Scuola Superiore
dell'Università degli Studi di Udine

Classe Scientifico-Economica

Colloquio di fine anno

TENSORFLOW PROBABILITY - PROBABILISTIC PROGRAMMING WITH TENSORFLOW AND PHYSICAL APPLICATIONS

Relatore:
Prof. Scagnetto Ivan

Allievo:
Corrado Alessio

Anno Accademico 2019-20

Contents

1 Introduzione

Lo scopo del progetto è la realizzazione di un **file storage distributito**.

Le caratteristiche che deve avere sono:

- *Efficienza*: I tempi di risposta devono essere ragionevoli e non degradare con l'aumento del numero di nodi. L'utilizzo delle risorse a disposizione deve essere massimizzato, ovvero non devono esserci risorse inutilizzate o carichi di lavoro troppo sbilanciati.
- *Accesso concorrente*: Più client possono effettuare richieste in modo concorrente. Deve essere tenuta in considerazione qualche politica di *fairness*, in modo che nessun client sia eccessivamente privilegiato o venga escluso. Deve essere garantita la *consistenza* specificamente in caso di accessi concorrenti alla stessa risorsa.
- *Replicazione dei dati (files)*: Per garantire un buon livello di *fault tolerance*, tutti i dati devono poter essere replicati. In questo modo problematiche su un nodo non compromettono il contenuto di un file.
- *Replicazione dei metadati (struttura del fs)*: La struttura del *file system* deve essere sempre replicata, in modo che non possa essere persa o corrotta .
- *Disponibilità e consistenza*: deve essere massimizzata la disponibilità delle risorse, mantenendo al contempo una consistenza forte. In particolare, ogni richiesta deve riferirsi all'ultima versione della risorsa (se non diversamente indicato).
- *Sicurezza*: Deve essere curata la gestione della sicurezza: sia a livello di connessione (utilizzando opportunamente cifratura e autenticazione) che di accesso (autenticazione dell'utente, privilegi sul fs). Deve essere minimizzata l'esposizione ad attacchi malevoli. Deve essere possibile ripristinare il sistema a seguito di un attacco o errore.

Numero di repliche dei dati Il numero di repliche di un file deve essere proporzionale sia all'importanza che al numero di accessi di esso: documenti con *dati critici* hanno numero di copie maggiore, per agevolarne la *disponibilità* e diminuire le *probabilità di perdita*.

Bilanciamento del carico Per migliorare l'efficienza viene effettuato un bilanciamento del carico: in ogni momento il sistema può copiare o spostare dati in modo da non sovraccaricare un singolo nodo. Inoltre, per il trasferimento in uscita di un documento si cerca di utilizzare il nodo con maggiore *banda disponibile*.

2 Architettura del sistema

Verranno di seguito analizzate diverse possibili architetture del sistema, comparandone pregi e difetti.

2.1 Single-server

La prima idea è quella di avere un *singolo server*, il quale memorizza *sia la struttura del fs che i dati*. Il client interagisce direttamente con esso per effettuare qualsiasi operazione.

Il fs può essere memorizzato a partire da una *directory nel filesystem del server*, oppure in una *partizione separata*. Con entrambe le soluzioni è il sistema operativo ad occuparsi della sua gestione.

La *replicazione dei dati* avviene mediante il salvataggio in diversi dispositivi di memorizzazione (es HDD). Per far ciò la scelta più consona è di utilizzare un sistema **RAID 10**: i dati vengono salvati in copia (RAID 1) per avere *tolleranza ai guasti* e in striping (RAID 0) per aumentare la *velocità di lettura*.

Con schede di rete (e connessioni) multiple è possibile migliorare la tolleranza sia ai *guasti* che alle *congestioni di rete*.

Vantaggi

- Semplice da implementare
- Modello più economico
- Il fs è gestito direttamente dal sistema operativo
- Minima esposizione contro attacchi hacker
- Con gli accorgimenti sopra descritti si ha già una buona fault tolerance.

Svantaggi

- *Single point of failure* per quanto riguarda il server (i dati sono parzialmente protetti dal meccanismo RAID)
- Nessuna protezione per *eventi eccezionali* (eg. catastrofe naturale, incendio nell'edificio, blocco della rete...).

.

Figure 1: Single-server

2.2 Virtual server

La soluzione single-server può essere migliorata aggiungendo un *livello di virtualizzazione*. Il sistema operativo non viene quindi eseguito direttamente sull'hardware ma all'interno di un container/VM.

La macchina virtuale viene periodicamente *copiata* (interamente o in modo incrementale) e salvata in un dispositivo di memorizzazione dedicato.

Nel caso ci fosse un *crash* o *errore critico del sistema operativo*, basta ricaricare la macchina virtuale più recente. La *perdita di dati* è limitata all'età dell'ultimo backup.

