud Tier.
- **File Storage** - aggiunge al Blob Storage i Filer server.
- **Object Storage** - aggiunge al File Storage i server S3, un layer per interfacciarsi con l'esterno e far apparire lo storage come se fosse mantenuto da AWS Simple Cloud Storage.
- **Data Warehouse** - aggiunge al File Storage delle librerie compatibili con Hadoop usabili da servizi come HDFS, Spark, Flink, Presto, Hbase, etc. La maggior parte dei sistemi eseguiti su Hadoop possono essere migrati su SeaweedFS semplicemente aggiungendo una file .jar al classhpath.

[1] https://github.com/chrislusf/seaweedfs/wiki

- **FUSE Mount** - aggiunge al File Storage un'interfaccia FUSE, per esportare un File System virtuale sul kernel linux, montando quindi una propria implementazione del File System (nel nostro caso di tipo SeaweedFS) al posto di quella originale.

1.1 Volume Server

I **Volume Server** possono contenere più *volume*, i quali a loro volta mantengono le informazioni dei singoli file che carichiamo. Di default ogni *volume* è di 30GB, le scritture sono append-only e viene compattato periodicamente per ridurre al minimo lo spazio in memoria sprecato.

SeaweedFS è particolarmente efficace per piccoli file perchè non li salva uno ad uno ma li comprime e raggruppa in *volume*, l'unità di base per varie operazioni come replicazione, bilanciamento, erase coding, ridondanza e scelte di TTL. All'interno del *volume*, ogni file è localizzabile grazie a un indice, un offset che permette di accedervi in lettura con velocità di accesso al disco O(1). L'identificativo del file, il FID, è composto da volume id, file key e file cookie.

Se è attiva la replicazione, ogni scrittura verrà replicata in tutti i *volume* con un forte obbligo di consistenza, ovvero se una qualsiasi delle scritture dovesse fallire, l'intera operazione risulterebbe fallita.

I *volume* possono avere un certo TTL (time to live) impostato, che elimina i file dopo una certa quantità di tempo. Questa funzionalità può essere utile se ad esempio si vogliono mantenere dei log o altre informazioni prodotte periodicamente ma si ha a disposizione uno spazio limitato. Ad ogni singolo file caricato può essere associato un TTL, che determinerà il suo assegnamento a un *volume* con quel TTL o la creazione di un *volume* apposito se non ce n'è uno già presente. Al termine del tempo a disposizione non vengono eliminati i singoli file bensì l'intero *volume*.

Ad ogni *volume* è possibile associare un tag con il tipo di disco, ad esempio SSD, HDD ma anche slowHDD o fastHDD ed al momento della scrittura si può specificare i requisiti in lettura per il file in modo che sia inserito in un *volume* con il disco più appropriato fra quelli in cui c'è ancora spazio.

I *volume* possono essere ulteriormente raggruppati in **Collection**, per rendere più semplice la gestione di tanti file che devono essere trattati allo stesso modo (ad esempio a livello di replicazione, tipo di disco, TTL). Se si usa l'integrazione per S3, ad ogni bucket corrisponde una collezione.

1.2 Master Server

Il **Master Server** è il server a cui ci si rivolge per caricare o prelevare un file, contengono solo le informazioni minimali per reperire il file nell'adeguato *volume*. Queste informazioni sono collezionate dinamicamente come "soft states" e fornite ai client come un DNS per localizzare ogni file.

È possibile avere più server di questo tipo ma in realtà ci si rivolge sempre al medesimo, eletto come leader con un algoritmo di consenso (Raft). Se il leader ha qualche problema, viene prontamente sostituito da un altro master, infatti tutti sono costantemente mantenuti aggiornati sulle stesse informazioni. Al fine di usare l'algoritmo di consenso, all'inizio il numero di *master* deve sempre essere dispari.

Nella modalità di default non c'è replicazione, quindi nel momento in cui un *volume server* non è più disponibile i file che manteneva non sono più reperibili. Ci si rende conto che il server non è più raggiungibile grazie alla mancata ricezione degli **heartbeat** di tale server, ovvero messaggi che ogni *volume server* invia periodicamente al *master* per confermargli il proprio stato attivo.

Ogni richiesta in **scrittura** chiede al master un FID che già corrisponde a una posizione in un *volume* con le caratteristiche richieste, il client invierà poi i dati direttamente al *volume* server con il *volume* indicato, che si occuperà della replicazione nel caso sia richiesta. Il client dovrebbe inoltre mantenere in memoria il FID ricevuto per poter reperire il file in futuro.

Ogni richiesta in **lettura** chiede al master un FID che verrà usato per inoltrare la richiesta direttamente al *volume* server esplicitato.

1.3 Filer Server

Il **Filer Server** è colui a cui ci si rivolge solitamente per fare le operazioni di lettura e scrittura. Contiene tutti i metadati relativi ai file che permettono di organizzare il file system in file e directory, dando l'illusione di avere un file system con la struttura comunemente usata e conosciuta dalla maggior parte degli utenti. Rende possibile fare richieste senza conoscere il FID dei file, ma usandone il path, quindi un modo più intuitivo e ricordabile.

I metadati sono mantenuti in database ampiamente conosciuti ed utilizzati:

- **SQL** - Postgres, YugabyteDB, CockroachDB, MySQL, MariaDB, MemSQL, TiDB
- **Non-SQL** - Redis, Tendis, Ledis, Cassandra, ScyllaDB, HBase, MongoDB, ElasticSearch, Etcd
- **Embedded** - LevelDB, RocksDB

Il Filer è stateless, per ogni operazione si rivolge al database per raccogliere i metadati e successivamente al *volume* server indicato per salvarli. Il Filer si connette anche ai master server per aggiornare le posizioni dei *volume* e richiedere i FID durante le scritture. Per file di grandi dimensioni, si occupa di dividerli in chunk e salvare tutti i metadati riguardanti quali chunk vanno a comporre ciascun file.

L'approccio utilizzato permette al sistema di essere facilmente scalabile per quanto riguarda la gestione dei metadati, infatti è sufficiente aggiungere risorse al database o ulteriori filer servers.

Al contrario degli inode, che mantengono i metadati direttamente con il file stesso, questo metodo permette al file system di non essere limitato nel numero di file che può mantenere ed essere considerato un *key-value-large store*.

1.4 FUSE Mount

Il mount è compatibile con POSIX e mantiene una cache con i metadati che viene asincronamente scritta nel database del Filer. La cache è usata per tutte le operazioni, rendendole veloci dovendo fare solo delle letture in locale.
Gli aggiornamenti dei metadati provenienti dalle operazioni vengono inizialmente registrate sul database del Filer tramite gRPC e poi salvata sulla cache locale, che rimane sempre consistente.

Capitolo 2

Elementi Avanzati

2.1 Replication e Backup

SeaweedFS mette a disposizione svariati modi per fare replicazione o mantenere dei backup con l'obiettivo finale di non perdere i dati e mantenere un'alta disponibilità.

2.1.1 Cluster Replication

I cluster di SeaweedFS possono essere sincronizzati fra diversi data center, creando delle copie identiche e consistenti.

Tutte le modifiche fatte ai metadati sono registrate su **file di log** persistenti, che possono essere controllati ed eseguiti allo stesso modo in altri data center. Il processo *filer.sync* ha proprio il compito di replicare tali log per sincronizzare un nuovo cluster, per fare ciò si registra al cambiamento dei metadati ed usa i log per eseguire gli aggiornamenti ed arrivare a creare una copia consistente del cluster target.

Il procedimento di replicazione mantiene dei check-point periodici in modo da poter essere ripresa da uno di tali check-point nel caso venga interrotta.

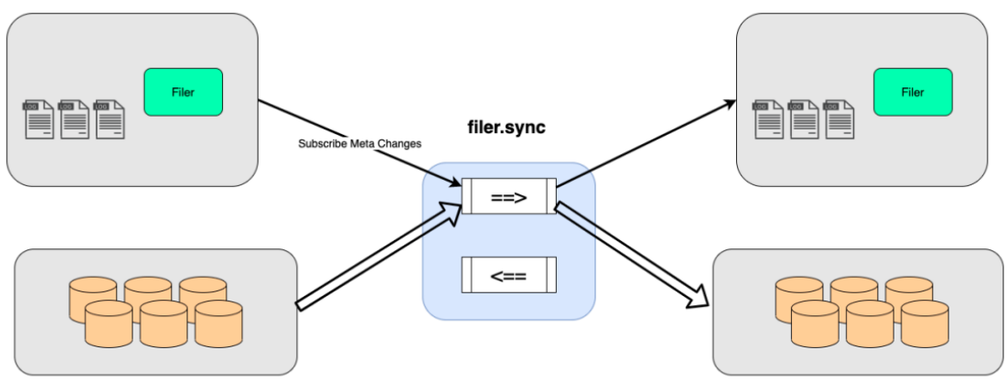


Figura 2.1: Rappresentazione di un'Active-Active Replication[1].

[1]https://github.com/chrislusf/seaweedfs/wiki

La replicazione per ogni gruppo di file può essere di due tipi:

- **Active-Active** - I client si connettono a un servizio di bilanciamento del carico che distribuisce i workload su più server attivi.
- **Active-Passive** - I client si collegano al server principale, che gestisce l'intero workload, mentre un server di backup rimane in standby, attivandosi solo in caso di guasto.

La replicazione può essere bi-direzionale, infatti ogni aggiornamento ha un **identificativo** unico e verrà riconosciuto ed ignorato se è già stato eseguito. Inoltre grazie all'identificativo più di due cluster possono connettersi al medesimo processo *filer.sync* senza rischiare di formare dei loop. Con tanti cluster si possono formare anche topologie più complesse rispetto a quella centralizzata, ad esempio ad anello o a stella.

2.1.2 Cluster Backup

Il procedimento è il medesimo usato per la replicazione appena descritto, ma in questo caso il cluster viene copiato su un server esterno al quale non si accede per una classica operazione di lettura ma viene mantenuto nel caso in cui l'infrastruttura principale dovesse subire delle perdite. In questo caso il processo è denominato *filer.backup* invece che *filer.sync*.

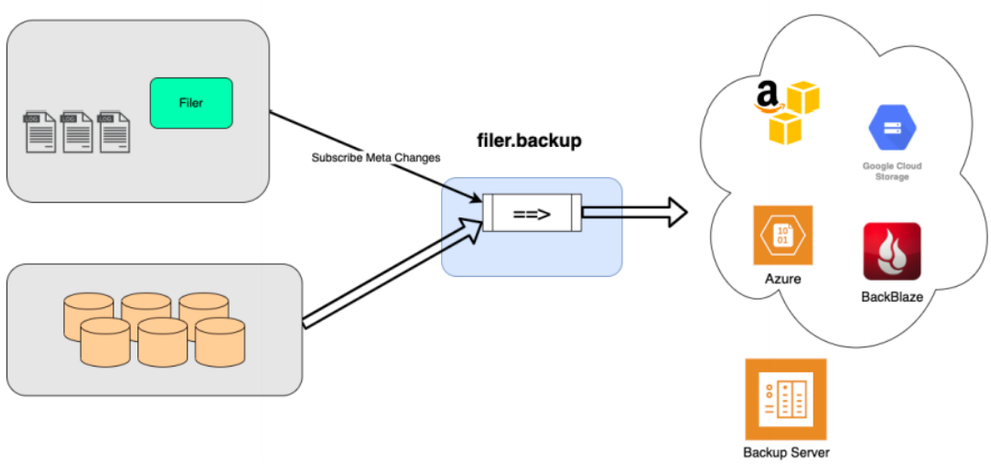


Figura 2.2: Rappresentazione di un Cluster Backup[2].

2.1.3 Volume Replication

Questa tipologia di replicazione è quella più intuitiva, si tratta di avere più copie del medesimo file al fine di restare **consistenti** nel caso una copia venisse a meno ed aumentare la **disponibilità** e la velocità di accesso file andando a reperire la copia

[2]https://github.com/chrislusf/seaweedfs/wiki

più comoda per il client che ne fa richiesta.

La posizione e quantità di repliche che si intende utilizzare è specificabile nel momento di avvio di un master server attraverso il parametro `-defaultReplication`. Il tipo è espresso da tre cifre xyz, in cui:

- **x** indica il numero di repliche in diversi data center.
- **y** indica il numero di repliche in diversi rack dello stesso data center.
- **z** indica il numero di repliche in diversi volume server dello stesso rack.

Il data center ed il rack di appartenenza di ciascun volume server è specificabile al momento della sua aggiunta al sistema, tramite i parametri `-dataCenter` e `-rack`. Alcuni esempi di valori assumibili dalla replicazione sono:

- **000** - Nessuna replica, una copia per ogni file.
- **010** - Una replica su un altro rack dello stesso data center, quindi in totale due copie del file in due rack dello stesso data center.
- **200** - Due repliche in data center diversi, quindi in totale tre copie in tre diversi data center.
- **110** - Due repliche, una in un altro rack e una in una altro data center, quindi in totale tre copie, di cui due nello stesso data center ma in diversi rack.

2.2 Sicurezza

Per quanto riguarda **riservatezza**, il contenuto dei file può essere criptato con una chiave mantenuta come metadata nel database del Filer. In questo modo i *volume* possono essere mantenuti in qualsiasi dispositivo o in cloud senza che siano leggibili in chiaro ed è sufficiente prestare attenzione alla protezione del Filer server.

Per la sicurezza delle **comunicazioni**, nel caso delle chiamate gRPC viene usato TLS mentre l'accesso tramite il protocollo HTTP è disponibile un supporto JWT per il *volume* service.

Per quanto riguarda la gestione dei **permessi**, quando si avvia un master è possibile specificare una *white list* contenente gli IP dai quali accettare le richieste in scrittura.

2.3 Garbage Collector

Quando ci sono delle eliminazioni, esplicite o dovute allo scadere del ttl, lo spazio sul disco non viene immediatamente liberato in modo sincrono bensì di ciò si occupa un Garbage Collector che entra in azione ogni 15 minuti.
Se è necessario che ci sia una pulizia del disco lo si può richiedere esplicitamente tramite l'operazione di vacuum. Questa operazione non è banale in quanto consiste nel fare una copia dei file del *volume* saltando i file all'interno che sono stati eliminati.

Inoltre è presente un background job che controlla lo spazio libero in ogni *volume*, se è più di una certa percentuale (di default il 30%) il job renderà il *volume* readonly, copierà i file non eliminati in un nuovo *volume* e lo sostituirà al precedente.

2.4 Large Files

Solitamente il sistema cerca di leggere e scrivere l'intero file in memoria, con una sola operazione al fine di avere un'alta concorrenza, ma nel caso di file di grande dimensione ciò non è possibile. Per supportare questa tipologie di documenti, SeaweedFS usa due tipi di file:

- **Chunk File** - Tanti, di dimensione gestibile dal sistema, ognuno contiene una parte del file di grandi dimensioni.
- **Chunk Manifest** - Un file json in cui è indicato il nome del file, il tipo, la dimensione ed una lista di chunk file con riportato il FID e la dimensione di ognuno.

Nella fase di **scrittura**, buona parte della computazione sarà delegata al client. Esso ha il compito di dividere il file in chunk, caricarli uno ad uno e salvarne il FID in un'apposita struttura json che rappresente il Manifest e sarà infine caricata a sua volta specificando il parametro `cm=true`.

In fase di **lettura** bisognerà fornire il FID del Manifest, tramite il quale il sistema andrà a reperire i FID dei Chunk File, li scaricherà e ricostruirà il file target.

La dimensione massima del chunk file non è specificata, i chunk possono essere anche di diversa grandezza ma è fondamentale che si possa mantenere l'interno chunk in memoria e alla stesso tempo non si divida il file in troppi chunk di piccola estensione.

La divisione in chunk può essere eseguita automaticamente specificando il parametro `-maxMB` per indicare la dimensione di ciascun Chunk File durante l'azione di upload, che restituirà direttamente il file id del Manifest.

2.5 Erasure Coding

L'Erasure Coding è un metodo di protezione dei dati in cui i file vengono divisi in frammenti, espanso con informazioni codificate ridondanti (blocchi di parità) poi archiviati in diversi dispositivi nel sistema. Nel caso uno di essi non sia più disponibile o venga corrotto, il file può essere ricostruito dai frammenti mantenuti nelle altre macchine, perciò viene proposto come alternativa alla replicazione.

L'implementazione utilizzata in SeaweedFS è la 10.4 Reed-Solomon e prende spunto dall'articolo Facebook's Warm BLOB Storage System[3].

Questo metodo viene messo in azione sui dati ***cold***, ovvero quelli che sono stati inseriti da un po' di tempo e non sono utilizzati spesso quindi non è conveniente

[3]https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-muralidhar.pdf

mantenere il dato in memoria normalmente ma è preferibile spezzettarlo e caricare i frammenti su vari server.
I dati che invece non sono interessati dall'Erasure Coding sono quelli ***hot***, i più usati, che solitamente corrispondono a quelli caricati più recentemente e vanno interamente registrati in un unico *volume* in modo da essere immediatamente disponibili.

I *volume* sono suddivisi in chunk da 1GB e per ogni 10 chunk vengono codificati 4 parity chunk. Un *volume*, che solitamente è di 30GB, verrà codificato in 14 frammenti EC ognuno di dimensione 3GB, composto quindi da 3 chunk. La dimensione complessiva finale sarà di 1.4x rispetto quella occupata originariamente dal *volume*. Nel caso in cui il *volume* sia di dimensione minore ai 10GB, si usano chunk da 1MB.

I 14 frammenti dovrebbero essere distribuiti fra i dischi, i server e rack nel modo più sparso possibile per proteggere da svariati tipi di failure. Una lettura può essere eseguita se sono presenti almeno 10 frammenti del *volume*.

La lettura rimane O(1) nella maggior parte dei casi dato che solitamente un file ha dimensione minore di 1GB ed è quindi contenuto interamente in un chunk o in due nel caso limite. Fa eccezione il caso di file di grandi dimensioni, che devono seguire una procedura diversa tramite Manifest.

I vantaggi che il sistema può trarre dall'Erasure Coding (EC) sono:

- **Efficienza nello storage**: è possibile recuperare l'intero file anche se vanno perduti 4 frammenti con una dimensione totale per file di 1.4x rispetto al file originale. Considerando che i 14 frammenti siano in posizioni diverse, ciò significa che il meccanismo è resistente alla failure di 4 server contenenti parti del file. Confrontando con la replicazione, si dovrebbero fare 5 repliche per avere la possibilità di perderne 4 ma reperire comunque il file, occupando uno spazio per file di 5x rispetto al file originale. Con la stessa capacità di recupero da una failure, usando l'EC al posto della replicazione c'è un risparmio totale di 3.6x di spazio.
- **Velocità in lettura**: usando blocchi della dimensione più indicata fra 1GB e 1MB in base all'entità dei dati da scaricare.
- **Ottimizazzione della gestione di piccoli file**: non c'è alcun requisito di dimensione minima per l'uso del meccanismo di EC.
- **Alta disponibilità**: i dati sono reperibili con una velocità ragionevole anche quando non sono rintraccabili fino a 4 frammenti.
- **Efficienza nell'uso di memoria**: utilizzo minimo, gli indici non sono caricati nella memoria dei *volume* usata per i file.
- **Startup veloce**: grazie all'omissione degli indici, che non è necessario caricare in memoria.
- **Posizionamento Rack-Aware**: i frammenti sono adeguatamente suddivisi fra i rack (oltre che fra i Data Center) in modo da massimizzare l'utilità di adozione il meccanismo di EC nel caso in cui uno o più rack siano completamente disconnessi.

- **Flessibilità nel layout dei server**: gestice i frammenti a livello di volume in modo da non dipendere dalla quantità dei server o dei rack, bensì di adattarsi al loro numero. Se sono presenti meno di 4 server, l'EC protegge da failure del disco, altrimenti protegge da failure di server. Se ci sono più di 4 rack protegge dalla failure di interi rack.

Alcuni lati negativi di questo approccio sono:

- Se alcuni dei frammenti non sono più disponbili, il caricamento del file è più lento in quanto lo deve in parte ricostruire.
- La ricostruzione dei frammenti mancanti richiede la trasmissione dell'intero *volume*.
- Non è supportato l'aggiornamento corrotto di frammenti, solo la loro eliminazione.
- Per compattare lo spazio nei *volume* bisogna riconvertire i file alla forma originale.

2.6 Optimization

Di seguito sono riportate alcune metodologie proposte dallo sviluppatore di SeaweedFS per renderne l'utilizzo ottimale:

- **Usare un indice LevelDB**
  Quando si inizializza un *volume* server, è possibile specificare il tipo di indice da usare con il parametro `-index`. Il default è `memory`, che rende veloci gli accessi a discapito dell'inizializzazione, che deve caricare tutti gli indici in memoria. Se si vuole favorire uno start up veloce penalizzando la velocità di accesso si può usare `leveldb`. LevelDB è un database NoSQL che utilizza il modello chiave-valore. Anche se non è il default, questa soluzione è consigliabile in quanto il rallentamento è insignificante rispetto ai tempi d'attesa già presenti dovuti alla latenza della rete.

- **Preallocare spazio sul disco**
  Nel momento di inizializzazione di un *volume* server può essere utile specificare lo spazio che si intende usare in modo da assicurarsi che i dati vengano registrati su blocchi adiacenti, facilitandone sia la lettura che la scrittura. Si usa il parametro `-volumePreallocate` per attivare questa funzionalità nei sistemi Linux con un FS che lo rende possibile (ad esempio XFS, ext4, Btrfs, etc).

- **Aumentare le scritture concorrenti**
  Di defualt, il numero dei *volume* è gestito dal sistema automaticamente e se non c'è replicazione è inizialmente di 7 *volume*, finchè non si esaurisce lo spazio. Indicano di creare più volume sin dall'inizio, le scritture saranno più veloci perchè sarà possibile eseguirle su più *volume* concorrentemente.
  L'aumento è effettuabile in due modi:

  – Con il comando `curl http://<master-ip>:<master-port>/vol/grow?count=<volume-quantity>`.Inoltre si possono specificare la collezione, il dataCenter, il rack o il tipo di replicazione dei *volume* che si vogliono includere nella modifica.

  - Nel caso si usi un file master.toml, si può modificare il parametro direttamente a livello di configurazione prima di avviare il master.

- **Aumentare le letture concorrenti**
  Ciò è ottenibile sia aumentando il numero di **volume** con uno dei metodi sopra indicati, sia aumentando il numero di repliche e la loro distribuzione.

- **Aggiungere hard drive** per aumentare lo spazio a disposizione.

- **Aumentare il limite di file leggibili dall'utente**
  Normalmente al sistema è richiesto di aprire solo alcuni file contemporaneamente, quindi il limite di default che solitamente è 256 o 1024 non crea problematiche. Quando il sistema viene usato in produzione è possibile avere molte richieste dalla rete che necessitano l'apertura di più file, in questo caso bisogna modificare il parametero di sistema ad un numero maggior, ad esempio `ulimit -n 10240`.

- **Creare chiavi personalizzate al posto di lasciarle generare al sistema.**
  Ogni chiave è composta da tre parti:

  - `volume_id`: il *volume* in cui si posizionerà il file
  - `needle_id`: un numero crescente
  - `file_cookie`: un numero random

  Per scegliere il `volume_id` è necessario assicurarsi che il *volume* prescelto abbia abbastanza spazio libero per contenere il file, invece il `needle_id` e il `file_cookie` si possono assegnare a piacimento avendo l'accortezza di mantenere il primo dei due crescente nella maggior parte dei casi. Ciò non è strettamente necessario ma per mantenre una struttura dati efficiente è meglio mantenerla come regola.
  La personalizzazione può essere utile nel caso in cui i file abbiano già degli id univoci assegnati con le caratteristiche adatte per questo scopo senza bisogno di generarne di nuovi.

- **Ottimizzare il caricamento di file di grandi dimensioni**
  Innanzitutto è auspicabile usare il metodo di auto-split reso disponibile dal sistema. Se il caricamento dovessere non andare a buon fine perchè richiede un tempo superiore ai 30 secondi, è necessario aumentare il tempo massimo di attesa attraverso il parametro `-idleTimeout` al momento dello startup del *volume* server.

- **Usare le collezioni come Name Space** per rendere più ordinata la composizione dei *volume*. Ad esempio usando le collezioni "Documenti", "Immagini" e "Log" per suddividere queste tre tipologie di file in *volume* diversi e dedicati.

- **Fare logging per monitorare la situazione**
  SeaweedFS supporta glog. In qualsiasi comando `weed` usato da shell è possibile specificare la directory in cui mantenere i log relativi con il parametro `-logdir` e il livello di dettaglio richiesto nei log tramine un valore numeri da 0 a 4 assegnato al parametro `-v`.

Capitolo 3

Metodi di accesso ai file

Per l'accesso ai file è possibile usare Java, attraverso una libreria ottenuta da quella usata per l'integrazione con Hadoop, inoltre i server Master, Volume e Filer mettono a disposizione delle API gRPC liberamente accessibili. Il metodo d'accesso diretto più semplice è sicuramente attraverso richieste HTTP, usando un browswer o `curl`.

In questo capitolo sono analizzatti i metodi di accesso diretto tramite HTTP. Queste istruzioni possono essere utili per monitorare lo stato del file system, i volumi attivi sui diversi server, le loro caratterestiche di replicazione e ttl.
Ad ognuna delle istruzioni è possibile aggiungere il parametro `pretty=y` per ricevere l'output in un formato più leggibile (JSON).

3.1 Master Server API

Di seguito sono elencate le operazioni effettuabili con curl:

- Assegnamento di una chiave.
  `curl "http://<master-ip>:<master-port>/dir/assign"`
  Restituisce l'url del server in cui si trova e il FID.

- Ricerca di un *volume*.
  `curl http://<master-ip>:<master-port>/dir/lookup?volumeId=<volume-id>`
  Restituisce l'url del server in cui si trova.

- Forzatura dell'esecuzione del garbage collector.
  `curl "http://<master-ip>:<master-port>/vol/vacuum"`

- Pre-allocazione di *volume* per fare scritture concorrenti invece che sequenziali.
  `curl "http://<master-ip>:<master-port>/vol/grow?count=<how-many-volume>"`

- Eliminazione di una collezione.
  `curl "http://<master-ip>:<master-port>/col/delete?collection=<coll-name>"`

- Controllo dello stato di un sistema.
  `curl "http://<server-ip>:<server-port>/cluster/status"`
  Restituisce una struttura dati che indica chi è il leader e chi sono i peer nel sistema indicato.

- Controllo dello stato dei *volume* scrivilibili.
  `curl "http://<master-ip>:<master-port>/dir/status"`

Restituisce una lista di data center, ognuno con la sua lista di rack, ognuno con la sua lista di *volume* server.
Per ogni *volume* server, è indicato quanti *volume* contiene, qual è il numero massimo che può contenere, quanti ancora ne può inizializzare e la sua posizione tramite url (ip e porta).
Inoltre sono riportate tutte le tipologie di *volume* disponibili in base alla collezione di cui fanno parte, il tipo di replicazione e il ttl, per ognuna è presente una lista con i *volume* che ne fanno parte.

- Controllo dello stato di tutti i *volume* server.
  `curl "<master-ip>:<master-port>/vol/status"`
  Per ogni *volume* fornisce svariate informazioni utili, ad esemio l'id, lo spazio disponibile, l'eventuale ttl, il tipo di disco, quanti file mantiene, il tipo di replicazione, etc. Come il comando precedente, raggruppa i *volume* server in rack che sono poi raggruppati a loro volta in data center.

Nella lista di comandi sono spesso citati `<master-ip>`, che di default è localhost, e `<master-port>`, che di default è la 9333.

Inoltre è disponibile un'interfaccia grafica denoniminata **Master UI** che mostra lo stato del master e varie informazioni utili sullo spazio residuo e i *volume* server a lui connessi, accessibile aprendo la pagina web che si trova all'indirizzo `http://<master-ip>:<master-port>`. In figura c'è un esempio di tale interfaccia dopo aver inizializzato master e due *volume* server con la procedura riportata nel capitolo precedente.

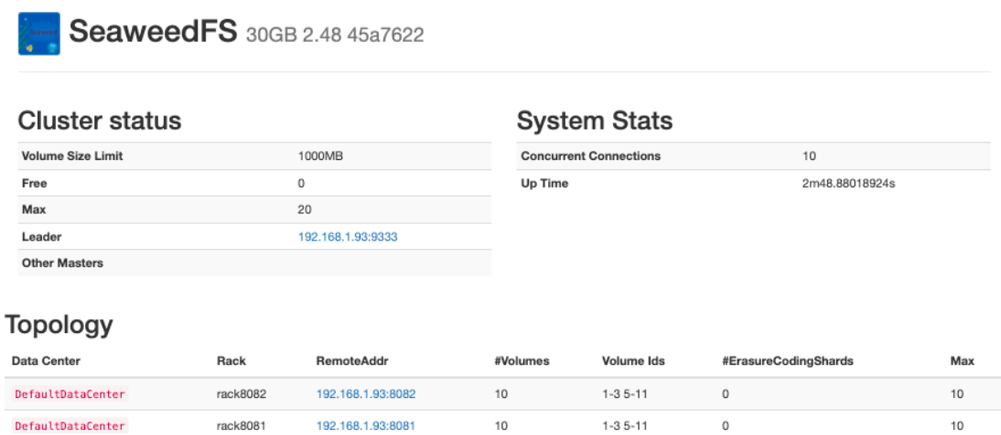


Figura 3.1: Esempio di Master UI.

Sono indicati il limite di dimensione per ogni *volume*, quanti sono i *volume* liberi e quanti quelli totali utilizzabili, qual è l'indirizzo ip del leader fra i master, quante sono le connessioni concorrenti e da quanto tempo è attivo il sistema. Inoltre per ogni volume server è specificato il Data Center, il Rack, l'indirizzo ip con la porta, il numeo di volume che contiene, gli id di tali volume e gli eventuali frammenti di EC presenti.

3.2 Volume Server API

Di seguito sono elencate le operazioni effettuabili:

- Lettura di un file tramite browser con `http://<volume-ip>:<volume-port>/<FID>`.
  É possibile usare anche `http://<master-ip>:<master-port>/<FID>`, che viene ridirezionato verso sull'indirizzo sopra citato dopo aver reperito tramite il master quali siano ip e porta del volume server in cui risiede il file identificato dal FID.
  Nel in cui il file sia un'immagine, è possibile specificare le dimensioni direttamente nell'indirizzo con i parametri `width`, `height` o `mode`.

- Caricamento di un file in una FID conosciuto.
  `curl -F file=<filepath> http://<volume-ip>:<volume-port>/<FID>`
  In questo caso l'indirizzo ip, la porta e il FID possono essere ottenuti con un'operazione di assegnamento eseguita sul master. Viene restituita la dimensione occupata dal file sul volume, che potrebbe essere minore rispetto a quella originale grazie alla compressione automatica.

- Caricamento diretto di un file.
  `curl -F file=<filepath> http://<master-ip>:<master-port>/submit`
  In questo caso vengono fatti assegnamento di una chiave e upload in una sola operazione, che restituisce il FID, l'url e la dimensione occupata dal file.

- Eliminazione di un file.
  `curl -X DELETE http://<volume-ip>:<volume-port>/<FID>`
  In questo caso è necessario specificare ip e porta del volume, non si può usare il master.

- Lettura del Manifest di un large file che è stato suddiviso in chunk.
  `curl http://<volume-ip>:<volume-port>/<FID>?cm=false`
  Restituisce il json corrispondende al Manifest.

- Controllo dello stato di un *volume* server.
  `curl "http://<volume-ip>:<volume-port>/status"`
  Restituisce la lista di *volume* che lo compongono, per ognuno riporta l'id, la dimensione, il tipo di replicazione, la versione, il numero di file che contiene, quanti sono i file ed i byte eliminati e se è in sola lettura o è scrivibile.

Dalla Master UI mostrata nel paragrafo precedente è possibile accedere anche ad una **Volume Server UI** cliccando sul *volume* server di interesse.

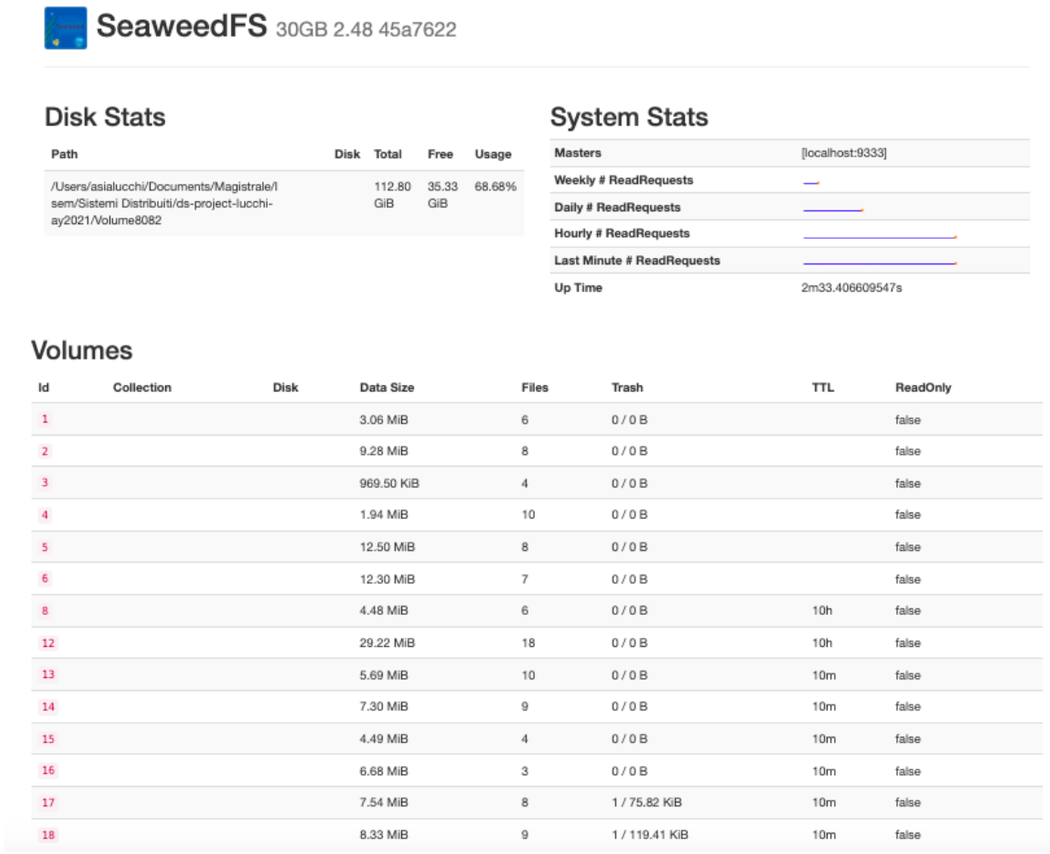


Figura 3.2: Esempio di Volume Server UI.

Anche in questo caso sono presenti alcune statistiche generali del server, sia per quanto riguardo lo spazio libero sia le letture effettuate nel tempo. Per ogni volume è specificato l'id, la collezione, il tipo di disco, la dimensione, il numero di file che contiene, quanti sono quelli cancellati (in attesa del gargabe collector), eventuale TTL e la scrivibilità.

Capitolo 4

Demo

4.1 Utilizzo da shell

Il primo approccio alla tecnologia comprende l'esecuzione di un master service e due *volume* service eseguendo il seguente codice da terminale (`setup_first_demo.sh`).

```
./weed master -mdir="./Master" -volumeSizeLimitMB=1000
./weed volume -mserver="localhost:9333" -dir="./Volume8081"
    -max=10 -port=8081
./weed volume -mserver="localhost:9333" -dir="./Volume8082"
    -max=10 -port=8082
```

Tale script sfrutta l'eseguibile `weed`, ottenuto scaricando la Release dal repository ufficiale[1], in particolare negli esempi è stata usata la versione 2.78 come citato nell'Introduzione.

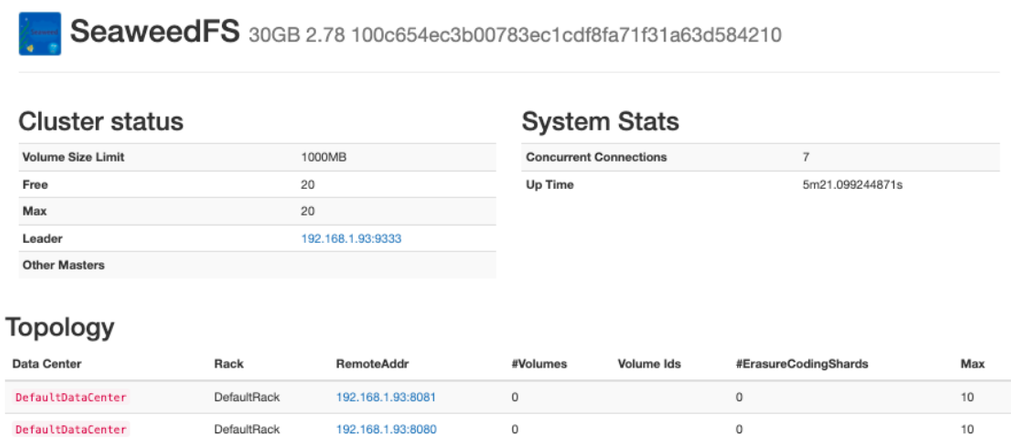


Figura 4.1: Informazioni visualizzate nell'UI usando i comandi sopra indicati.

Nell'esempio, nel primo comando viene utilizzato il parametro `-volumeSizeLimitMB` per ridurre la dimensione di ogni volume da 30 a 1 GB a causa dei limiti di spazio della macchina usata per effettuare le prove.

[1] https://github.com/chrislusf/seaweedfs/releases

Il secondo comando invece usa il parametro `-max` per indicare il numero massimo di *volume* da usare ed esplicita il master service a cui collegarsi con il parametro `-mserver`, il quale se non specificato diversamente si trova sulla porta 9333. I due *volume service* vengono invece messi in esecuzione rispettivamente sulle porte 8081 e 8081, specificate con il parametro `-port`.

In questo caso master e i *volume* si trovano sulla stessa macchina, ma in ottica distribuita il volume potrebbe trovarsi su una macchina diversa e collegarsi al proprio master specificando un indirizzo IP invece di localhost.

Come si può notare in Figura 5.1, a questo punto nessuno dei *volume* service caricati contiene effettivamente dei *volume*, che saranno creati solo al momento dell'inserimento di file. Sempre usando il terminale, si può procedere con il caricamento di file o cartelle usando il comando di upload (`upload.sh`).

```
./weed upload -dir="./TestDir"
```

Usando questo comando otteniamo, oltre al caricamento dei file, una stringa di log per ogni file che ci indica varie informazioni, in particolare dove esso è stato caricato e quindi come reperirlo. Di seguito è riportato un esempio di alcuni log.

```
[{"fileName":"Quickstart.json","url":"192.168.1.93:8080/5,03e00b6bae0a",
    "fid":"5,03e00b6bae0a","size":2349}]
[{"fileName":"README.md","url":"192.168.1.93:8081/2,03e1150de245",
    "fid":"2,03e1150de245","size":50519}]
```

Il primo file è stato caricato nel volume 5, che si trova nel volume server alla porta 8080, mentre il secondo è stato caricato nel volume 2, che si trova nel volume server alla porta 8081.

La cartella *TestDir*, che viene caricata per intero, ha dimensione 2,12 GB e contiene due file di grandi dimensioni e circa 180 file di piccole dimensioni, sotto i 200KB.

Lo spazio occupato in seaweedfs è leggermente inferiore, probabilmente ciò è dovuto alla compressione dei file di grandi dimensioni combinata però con l'inserimento di un piccolo overhead per ciascun file.

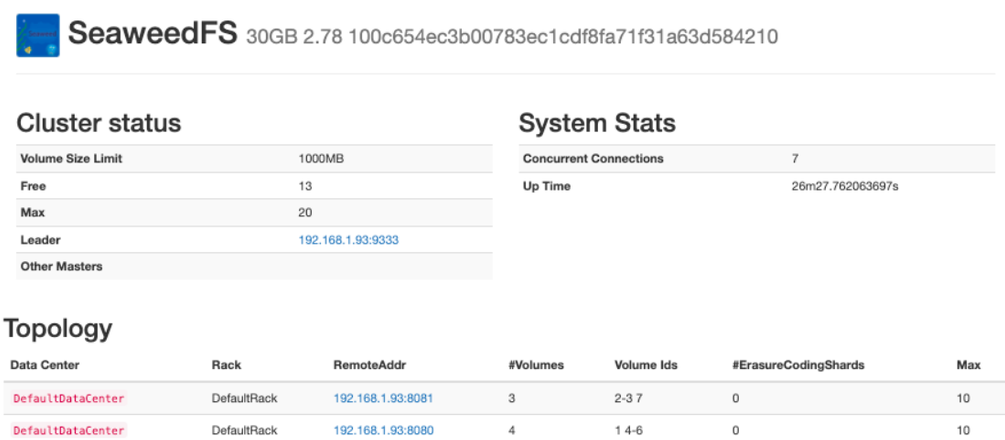


Figura 4.2: UI del master una volta che sono stati caricati i file.

Dalla figura si può innazitutto notare la come l'id di ogni *volume* è univoco nell'intero sistema gestito dal master e non all'interno del singolo *volume server*. Ciò permette **Location Transparency**: ogni volume è identificabile a prescindere dalla posizione fisica in cui si trova e si può spostare facilmente.

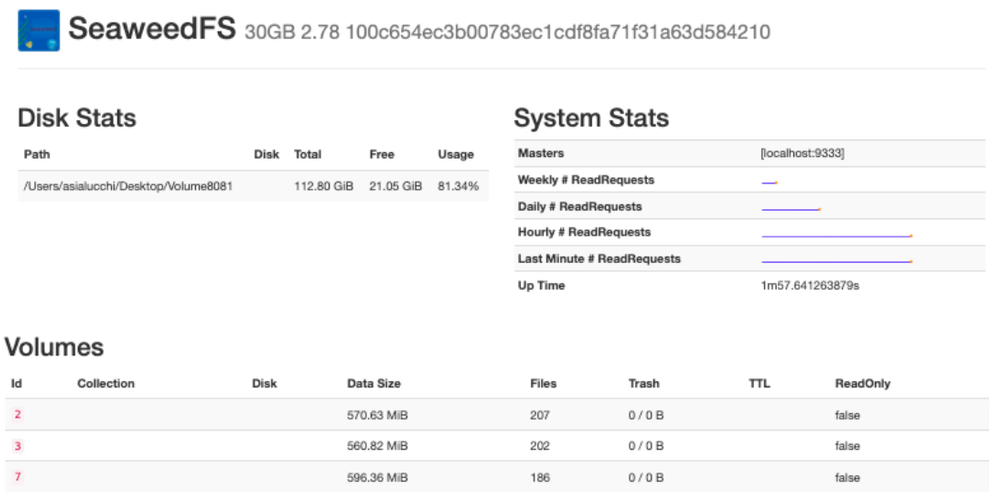


Figura 4.3: UI di uno dei volume server una volta che sono stati caricati i file.

Per mantenere i file di *TestDir* al sistema basterebbero 3 *volume* in termini di spazio, dato che ogni *volume* ha dimensione massima di 1GB. In realtà ne vengono creati un numero ben maggiore ed in ognuno di essi la quantità di spazio occupato è decisamente bassa, in questo caso circa un quarto dello spazio disponibile per *volume*. Inoltre i *volume* sono stati equamente distribuiti fra i due *volume server* inizializzati.

La motivazione di queste scelte non è dettagliatamente descritta nella documentazione, ma tenendo in considerazione che la disponibilità è uno dei punti di forza del file system, probabilmente questa soluzione è atta a promuoverla.

4.2 Java Client

Per comunicare con il File System tramite un client Java è necessario avviare anche un Filer Server che permette di organizzare i file a livello logico in un albero di cartelle e sottocartelle.

Per poter usare il Filer, si aggiunge nella directory di esecuzione un file `filer.toml` che indica come configurare il Filer, generato tramite il comando `weed scaffold -config=filer -output="."`. Inoltre, rispetto alle istruzioni usate nel paragrafo precedente, è presente un'ulteriore istruzione `./weed filer` che ne permette l'avvio (la sequenza completa di istruzioni si trova in `/scripts/filer/setup.sh`).

Anche il Filer ha una dashboard visualizzabile all'indirizzo `http://<filer-ip>:<filer-port>/`, di default l'ip è `localhost` e la porta è la `8888`.

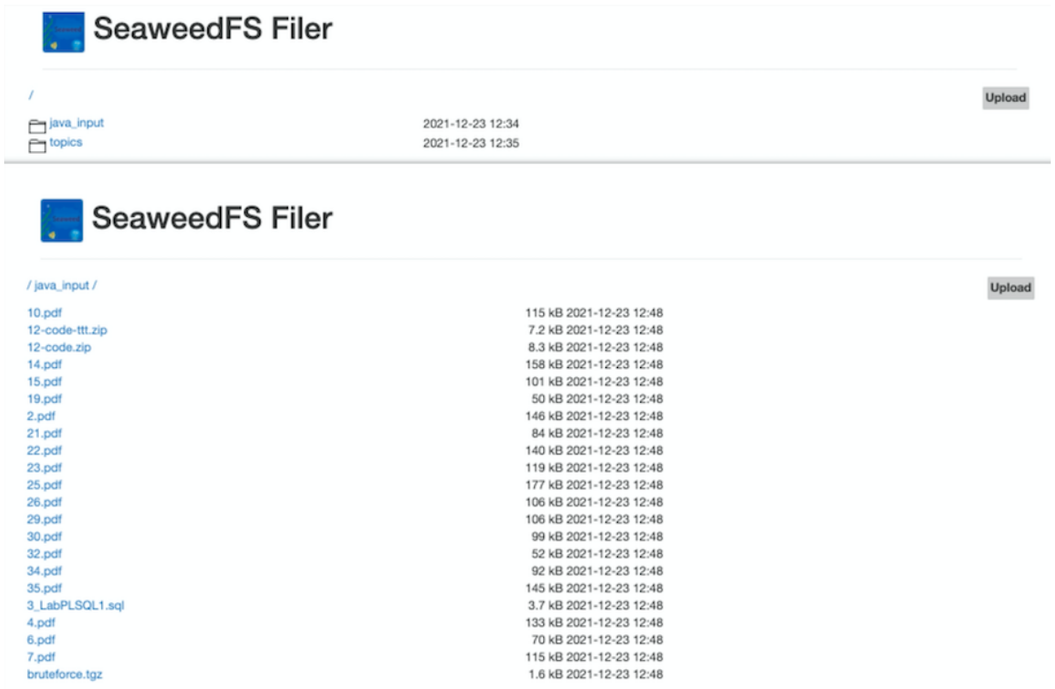


Figura 4.4: UI del Filer, visualizzazioni della cartella principale e della cartella `java_input` dopo aver effettuato l'upload di alcuni file.

Una volta lanciati i servizi che rendono disponibili master, volumes e filer è possibile gestire il File System tramite Java sia con il framework `gRPC` per caricare e scaricare file che usando il framework `REST`, sfruttando le API descritte nel capitolo precedente.

In Java si crea una client per tale Filer Server che esegua chiamate gRPC sulla porta di default `18888`. A questo punto si possono sfruttare:

- **SeaweedOutputStream** per caricare file specificando il server di riferimento e il percorso per raggiungerlo all'interno del Filer, infine copiando il file di cui si desidera fare l'upload nell'output stream.
- **SeaweedInputStream** per scaricare file specificando il server di riferimento e il percorso di per raggiungerlo all'interno del Filer, infine copiando il contenuto dell'input stream in un percorso predefinito in locale.

Inoltre sfruttando REST si possono inviare tutte le richieste POST, GET, PUT e DELETE indicate nelle API tramite un client http, in questo caso si usa al porta di default `8888` per comunciare con il Filer Server.

Nella cartella `java` si può trovare un semplice programma esplificativo che usa i meccanismi appena descritti caricando dei file in una cartella del Filer Server, eliminandone alcuni e scaricando nuovamente quelli rimasti.

Capitolo 5

Testing

In questo capitolo si utilizzano alcuni dei meccanismi descritti in precedenza, mostrando i comandi che permettono di metterli in atto e l'evidenza sulla UI o sui log del loro funzionamento. Ho scelto di usare la shell per dirigere il sistema in quanto mi è sembrato il modo più comodo e flessibile per apportare al volo modifiche che mi portassero ad evidenziare il meccanismo voluto. Tutte gli script usati in questo capitolo si possono trovare nella directory `scripts`.

5.1 TTL

In questa caso si suppone di aver eseguito tutti i comandi riportati nel capitolo 4. Si caricano nuovamente i medesimi file, questa volta però inserendo un time to live di 10 minuti tramite l'apposito parametro `-ttl`, trascorsi i quali i file caricati verranno eliminati.

Nelle figure è riportata la situazione che si verifica una volta inseriti i dati.

```
./weed upload -dir="./TestDir" -ttl=10m
```

Usando questo comando almeno un nuovo volume verrà dedicato ai file con ttl di 10m e non potrà contenere file con ttl diverso.

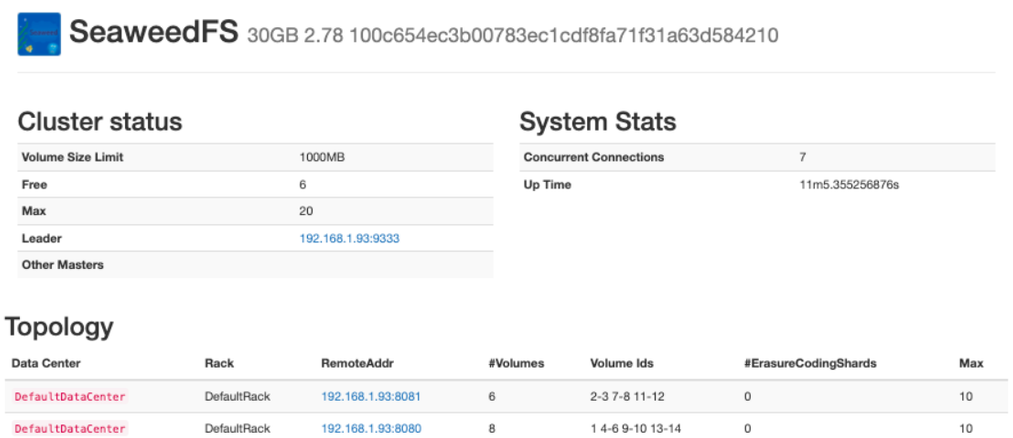


Figura 5.1: Master una volta caricati i file con ttl a 10m.

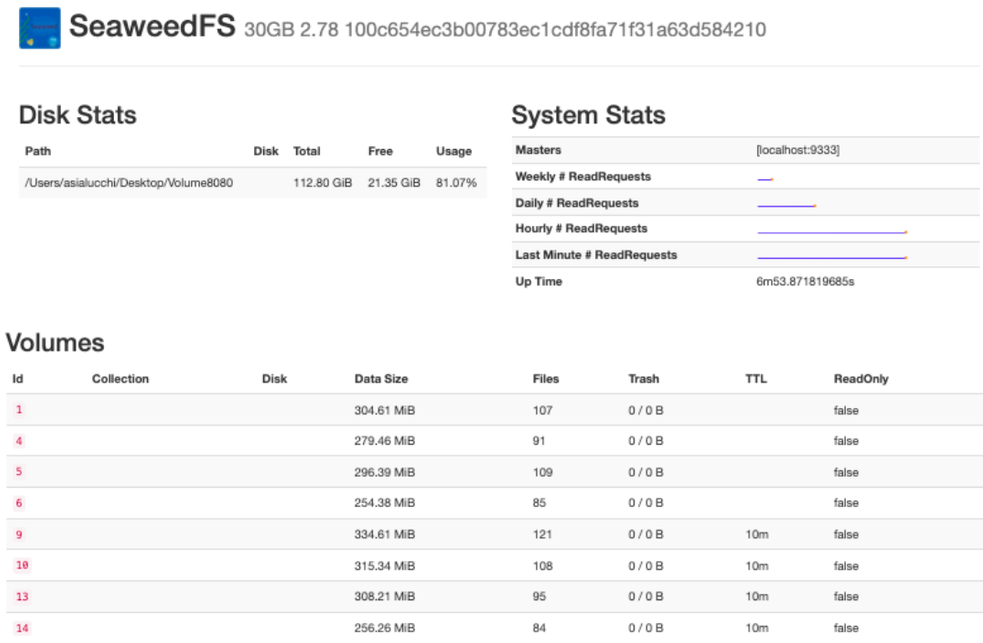


Figura 5.2: Uno dei volume server in cui sono stati cretati volume con ttl a 10m.

Una volta trascorsi i 10 minuti, i file con tale ttl vengono eliminati rimuovendo completamente i *volume*, dato che essi stessi erano associati ad un ttl=10m. Si può notare come non rimane nessuno residuo della loro presenza, infatti anche il numero di *volume* liberi riportati nella schermata master ritorna al precedente valore.

Nelle figure è riportata la situazione che si verifica dopo 10 minuti.

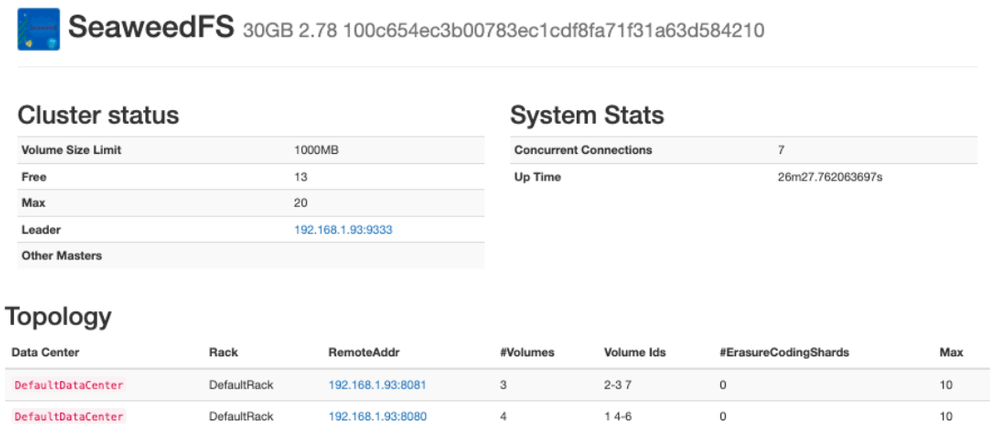


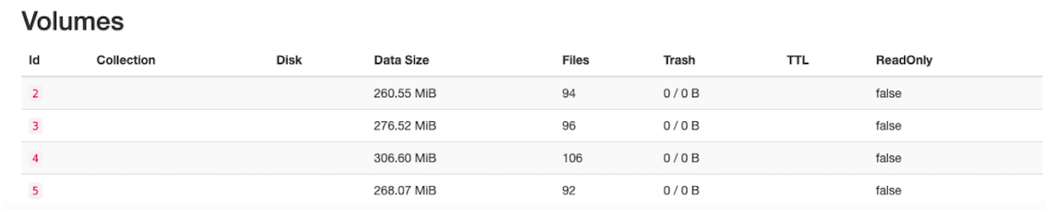
Figura 5.3: Master dopo 10 minuti.

5.2 Garbage Collector

È stato fatto un test del funzionamento del garbage collector utilizzando il master e i volume descritti nella demo e caricando i file della cartella TestDir. Attraverso le seguenti istruzioni sono solo eliminati alcuni file nel server sulla porta 8081.

```
curl -X DELETE http://10.201.102.225:8081/2,02b5cd37627c
curl -X DELETE http://10.201.102.225:8081/5,02ab4fe223f3
curl -X DELETE http://10.201.102.225:8081/4,02a5346e30fb
curl -X DELETE http://10.201.102.225:8081/2,02a3a6141b14
```

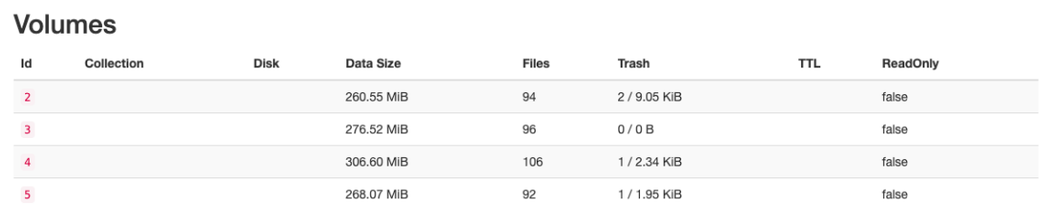
Di seguito è rappresentato come cambia la situazione nel server 8081 al momento dell'eliminazione dei file.



Volumes

Id	Collection	Disk	Data Size	Files	Trash	TTL	ReadOnly
2			260.55 MiB	94	0 / 0 B		false
3			276.52 MiB	96	0 / 0 B		false
4			306.60 MiB	106	0 / 0 B		false
5			268.07 MiB	92	0 / 0 B		false

Figura 5.4: Volume server 8081 una volta caricati i file.



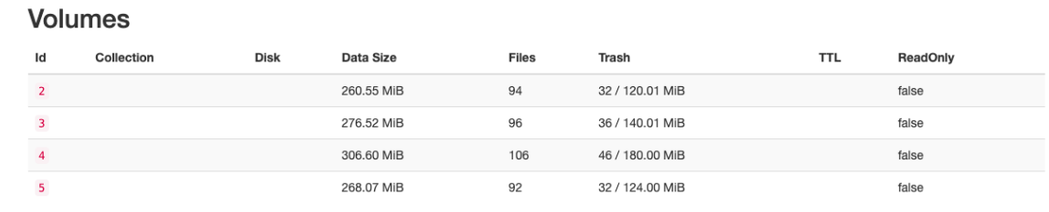
Volumes

Id	Collection	Disk	Data Size	Files	Trash	TTL	ReadOnly
2			260.55 MiB	94	2 / 9.05 KiB		false
3			276.52 MiB	96	0 / 0 B		false
4			306.60 MiB	106	1 / 2.34 KiB		false
5			268.07 MiB	92	1 / 1.95 KiB		false

Figura 5.5: Volume server 8081 subito dopo l'eliminazione dei file.

Come si può notare, nella colonna *Trash* sono indicati la quantità di file che l'utente ha chiesto di eliminare e lo spazio totale da essi occupato. Tuttavia il numero e la dimensione di ogni volume ancora non è cambiata, infatti l'eliminazione vera e propria avverrà solo nel momento di attivazione del Gargage Collector. Essendo i file eliminati ancora in realtà registrati nel sistema, potrebbe essere utile avere un modo per recuperarli nel breve termine.

Nella documentazione è indicato che il Garbage Collector dovrebbe procedere all'eliminazione circa ogni 15, ma dopo più di 20 minuti ancora non è stata eseguita. Ho pensato che questo potesse essere dovuto alla dimensione molto piccola dei file eliminati, quindi ho provato a cancellare anche un file di più grandi dimensioni (circa 1GB), che era stato spezzettato in più *volume* come descritto nel paragrafo 2.4, ritrovandomi nella seguente situazione.

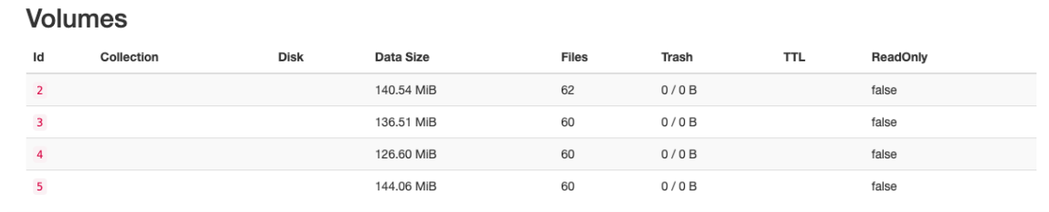


Volumes

Id	Collection	Disk	Data Size	Files	Trash	TTL	ReadOnly
2			260.55 MiB	94	32 / 120.01 MiB		false
3			276.52 MiB	96	36 / 140.01 MiB		false
4			306.60 MiB	106	46 / 180.00 MiB		false
5			268.07 MiB	92	32 / 124.00 MiB		false

Figura 5.6: Volume server 8081 dopo l'eliminazione di un ulteriore file di grandi dimensioni.

In questo caso, una volta trascorsi i 15 minuti, il Garbabe Collector ha effettivamente liberato da memoria dai file eliminati, diminuendo la dimensione e il numero di file di ciascun *volume*.



Volumes

Id	Collection	Disk	Data Size	Files	Trash	TTL	ReadOnly
2			140.54 MiB	62	0 / 0 B		false
3			136.51 MiB	60	0 / 0 B		false
4			126.60 MiB	60	0 / 0 B		false
5			144.06 MiB	60	0 / 0 B		false

Figura 5.7: Volume server 8081 dopo l'attivazione del Garbage Collector.

5.3 Volume Replication

In questo caso si inizializza diversamente il master per inserire una replicazione, ad esempio se ne usa una di tipo 010 aggiungendo il parametro `-defaultReplication` all'istruzione per inizializzare il master e fornendogli alcuni *volume* server posizionati in rack diversi, specificati con il parametro `-rack`(`setup_volume_replication.sh`).

```
./weed master -mdir="./Master" -volumeSizeLimitMB=1000
    -defaultReplication=010

./weed volume -mserver="localhost:9333" -dir="./Volume8080"
    -max=10 -port=8080 -rack=rack1
./weed volume -mserver="localhost:9333" -dir="./Volume8081"
    -max=10 -port=8081 -rack=rack1
./weed volume -mserver="localhost:9333" -dir="./Volume8082"
    -max=10 -port=8082 -rack=rack2
./weed volume -mserver="localhost:9333" -dir="./Volume8083"
    -max=10 -port=8083 -rack=rack1 -dataCenter=myDT
./weed volume -mserver="localhost:9333" -dir="./Volume8084"
    -max=10 -port=8084 -rack=rack4
```

In questo caso ci sono quattro volume server nel Data Center di default, 2 nel rack1, 1 nel rack2 e 2 nel rack4, poi c'è un ulteriore volume server in un Data Center separato con un suo rack1. La situazione nel master è la seguente.

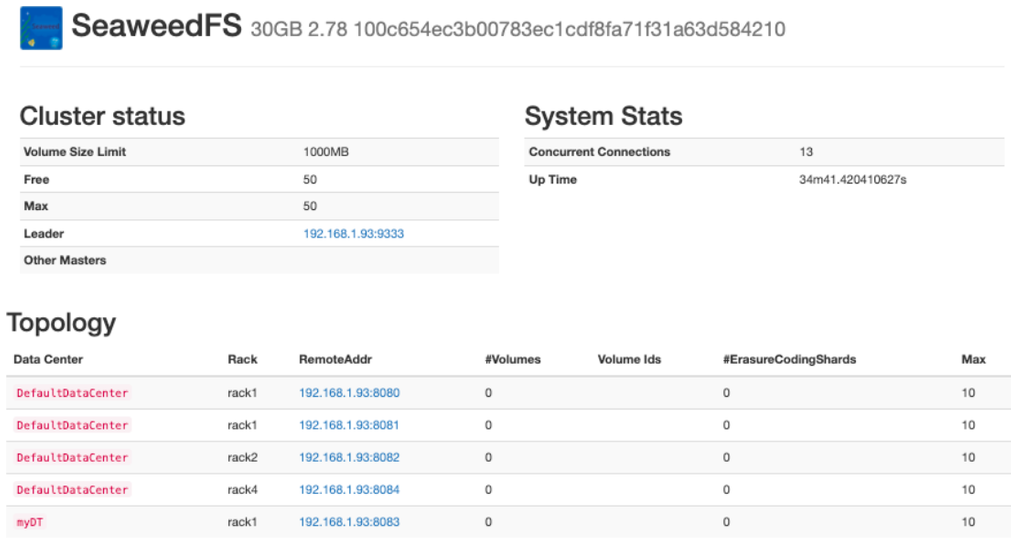


Figura 5.8: Situazione iniziale nel master dopo l'esecuzione dei comandi.

La replicazione 010 impone che ogni *volume* sia presente due volte, con il medesimo id, in rack diversi. Quando si caricano i file e ad ognuno viene assegnato un FID, si riferisce ad entrambi *volume* e quindi il file può essere fornito da uno qualsiasi dei due al momento di una richiesta in lettura. A questo punto viene caricata sul file system la solita cartella `./TestDir` e sul master si evidenzia la seguente situazione.

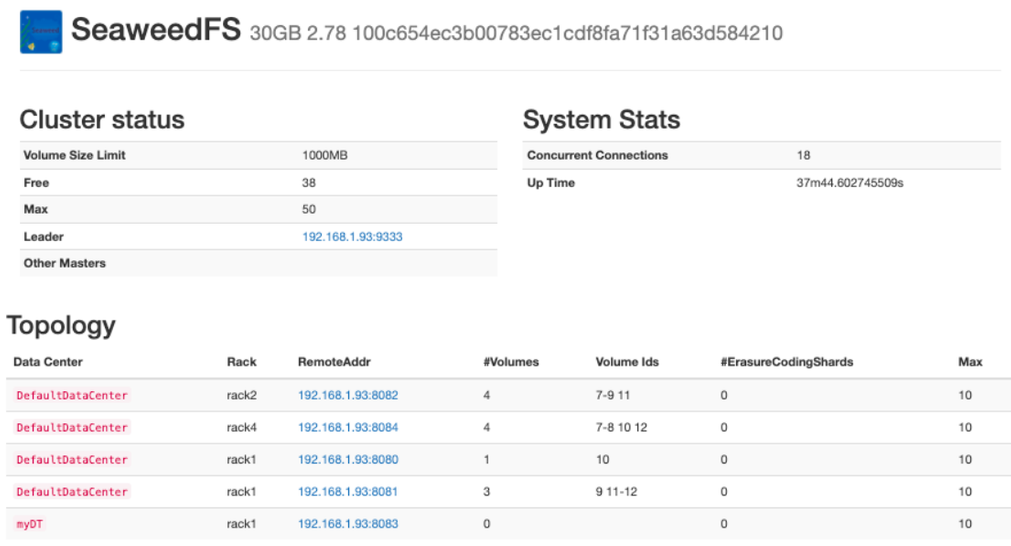


Figura 5.9: Master dopo il caricamento dei file di test.

I file sono stati caricati solamente sui volume server alle porte 8080, 8081, 8082 e 8084, ovvero nel Data Center di default, in quanto per essere caricati su un Data

Center separato bisogna specificare il parametro `-dataCenter` al momento dell'upload. Dato che per caricare i dati il sistema ha usato 6 *volume*, con id da 7 a 12, ed ognuno è stato replicato due volte come richiesto dall'infrastruttura del master, il totale di *volume* occupati è 12 e quelli lasciati liberi 50-12=38, come riportato nella UI. Si può notare che ogni volume è stato effettivamente caricato in due rack separati.

```
volume 7  => rack2 + rack4
volume 8  => rack2 + rack4
volume 9  => rack2 + rack1(8081)
volume 10 => rack4 + rack1(8080)
volume 11 => rack2 + rack1(8081)
volume 12 => rack4 + rack1(8081)
```

Solo il master sa quali sono i *volume server* che contengono i vari *volume*, all'interno di un *volume server* non c'è nessun riferimento al fatto che esso abbia un "gemello" nè alla sua posizione.

Facendo un richiesta in lettura, ad esempio al file con FID `12,0569613d5e4e`, esso è reperibile sia all'indirizzo `192.168.1.93:8084/12,0569613d5e4e` che all'indirizzo `192.168.1.93:8081/12,0569613d5e4e`. Se si tenta poi di reperire il file usando l'indirizzo `192.168.1.93:9333/12,0569613d5e4e`, ovvero attraverso il master server, si viene re-indirizzati verso il *volume server* sulla porta 8084. Se invece si cerca solamente attraverso il master server un qualsiasi altro FID che si trova sul *volume* 12, si viene re-indirizzati sulla porta 8081. Probabilmente il server sulla porta 8081 è stato registrato come "principale" per il *volume* 12, però nel momento in cui si accede in maniera diretta a un certo FID specificandone il *volume server* (come fatto nell'esempio), questo metadato viene registrato e il reindirizzamento viene fatto verso questo *volume server* piuttosto che verso quello di default.

In questa situazione, se uno qualsiasi dei rack (o parte di esso, nel caso di rack1) venissime a meno, tutti i file resterebbero ugualmente reperibili perchè ne esiste almeno una copia. Ad esempio, supponiamo che fallisca rack4, la situazione sarebbe la seguente.

```
volume 7  => rack2
volume 8  => rack2
volume 9  => rack2 + rack1(8081)
volume 10 => rack1(8080)
volume 11 => rack2 + rack1(8081)
volume 12 => rack1(8081)
```

Come si può notare tutti i *volume* restano reperibili. Se rack4 si è semplicemente disconnesso senza perdere i dati dei *volume* in locale è facilmente ripristinabile, una volta che la comunicazione riprende si ritorna alla situazione precedente. Se fallisce un ulteriore rack prima che rack4 venga ripristinato, alcuni file non sono più reperibili, ad esempio se viene a mancare rack2 tutti i dati caricati sui *volume* 7 e 8 non sono disponibili. La situazione dal punto di vista del master è la seguente.

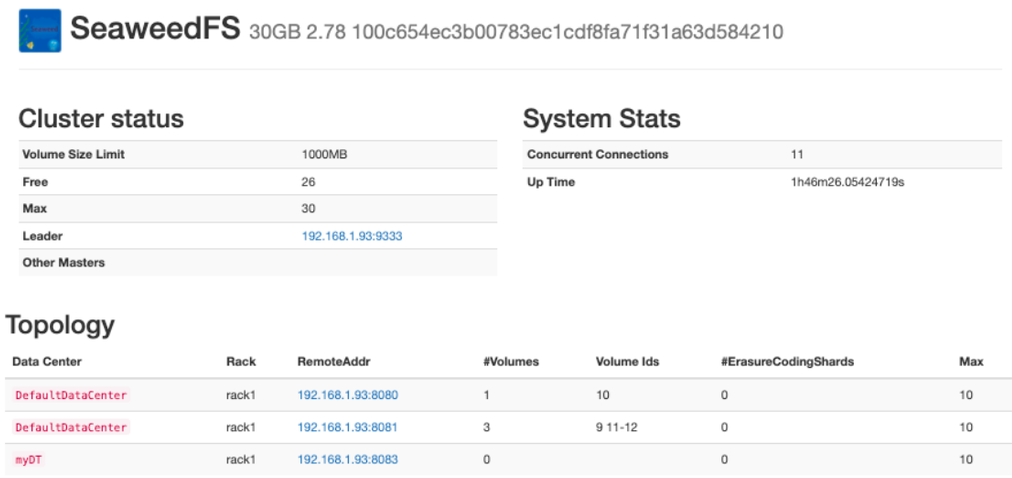


Figura 5.10: Master dopo la failure di alcuni rack.

5.4 Master Failure

Se si utilizza un unico master, nel momento in cui esso viene a mancare tutti i *volume server* se ne rendono conto nel giro di pochi secondi grazie al meccanismo di **hearbeat**. Essi restano in attesa del ripristino continuando a inviare messaggi di heartbeat ogni 5 secondi all'indirizzo del master che gli è stato specificato al momento della creazione. Di seguito è riportato un estratto del log prodotto dai *volume server*.

```
I1118 10:34:54 57333 volume_grpc_client_to_master.go:69]
    heartbeat error [...]
I1118 10:34:59 57333 volume_grpc_client_to_master.go:104]
    SendHeartbeat to localhost:9333: [...]
```

Si possono utilizzare più master che mantengono registrata la medesima informazione attraverso il parametro `-peers` che indica tutti gli indirizzi dei master del file system. Se il loro numero è pari, l'operazione di avviamento del master non va a buon fine. Di seguito sono riportati i comando per l'avviamento di tre master legati fra loro (`setup_multiple_masters.sh`).

```
./weed master -ip=localhost -port=9301 -mdir="./Master" -volumeSizeLimitMB=1000
    -peers=localhost:9301,localhost:9302,localhost:9303
./weed master -ip=localhost -port=9302 -mdir="./Master1" -volumeSizeLimitMB=1000
    -peers=localhost:9301,localhost:9302,localhost:9303
./weed master -ip=localhost -port=9303 -mdir="./Master2" -volumeSizeLimitMB=1000
    -peers=localhost:9301,localhost:9302,localhost:9303
```

Il primo master che viene inizializzato è il leader (nel nostro caso, il master `localhost:9301`), i successivi notano la sua presenza e lo identificano come tale. Poi si inizializzano i *volume* server (con i medesimi comandi usati i precedenza), essi possono specificare come master uno qualsiasi dei peers presenti ma gli sarà immediatamente comunicato il nome del leader a cui invieranno effettivamente gli heartbeat.

```
./weed volume -mserver="localhost:9301" -dir="./Volume8081" -max=10
    -port=8081 -rack=rack1
./weed volume -mserver="localhost:9302" -dir="./Volume8082" -max=10
    -port=8082 -rack=rack2
./weed volume -mserver="localhost:9303" -dir="./Volume8083" -max=10
    -port=8083 -rack=rack3
```

Anche se le comunicazioni sono tutte rivolte al leader, gli altri master sono costantemente aggiornati e consistenti con esso.
L'**upload** dei file è consentito indicando come master solo il leader e non un qualsiasi master.

Se fallisce un **qualsiasi master** che non è leader, semplicemnte non ci si potrà più rivolgere a tale indirizzo per caricare o leggere i dati ma si dovrà usare uno degli altri peer.

Se si disconnette il **leader** (nell'esempio, `localhost:9301`), il comportamento del sistema evidenziato dai log sembra essere il seguente:

- grazie al meccanismo di heartbeat, i volume server si accorgono immediatamente della mancata comunicazione ed il master con cui erano stati inizializzati si rende conto della loro disconnessione
- gli altri master cercano anch'essi di comunicare con il leader, fallendo
- si avvia una nuova elezione del leader
- una volta determinato il leader, viene comunicato a tutti i master

Non c'è un processo che comunica a tutti i *volume* server l'indirizzo del nuovo leader. I *volume* server quando si rendono conto che il leader non risponde tentano di comunicare, inviando degli heartbeat, con il leader stesso (ci potrebbe essere stato solo un errore di comunicazione temporaneo) e con il master indicatogli al momento dell'inizializzazione. Considerando l'esempio, di seguito sono riportati i tentativi di comunicazione dei vari *volume* server.

```
Volume8081 -> localhost:9301
Volume8082 -> localhost:9301 + localhost:9302
Volume8083 -> localhost:9301 + localhost:9303
```

Per i volume sulle porte 8082 e 8083 la comunicazione andrà a buon fine rispettivamente con i master sulle porte 9302 e 9303 che gli indicheranno l'identità del nuovo leader una volta terminata l'elezione. Al contrario, il volume sulla porta 8081, benchè ancora attivo non riuscirà più a comunicare con il gruppo di master a meno che l'unico master con cui comunica, l'ex leader, non torni attivo. Il nuovo leader non tenterà di comunicare con tale *volume* server per verificare se è ancora connesso ma lo giudicherà direttamente come fallito dato che non ne riceve gli heartbeat, mentre il realtà esso è ancora attivo.

5.5 Erasure Coding

Per applicare l'Erasure Coding, è sufficiente avere un file `master.toml` nella medesima cartella dell'eseguibile weed, tale file è generabile con il comando `weed scaffold -config=master`. La configurazione di default presente in tale file fa EC su i volume pieni al 95% che non sono stati utilizzati per un'ora. Per velocizzare la fase di test, sono stati modificati questi parametri in modo da fare EC su i volume pieni al 70% che non sono stati utilizzati per 5 minuti.

```
ec.encode -fullPercent=70 -quietFor=5m
```

Inoltre il parametro `sleep_minutes` è stato ridotto a 8 minuti nel tentativo di controllare se c'è la possibilità di fare Erasure Coding più frequentemente.

5.5.1 Primo test

È stato fatto un setup iniziale con due *volume* server contenenti al massimo rispettivamente 3 e 5 *volume* dalla dimensione massima di 1000MB, usando i seguenti comandi (`setup1.sh`).

```
./weed master -mdir="./Master" -volumeSizeLimitMB=1000
./weed volume -mserver="localhost:9333" -dir="./Volume8081"
    -max=3 -port=8081
./weed volume -mserver="localhost:9333" -dir="./Volume8082"
    -max=5 -port=8082
```

Successivamente sono stati fatti diversi upload (`upload1.sh`) fino ad ottenere la seguente situazione nei due *volume* server.

Volumes

Id	Collection	Disk	Data Size	Files	Trash	TTL	ReadOnly
1			657.40 MiB	300	0 / 0 B		false
4			702.42 MiB	309	0 / 0 B		false
7			548.89 MiB	280	0 / 0 B		false

Figura 5.11: Volume sulla porta 8081 subito dopo il caricamento dei file.

Volumes

Id	Collection	Disk	Data Size	Files	Trash	TTL	ReadOnly
2			685.90 MiB	303	0 / 0 B		false
3			597.27 MiB	277	0 / 0 B		false
5			670.21 MiB	313	0 / 0 B		false
6			721.70 MiB	308	0 / 0 B		false

Figura 5.12: Volume sulla porta 8082 subito dopo il caricamento dei file.

Come si può notare, i *volume* con id 4 e 6 superano la soglia del 70% rispetto alla dimensione massima, quindi ci si aspetta che nel giro di poco tempo sia eseguita la procedura di EC su di essi. Nelle specifico ciò dovrebbe avvenire dopo al massimo 13 minuti: 5 perchè lo stato del volume passi da *hot* a *cold* ed al massimo 8 per eseguire un controllo. Dato che ciò non avviene, si fanno due possibili ipotesi:

- C'è un minimo per il tempo specificato nel parametro `-quietFor`, tale minimo in ogni caso non potrà essere superiore a quello di default di un'ora. Quindi se questo dovesse essere il problema, al massimo dopo 1h e 8 minuti dal caricamento, l'erasure coding dovrebbe essere eseguito.

- Il superamento rispetto al percentuale specificata del 70% è troppo basso, solo di pochi MB (circa 2 per il *volume* 4 e circa 22 per il *volume* 6. Dato che il sistema di default ha a che fare con *volume* di 30GB, forse questo scostamento è considerato troppo piccolo per far scattare la procedura di EC.

In realtà, analizzando i log prodotti dal master, ci si rende conto che effettivamente esso ogni 8 minuti individua i due *volume* con una percentuale di spazio occupato troppo alta ma non riesce a fare l'EC per mancanza di spazio disponibile. I log indicano che non ci sono abbastanza EC shards slot, perchè ce ne sono solo 10 mentre come indicato nel paragrafo 2.5, ne servono 14. Considerando che ogni shard è di 100MB, servono 1400MB di spazio libero, ma ogni volume è al massimo di 1000MB e c'è 1 solo volume disponibile rispetto agli 8 totali (3 sul primo *volume* server e 5 sul secondo).

5.5.2 Secondo test

L'obiettivo è quello di ricreare una situazione in cui il numero dei *volume* è inferiore al massimo ed in cui quelli presenti raggiungono una percentuale di riempimento adeguata all'esecuzione della procedura. Una possibilità sarebbe stata quella di diminuire di molto la percentuale di riempimento che fa attivare l'EC, ma ciò non sarebbe stato coerente con i presupposti per cui la metodologia viene applicata. Se si fosse semplicemente impostato un numero massimo di *volume* molto alto probabilmente il sistema avrebbe distribuito file su tutti tali volume raggiungendo prima un numero vicino al loro massimo piuttosto che un percentuale di riempimento sufficiente ad eseguire l'EC.

Per essere sicuri di ricreare la situazione desiderata si è quindi pensato di inserire file con diverso ttl, che quindi occupassero certi *volume* solo per poco tempo. In questo modo il sistema può avere solo una piccola parte di *volume* senza ttl ed è costretto a riempirli maggiormente nel corso dei vari caricamenti, una volta che il ttl scade ci troviamo proprio in una situazione in cui abbiamo sia dei *volume* liberi che dei *volume* sufficientemente pieni da attivare l'EC.

Inoltre il tempo di "raffreddamento" minmo dei dati (parametro `-quietFor`) è stato aumentato a 10 minuti per poter avere tempo di notare le differenze prima e dopo l'EC e sono stati inizializzati 4 *volume* server con al massimo 5 *volume* da 1000MB ciascuno (`setup2.sh`).

```
./weed master -mdir="./Master" -volumeSizeLimitMB=1000
./weed volume -mserver="localhost:9333" -dir="./Volume8080"
    -max=5 -port=8080
```

```
./weed volume -mserver="localhost:9333" -dir="./Volume8081"
    -max=5 -port=8081
./weed volume -mserver="localhost:9333" -dir="./Volume8082"
    -max=5 -port=8082
./weed volume -mserver="localhost:9333" -dir="./Volume8083"
    -max=5 -port=8083
```

Dopo il caricamento di varie tipologie di file (`uplaod2.sh`) con un diversi ttl (fra 2 e 5 minuti) tutti i *volume* sono occupati e la situazione del file system è la seguente.

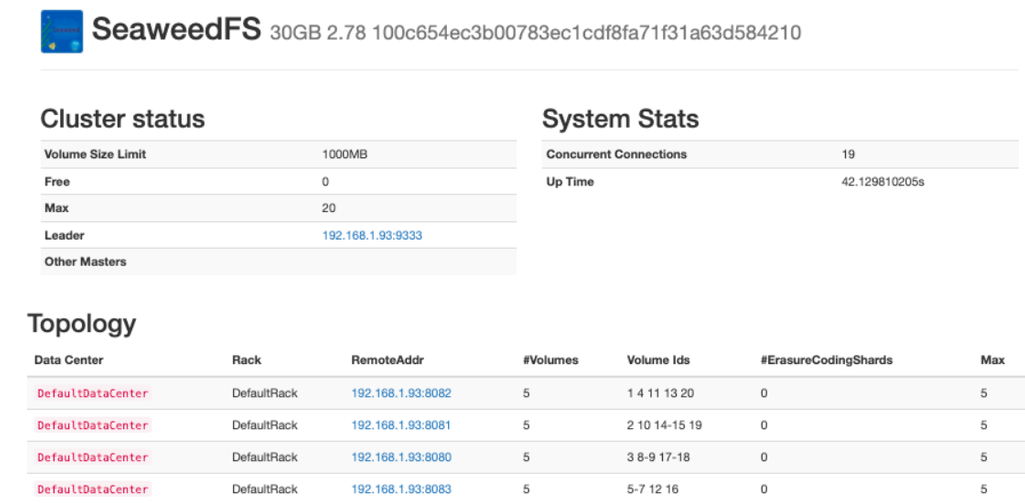


Figura 5.13: Master subito dopo il caricamento dei file.

Attendendo 5 minuti, i volume con un ttl minore o uguale vengono eliminati a tale tempo, dando luogo alla seguente configurazione nel master.

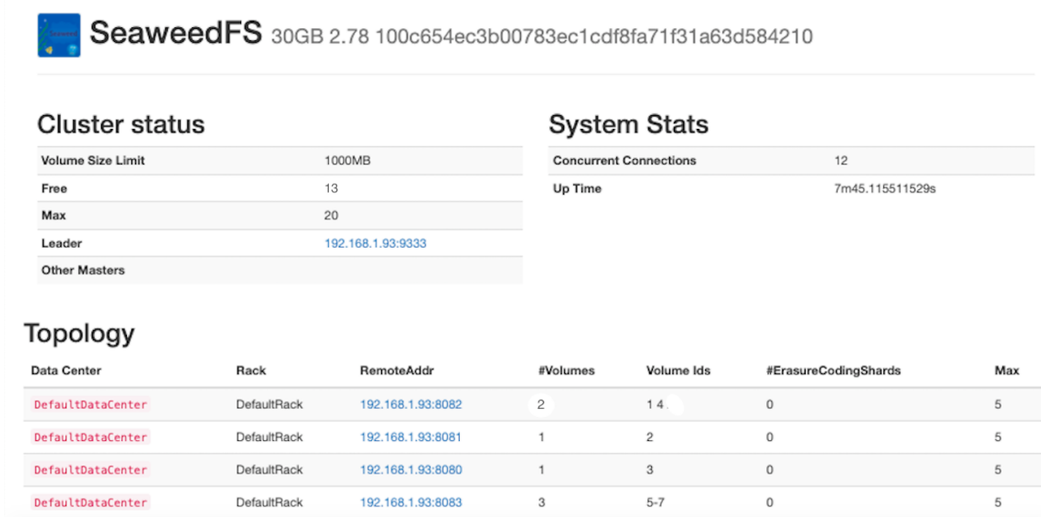


Figura 5.14: Master dopo la scadenza del ttl di tutti i *volume* inizializzati.

Ora ci sono ben 13 *volume* liberi che possono contentere gli shard prodotti dall'erasure coding. Lo stato dei 4 *volume* server è il seguente.

```
-----------------------------------------
Volume Server port      Id       Size
-----------------------------------------
 8080                    3       679.18MB
-----------------------------------------
 8081                    2       697.41MB
-----------------------------------------
 8082                    1       701.84MB
 8082                    4       655.14MB
-----------------------------------------
 8083                    5       744.47MB
 8083                    6       550.87MB
 8083                    7       656.50MB
-----------------------------------------
```

Come si può notare, solo i volume numero **1** e **5** superano la soglia del 70% della capienza massima, perciò l'EC verrà eseguito solo su di essi. Restiamo in attesa del controllo che avverrà dopo 8 minuti, in questo caso non si avvierà l'EC perchè certamente nessuno dei *volume* è *cold* da 10 minuti, dato che è stato creato più recentemente.

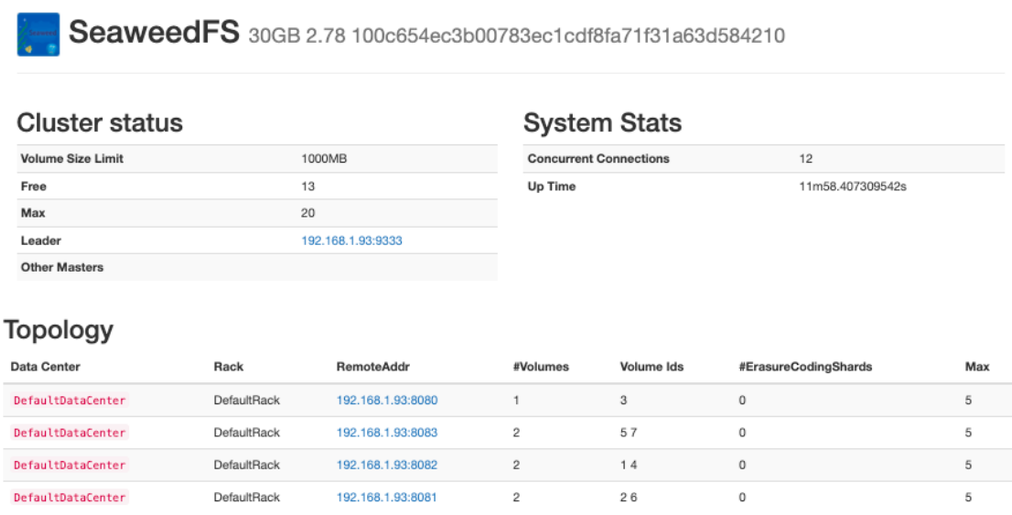


Figura 5.15: Master dopo il primo controllo di EC (8+ minuti).

.

Come previsto non viene eseguito l'EC, però grazie alla presenza dello script nel file `master.toml`, che comprende l'esecuzione di `ec.balance`, viene fatto un bilanciamento dei *volume* fra i vari server, infatti il *volume* numero 6 viene trasferito dal server sulla porta 8083, che aveva tre *volume*, al server sulla porta 8081, che ne aveva solamente uno. Attendendo ulteriori 8 minuti, viene realizzata la divisione in shard.

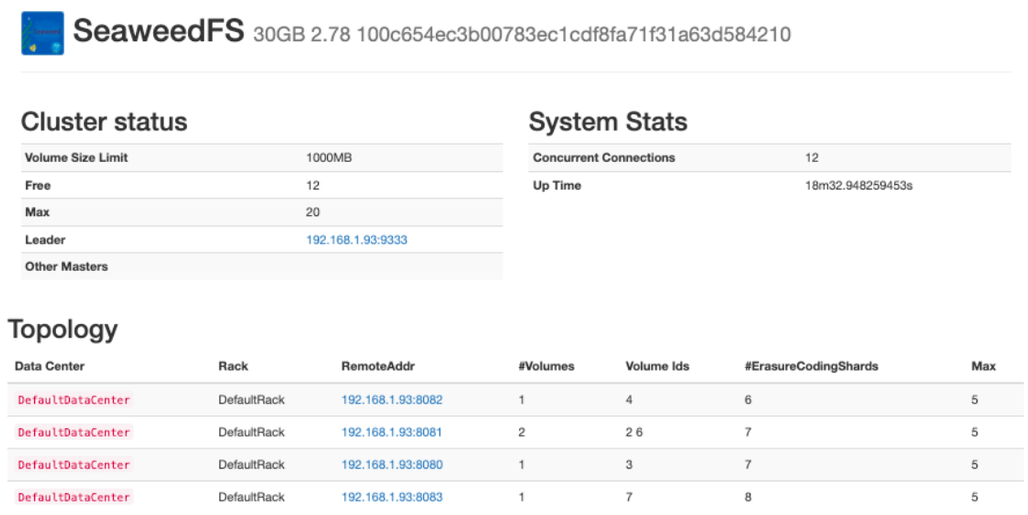


Figura 5.16: Master dopo l'EC (16+ minuti).

Come si può notare, i *volume* **1** e **5** non sono più presenti ma in compenso ci sono un totale di 28 shards, infatti ognuno di tali *volume* è stato spezzato in 14 shard di dimensione 100MB. Di seguito è riportato lo stato di ciascuno dei 4 *volume* server dopo l'EC.

```
SHARDS
-------------------------------------------------------------------------------
Server            Volume Id     Shard Size      Shards        Created at
-------------------------------------------------------------------------------
 8080               1             71.00 MB       [1 5 9 13]    24/11/21(15.03)
 8080               5             75.00 MB       [3 7 11]      24/11/21(15.03)
-------------------------------------------------------------------------------
 8081               1             71.00 MB       [3 7 11]      24/11/21(15.03)
 8081               5             75.00 MB       [0 4 8 12]    24/11/21(15.03)
-------------------------------------------------------------------------------
 8082               1             71.00 MB       [2 6 10]      24/11/21(15.03)
 8082               5             75.00 MB       [2 6 10]      24/11/21(15.03)
-------------------------------------------------------------------------------
 8083               1             71.00 MB       [0 4 8 12]    24/11/21(15.03)
 8083               5             75.00 MB       [1 5 9 13]    24/11/21(15.03)
-------------------------------------------------------------------------------

VOLUMES
-----------------------------------------
Volume Server port      Id       Size
-----------------------------------------
 8080                    3       679.18MB
-----------------------------------------
 8081                    2       697.41MB
 8081                    6       550.87MB
```

```
-----------------------------------------
 8082                    4       655.14MB
-----------------------------------------
 8083                    7       656.50MB
-----------------------------------------
```

Con tali dati si possono fare alcune considerazioni:

- **Storage**

  Ora in totale il *volume* con id 1 occupa uno spazio di 71*14Mb, ovvero 994MB. In realtà se si considera la sua dimensione iniziale di 701.84 MB, in totale gli shard sarebbero dovuti essere 982.576MB, ovvero 70.184MB ciascuno, ma probabilmente la granularità più fine che si può avere a livello di dimensioni di shard è il singolo MB perciò sono stati tutti arrotondati a 71MB. Allo stesso modo, il *volume* 5 occupa 1.05GB.

- **Distribuzione**

  Gli shard sono stati messi in un ordine specifico nel vari sever, ad esempio quelli del *volume* 1 seguono l'ordine 8083 -> 8080 -> 8082 -> 8081. Probabilmente questo meccanismo serve a rendere possibile il ripristino di tutti i file quando si eliminano degli shard perchè quelli eliminati non saranno ad esempio tutti all'inizio o tutti alla fine del *volume*, bensì resteranno in ogni dei "pezzetti" di ogni zona del *volume*.

- **Fault Tolerance**

  Uno dei punti di forza dell'EC è quello di mantenere disponibli tutti i file anche quando al massimo 4 degli shard vengono eliminati. Per sfruttare questo meccanismo bisogna disporre di almeno 3 server, caso in cui alcuni server avrebbero 4 shards ed altri 5. Nell'esempio, in cui sono presenti 4 server, tutti i server hanno meno di 5 shards quindi se uno dovesse fallire, a prescindere da quale, i file nel volume su cui è fatto EC resterebbero reperibili. Ciò rappresenta un vantaggio rispetto alla situazione precedente, ad esempio se fallisse il server sulla porta 8082 prima sarebbero andati perduti tutti i file sui *volume* 1 e 4, mentre ora vanno perduti solo quelli sul *volume* 4.

A questo punto si può fare un tentativo di lettura dei file presenti nelle shard, ad esempio inserendo nel browser l'indirizzo `192.168.1.93:9333/1,47cda1e48ef3`. Il master rimanda il riferimento ad uno dei server (ad esempio `<IP>:8080/<FID>`) ed il file viene caricato senza una latenza percepibile da un utente. Curiosamente tale lettura avviene a prescindere dalla porta del server speficata nell'indirizzo, anche se secondo le documentazione esso si trova interamente in uno o al massimo due server (nel caso si trovi a metà fra due shard).

Anche il file viene richiesto in lettura, quindi il volume non si può considerare *cold*, lo stato precedente del volume non viene ripristinato, esso resta diviso in shards. Anche facendo un numero elevato di richieste la situazione non cambia. Ciò da un lato è conveniente perchè si è più *fault tolerant*, ma dall'altro si occupa una maggiore quantità di spazio ed è più difficoltoso ricostruire i file, il che può diventare un problema quando il carico di lavoro su un gruppo di file diviso in shard diventa molto elevato.

A questo punto si può testare l'effettiva **fault tolerance** del sistema in queste condizioni, ad esempio disconnettendo il server sulla porta 8081. Lo stato dei singoli server resta il medesimo, mentre lo stato del master diventa il seguente.

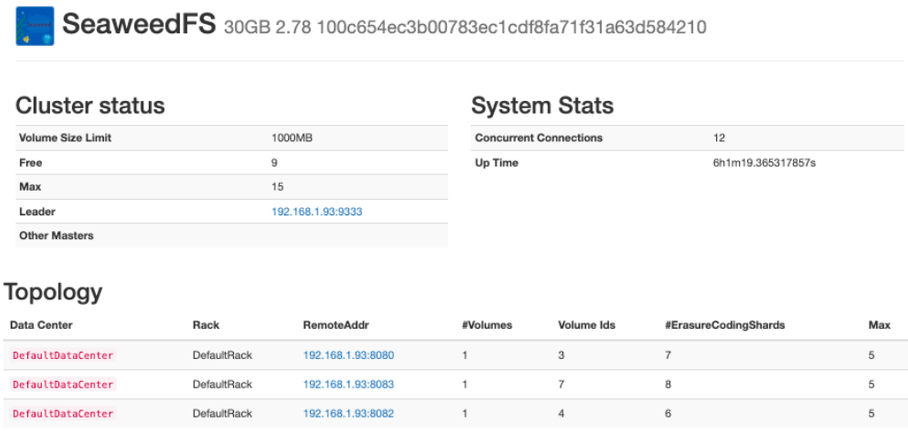


Figura 5.17: Master dopo la disconnessione del server sulla porta 8081

.

Ora gli shard totali non sono più 28, bensì 21, perchè sono andati perduti gli shard `[3 7 11]` del *volume* 1 e gli shard `[0 4 8 12]` del *volume* 5. Tuttavia essendo presenti 10 shard o più per *volume*, è possibile fare richieste in lettura a qualsiasi file presente in questi volume, tutte le prove eseguite in questo senso sono andate a buon fine. Al contrario i file dei *volume* 2 e 6, interamnete contenuti sul server 8081 dati non vi era stato fatto erasure coding, sono andati completamente perduti.

Successivamente si connette nuovamente il il server sulla porta 8081. A questo punto si verifica una situazione imprevista, come si può notare nella seguente figura che rappresenta lo stato del master.

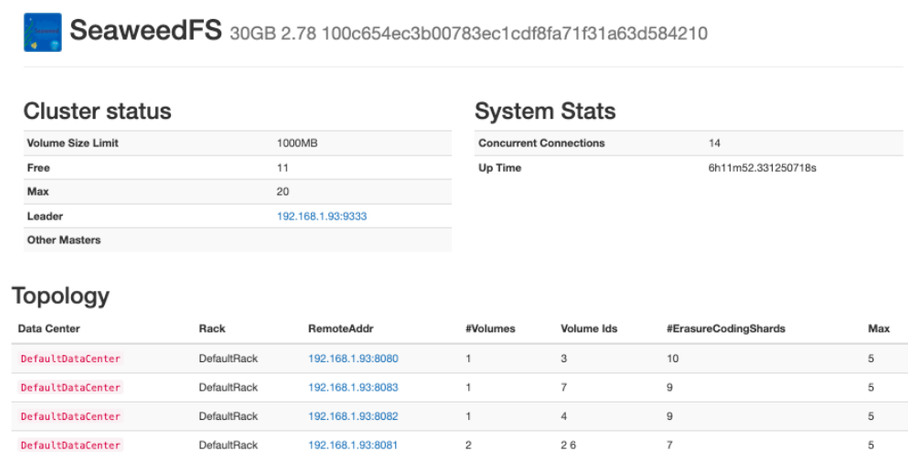


Figura 5.18: Master quando il server sulla porta 8081 ripristina la sua connessione

.

Gli shard non sono 28, come ci saremmo aspettati, ma 35, esattamente 7 in più, ne possiamo quindi dedurre ci sono 7 shard presenti in due copie. Di seguito è riportata la distribuzione degli shard nei *volume* server.

SHARDS

Server	Volume Id	Shard Size	Shards	Created at
8080	1	71.00 MB	[5 **7** 9 **11** 13]	24/11/21(15.03)
8080	5	75.00 MB	[**0** 2 3 7 11]	24/11/21(15.03)
8081	1	71.00 MB	[3 7 11]	24/11/21(15.03)
8081	5	75.00 MB	[0 4 8 12]	24/11/21(15.03)
8082	1	71.00 MB	[2 **3** 6 10]	24/11/21(15.03)
8082	5	75.00 MB	[**4** 6 **8** 10 **12**]	24/11/21(15.03)
8083	1	71.00 MB	[0 1 4 8 12]	24/11/21(15.03)
8083	5	75.00 MB	[1 5 9 13]	24/11/21(15.03)

Come si può notare alcuni shard hanno cambiato server ma soprattutto i duplicati, evidenziati in grassetto, sono tutte copie dei frammenti presenti nel server sulla porta 8081. Questo meccanismo permette di aver a disposizione i frammenti *instabili* anche sugli altri server, probabilmente ciò può essere utile nel caso in cui il totale degli shard mantenuti degli altri server sia inferiore a 10. Gli shard duplicati possono essere considerati instabili perchè mantenuti su una macchina che ha si è appena sconnessa per breve tempo, sulla quale quindi è possibile si verifichino altri problemi di comunicazione nei minuti successivi.

Al successivo controllo che avviene ogni 8 minuti gli shard duplicati vengono individuati e se ne elimina una copia di ciascuto, ripristinando per il master la situazione in figura 5.16. I server mantengono al massimo 4 shard di ogni volume, anche se con una disposizione diversa rispetto a quella precedente.

Infine si può disconnettere più di un server e controllare che i file sui volumi 1 e 5 effettivamente non siano leggibili. Ad esempio disconnettendo i file server sulle porte 8080 e 8081, restano solo 6 frammenti del *volume* 1 e 8 frammenti del *volume* 5, infatti parte delle richieste in lettura non vanno a buon fine. Tuttavia se i file richiesti casualmente si trovano su uno degli shard ancora presenti, essi sono reperiti senza particolare latenza. Chiaramente sarà molto improbabile che il file di grandi dimensioni (1GB) composto da tanti chunk sparsi fra vari *volume* e per tutti gli shard di un *volume* frammentato sia ricostruibile.

5.5.3 Terzo test

Nel terzo tipo di setup (`setup3.sh` + `upload3.sh`) la dimensione massima per *volume* è decisamente inferiore, di 200MB, quindi la situazione è leggermente differente in quanto tutti i *volume* sono suddivisi in shard.

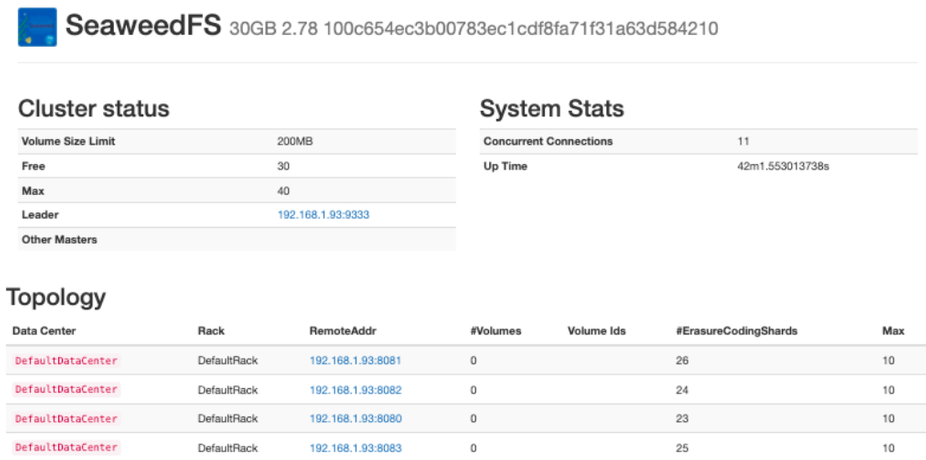


Figura 5.19: Master dopo l'Erasure coding.

In totale sono presenti 98 shard che permettono di ricomporre 7 *volume*. Viene indicato che c'è ancora lo spazio sufficiente a contenere 30 *volume*, ovvero 10 in meno rispetto al numero massimo, perchè ognuno dei 7 *volume* divisio in shard occupa la sua dimensione aumentanta del 40%, ovvero 7x1.4 = 9.8 *volume*.

Anche in questo caso ogni server non contiene più di 4 shard provenienti dallo stesso *volume* in modo da essere tollerante alla failure di un qualsiasi server.

Pur inviando una grande quantità di richieste in lettura (rendendo i dati *hot*) non si torna mai alla ricomposizione delle shard in *volume* a livello fisico.

Capitolo 6

Benchmarking

6.1 Replication

Il sistema mette a disposizione uno script per fare benchmarking che comprende la scrittura e la lettura di poco più di 1 milione di file di dimensione 1 KB. Di default si inviano le richieste al master su `localhost:9333` e per i *volume* creati si dichiara la collection "*benchmark*".

Durante la **scrittura**, si invia una richiesta di assegnamento di una chiave ad ogni file ed una richiesta di caricamento vero e proprio del file. La **lettura** è fatta randomicamente e le richieste servono sia per reperire il la posizione del file di interesse che per la lettura vera e propria.

Di seguito sono riportati alcuni dei risultati ottenuti eseguendo il benchmarking in locale testando vari tipi di replicazione (000 indica il benchmarking di default, senza replication). I log completi di ogni test si possono trovare nella directory `benchmark_logs`.

#	Replication	Action	Time	Setup
1	000	Write	249 seconds	first_demo
		Random read	83 seconds	
2	001	Write	586 seconds	first_demo
		Random read	156 seconds	
3	010	Write	524 seconds	volume_replication
		Random read	136 seconds	
4	100	Write	505 seconds	volume_replication
		Random read	141 seconds	
5	002	Write	679 seconds	big_demo
		Random read	140 seconds	

Chiaramente non si può fare eccessivo affidamento su queste misure:

– in primo luogo perchè sono state eseguente in locale e non in un sistema realmente distribuito;

– inoltre la cpu della macchina probabilmente durante l'esecuzione di alcuni dei test fosse stessa facendo altri calcoli ad esempio dovuti ad aggiornamenti o ad altri programmi eseguiti sulla macchina nello stesso momento.

Considerando #**1** e #**2**, si possono fare le seguenti osservazioni:

- per la scrittura si suppone che #2 impieghi circa il doppio di #1 in quanto deve scrivere il doppio delle copie, ciò effettivamente avviene anche se con un leggero ritardo di #2;

- per la lettura si suppone che impieghino entrambi circa lo stesso tempo mentre in realtà #2 ci mette il doppio #1 anche in questo caso.

Come prevedibile, le tempistiche in #**2**, #**3** e #**4** sono simili, infatti la quantità di file da caricare è la stessa, anche se in tutti e tre i casi il tempo in lettura è stranamento molto più grande del caso senza replicazione. Ci potrebbe essere una minima latenza in più dovuta al fatto che per ogni file sono reperiti due indirizzi invece di uno, ma mi sembra improbabile che possa giustificare una ritardo del genere, credo sia più plausibile dire che il tempo in lettura di #1 è stato casualmente molto breve.

Nel caso #**5**, ci sono tre copie di ogni file, perciò si suppone che il tempo il scrittura triplichi rispetto ad #1, in questo caso il tempo effettivo è leggermente inferiore ma comunque non eccessivamente scostato rispetto alle aspettative. Il tempo di lettura rimane in linea con i tre test intermedi e questo rafforza l'ipotesi formulata nel paragrafo precedente, ovvero che il tempo in lettura di #1 sia l'eccezione.

6.2 Read O(1)

Per avvalorare l'affermazione fatta dallo sviluppatore che ogni lettura sia O(1), ovvero abbia velocità costante, si dovrebbero confrontare le velocità in lettura per file di piccole o grandi dimensioni. Considerando che anche nei test di scrittura di più di 1M di file ci sono dei considerevoli scostamenti dovuti probabilmente allo scheduling interno della cpu, andare a confrontare operazioni che impiegano millesimi di secondo non produrrebbe risultati utili a dimostrare alcuna evidenza, perciò l'idea di verificare che la lettura sia effettivamente O(1) è stata abbandonata.

Capitolo 7

Conclusioni

7.1 Aspetti positivi

I vantaggi più evidenti di SeaweedFS sono:

- **Alta disponibilità dei dati**
  - Non c'è un Single Point Of Failure, essendo organizzato in tanti *volume* e dando la possibilità di avere più master.
  - C'è supporto a varie tecniche che permettono di dare priorità alla disponibilità: la replicazione asincrona active-active, l'erasure coding, il checksum dei file, la replicazione in diversi volume, rack o data center, il backup.
  - Location Transparency, ovvero i *volume* possono essere spostati in vari server o sul cloud senza compromettere la disponibilità dei file che contengono, infatti sono individuati in base ad un identificativo, non la posizione (l'indirizzo IP).
- **Ottimizzazione delle performance e dello spazio**
  - Ottimizzato per la gestione di tanti file di piccole dimensioni, l'overhead per ciascuno è di 40 bytes (contro ad esempio i 536 bytes dei Linux inode).
  - La lettura avviene sempre in O(1) direttamente sui *volume* server.
  - Linearmente scalabile.
  - Non ci sono bottleneck, al contrario di altri sistemi come HDFS.
  - In base ai requisiti si possono mettere i dati in tipologie di dischi diversi (ex. SSD, HDD).
  - Varie tipologie di file sono compresse per limitare lo spazio che occupano, inoltre periodicamente i *volume* server vengono ulteriormente compattati.
  - Non c'è limite alla dimensione delle directory o i file che possono essere mantenuti.
- **Operazioni semplici**
  - Per aumentare la capacità, è sufficiente aggiungere *volume* server.
  - Migrare i dati di un *volume* server è una procedura immediata, basta spostare i pochi grandi file salvati in locale.

- I metadati dei Filer sono salvati in formati standard ed immagazzinati in sistemi comuni, ad esempio MySql o MondoDB, ce ne sono tante tipologie disponibili ed è semplice migrare da un sistema all'altro.
- L'utilizzo in generale è intuitivo e il setup iniziale è semplice e veloce.

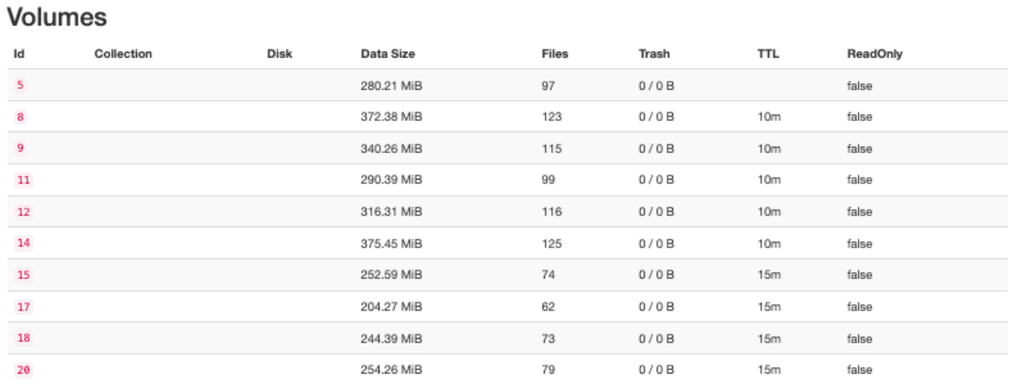
7.2 Aspetti negativi

Alcune problematiche o difetti di SeaweedFS che ho individuato nella mia analisi sono:

- Non è possibile fare il listing di una directory o di tutti i file presenti all'interno di uno specifico *volume*.
- Nell'interfaccia grafica dedicata al *volume* server, non è indicato il momento di creazione di ogni *volume* ed il TTL non è modificato periodicamente per indicare il tempo rimanente, perciò da questa schermata è impossibile dedurre fra quanto i *volume* verrano eliminati.
- Nell'interfaccia grafica dedicata al master, non è indicato il tipo di replicazione di default scelta.
- Per dischi grandi, che quando si è iniziato lo sviluppo non esistevano, è necessario usare dei FID da 17 bytes per poter utilizzare tutto lo spazio reso disponibile dal disco, mentre in precedenza si usavano FID da 16 bytes. La nuova versione non è compatibile con la vecchia, quindi per modificarla è necessario registrare nuovamente tutti i file. Inoltre l'aumento di un unico byte è probabilmente pensato per ridurre al minimo lo spazio occupato dai FID, ma nel caso non sia sufficiente in futuro è possibile che si faccia un'ulteriore versione incompatibile (ex. da 18 bytes) che implicherà un nuovo caricamento. Infine la scelta di immagazzinare 17 bytes non è consona con la struttura comune della memoria che lavora sempre a multipli di 8.
- In alcuni casi avere come unità di misure minima il *volume* può essere scomodo, ad esempio nel caso in cui si vogliano avere svariati TTL differenti per le tipologie di file ma non sarebbe necessario allocare lo spazio per un intero *volume* data la quantità di file con il TTL specificato. Anche se essendo un File System distribuito si presuppone che abbia poche tipologie[1] di file piuttosto che un numero elevato, si potrebbe dare la possibilità di scegliere granularità più piccole rispetto al *volume* in casi particolari o ancor meglio gestire in automatico separatamente i file con caratteristiche "degeneri" che occupano in realtà pochissimo spazio ma provocano l'allocazione di una grande quantità. Questa problematica si potrebbe risolvere scegliendo una dimensione massima per ogni *volume* che non vada ad allocare spazio inutile, ma poi sorgerebbe il problema di avere tantissimi *volume* di piccole dimensioni, simili a quelle dei file stessi, a questo punto lo scopo stesso di avere dei *volume* verrebbe a mancare.

[1] per tipologia non si intende il tipo di file ma tutte le caratteristiche che combinate fra loro portano a creare un *volume* specifico per file di una certa tipologia, ad esempio il ttl, la collection o il tipo di disco

- Elaborando ulteriormente il punto precedente e tenendo in considerazione i test eseguiti nei capitoli 4 e 5, si può ragionare sulla scelta di creare molti *volume* riempiti solo in piccola percentuale (nell'esempio, circa il 20%) piuttosto che la quantità minima di *volume*. Nel momento in cui si caricano file con qualche caratteristica diversa (ex.ttl diverso) e per ognuno di essi vengono inizializzati svariati volume riempiti sono in parte, si può arrivare ad una situazione in cui anche se c'è ancora molto spazio disponibile, più della metà, ma ci sono tanti *volume* semi-vuoti e quindi non è permesso caricare ulteriori file con caratteristiche diverse. Suppongo che la scelta sia dettata da ragioni di performance e dal fatto che ci si aspetta che vengano inseriti ulteriori file dello stesso tipo, ma ci sono anche degli aspetti negativi.



Volumes

Id	Collection	Disk	Data Size	Files	Trash	TTL	ReadOnly
5			280.21 MiB	97	0 / 0 B		false
8			372.38 MiB	123	0 / 0 B	10m	false
9			340.26 MiB	115	0 / 0 B	10m	false
11			290.39 MiB	99	0 / 0 B	10m	false
12			316.31 MiB	116	0 / 0 B	10m	false
14			375.45 MiB	125	0 / 0 B	10m	false
15			252.59 MiB	74	0 / 0 B	15m	false
17			204.27 MiB	62	0 / 0 B	15m	false
18			244.39 MiB	73	0 / 0 B	15m	false
20			254.26 MiB	79	0 / 0 B	15m	false

Figura 7.1: Esempio di presenza di spazio libero ma non utilizzabile.

In figura è riportato un esempio di questa situazione. In questo caso il *volume server* inizializzato può contenere al massimo 10 volumi, ognuno con dimensione masima di 1GB, per un totale di 10GB. Al momento del caricamento di dati con ttl 10m o 15m, però, essi sono stati suddivisi in più *volume* del necessario. In questo momento lo spazio occupato nel *volume server* è di circa 3GB, ovvero il 30% di quello disponibile. Se si cercano di caricare ulteriori file con le stesse caratteristiche (senza ttl, con ttl 10m o 15m, collection e disk vuoti) sarà possibile, mentre non si potranno caricare file con ttl=1h perchè non è possibile allocare *volume* per contenerli.

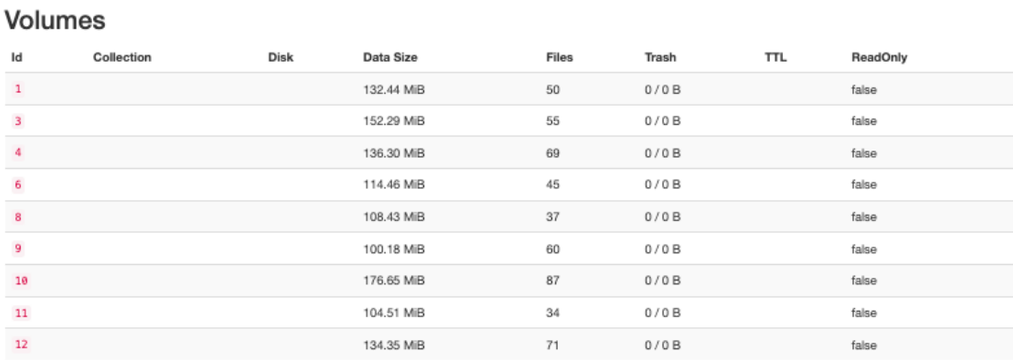
Questo aspetto non è necessariamente un difetto ma va preso in considerazione prima di decidere di utilizzare il sistema, in particolare dal momento che di default il numero massimo di *volume* è 8. Bisogna tenere conto di questo fatto se si vogliono utilizzare tante tipologie di file diversi, perchè in questo caso è necessario aumentare il `-max` a un numero molto più alto.

- A volte eliminando definitivamente i file che i volume server creano in locale nelle cartelle specificate (negli esempi:*./Volume8080*, *./Volume8081*, etc.) i dati restano ugualmente reperibili, sono tracciati dal master, i volume ad essi legati sono presenti nella UI ma soprattutto occupano spazio. Il problema rimane anche fermando e facendo ripartire il *volume* service. Non sono riuscita a darmi una spiegazione a questo comportamento nè a trovare una serie di comandi che possano riprodurlo.

- Viene indicato di usare un numero di master dispari in modo da poterne eleggere uno con algoritmo di consenso senza situazioni di parità, ma nel momento in cui uno dei master viene a meno il loro numero diventa pari e bisogna eleggere un leader in questa situazione, quindi l'algoritmo usato dovrebbe in ogni caso prenderla in considerazione.

- Sarebbe più comodo se fosse consentito l'upload dei file indicando uno qualsiasi dei master, non unicamente indicando il leader.

- Usando una dimensione massima per *volume* bassa (parametro `-volumeSizeLimitMB`), tale dimensione non è rispettata. Nel seguente esempio era stato indicato un massimo di 100MB per *volume*, ma durante l'upload sono stato caricati i file nei *volume* fino a raggiungere 152MB.



Volumes

Id	Collection	Disk	Data Size	Files	Trash	TTL	ReadOnly
1			132.44 MiB	50	0 / 0 B		false
3			152.29 MiB	55	0 / 0 B		false
4			136.30 MiB	69	0 / 0 B		false
6			114.46 MiB	45	0 / 0 B		false
8			108.43 MiB	37	0 / 0 B		false
9			100.18 MiB	60	0 / 0 B		false
10			176.65 MiB	87	0 / 0 B		false
11			104.51 MiB	34	0 / 0 B		false
12			134.35 MiB	71	0 / 0 B		false

Figura 7.2: Esempio di superamento dei limiti di dimensione imposti.

- Una volta che un volume è stato suddivviso in shard tramite la procedura di Erasure Coding, non tornerà mai alla sua forma originale anche inviandogli molte richieste in lettura e quindi trasformando i suoi dati in informazioni *hot*, non più *cold*.

- Ciò che è indicato nella documentazione non corrisponde sempre a ciò che accade nel sistema. Non è chiaro se ciò sia dovuto a bug o a mancati aggiornamenti della documentazione. Alcuni limiti imponibili tramite parametri non vengono sempre rispettati ed alcune API non funzionano come specificato, ad esempio il parametro `cm` che se indicato come `false` dovrebbe mostrare il json del Manifest se il file è diviso in chunk non sempre lo fa.

- Come indicato alla fine della sezione 5.4 è possibile che dopo l'eliminazione del leader uno o più *volume* server siano attivi ma non riescano ad essere "visti" dal gruppo di master che li considerano disconnessi. Ciò è dovuto al fatto che essi attendono un heartbeat dal *volume* server, che però non può madarglielo perchè non ne conosce gli indirizzi in quanto l'unico riferimento che aveva era il leader venuto a mancare. Sarebbe utile se dopo un'elezione fossero i master stessi a comunicare a tutti i *volume* server, di cui conoscono gli indirizzi, l'identità del nuovo leader, in modo da evitare questa situazione.

Di tutti gli lati negativi annotati questa penso che sia il più interessante nell'ambito del corso in quanto riguarda essenzialmente la **fault tolerance**, uno degli aspetti più critici di un sistema distribuito.

7.3 Opinione personale

In conclusione ritengo che la tecnologia studiata sia molto interessante perchè permette di costruire un'infrastruttura distruibuita in pochi semplici, passaggi. L'accesso ai file è veloce, c'è la possibilità parametrizzare il file systema in vari modi (ex. con la replicazione) e sfruttare a piacimento i meccanismi avanzati messi a dispozione. Le interfacce grafiche che permettono sia di avere una visione d'insieme del file system che di monitorare lo stato di tutti i server sono molto comode. La **disponibilità** dei dati e l'**intuitivà** sono chiaramente i maggiori punti di forza del sistema.

Tuttavia nella mia breve esperienza con il sistema ho già avuto a che fare con una certa quantità di **bug** ed assenza di **feature**, come si può dedurre dalla sezione precedente. Alcune problematiche, ad esempio la mancata possibilità di fare listing dei file ma soprattuto il fatto che il dopo l'elezione di un nuovo master server non si riesca a recuperare tutti i volume ancora attivi ritengo siano difetti piuttosto gravi ma anche risolvibili, all'apparenza, con uno sforzo non eccessivo.