Vantaggi

- *Recovery* buona e in tempi brevi in caso di crash
- Naturale implementazione in ambienti di loro natura virtualizzati (es *container*)

Svantaggi

- Durante il *periodo di transizione* il sistema non risponde
- *Prestazioni* leggermente ridotte a causa della virtualizzazione

Figure 2: Single-server con virtualizzazione

2.3 Cloud backup

Il backup può essere effettuato nel cloud, al posto che in un dispositivo dedicato all'interno del datacenter. Inoltre il backup può essere esteso anche ai dati, oltre che al fs.

Vantaggi

- Protezione in caso di *eventi eccezionali*: i dati non risiedono in un unico luogo fisico
- Il provider può fornire ulteriori garanzie di replicazione

Svantaggi

- In generale effettuare un backup nel cloud, anche se solo incrementale, richiede un *elevato utilizzo di banda* e *tempi maggiori*
- I *costi* possono essere maggiori

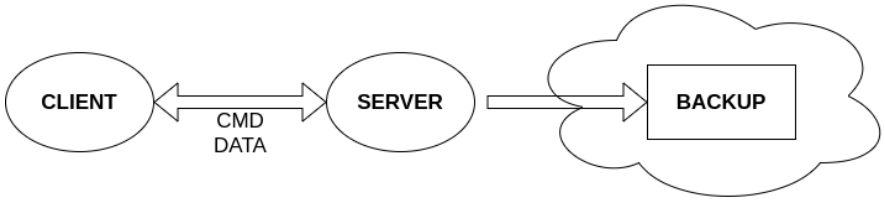


Figure 3: Aggiunta del backup nel cloud

2.4 Server as distributed database

Al posto di utilizzare il filesystem del sistema operativo, è possibile utilizzare un database distribuito. Sia metadati che dati vengono *distribuiti automaticamente* nei nodi, garantendo consistenza e fault tolerance.

Vantaggi

- I dati vengono automaticamente replicati e gestiti dal dbms
- Utilizzando un dbms in commercio, si beneficia dall'avere tutto già pronto e costantemente aggiornato.

Svantaggi

- Maggiore complessità
- La connessione avviene tra il client ed un nodo, il quale poi distribuisce i dati bidirezionalmente agli altri nodi. Se questo nodo dispone di una banda limitata, ciò può rappresentare un bottleneck dal punto di vista delle prestazioni. Inoltre l'utilizzo di banda complessiva può essere doppio rispetto al necessario: nel caso i dati vadano trasferiti dal client al server (gateway) al server di destinazione o vice versa.

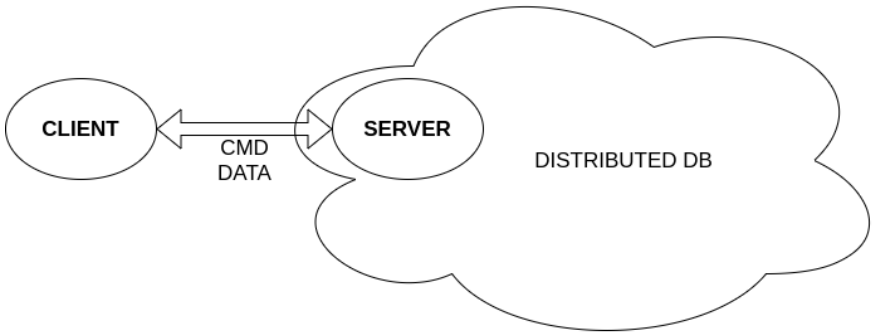


Figure 4: Utilizzo di un database distribuito

2.5 Meta+Data server

L'idea è di far *comunicare direttamente il client con i server in cui verranno memorizzati i dati*. Così facendo si elimina il bottleneck del nodo che prima doveva fungere da "gateway".

Un primo sviluppo è quello di *separare i compiti*: si hanno quindi un **meta server** e uno o più **data server**. Il meta server si occupa di memorizzare e gestire il filesystem (metadati), mentre i data server memorizzano soltanto i dati conenuti nei documenti.

Per effettuare una qualsiasi *operazione di trasferimento* il client si rivolge inizialmente al meta server, dal quale ottiene la configurazione per contattare il giusto data server e trasferire i dati. Le *operazioni sul filesystem* sono eseguite all'interno del meta server. In questo modo ci sono *due comunicazioni bidirezionali*: client-meta, per la gestione del fs; client-data, per il trasferimento dei dati.

Per migliorare le prestazioni, soprattutto nel caso in cui il client abbia una banda più ampia rispetto ai data server, i documenti di grandi dimensioni vengono spezzati in chunks e distribuiti.

Come nel caso single-sever, il meta server è unico, virtualizzato e sottoposto a backup periodico.

Vantaggi

- Trasferimenti concorrenti con data server multipli permettono di migliorare le prestazioni nel caso in cui il client abbia una banda elevata rispetto ai dataserver
- Un *unico meta server* permette di mantenere in modo semplice la consistenza del filesystem

Svantaggi

- La frequenza dei *backup* del meta server è fondamentale per minimizzare la perdita di dati in caso di fault
- Tutti i data server devono essere *raggiungibili* dal client (maggiore attenzione alla sicurezza e struttura della rete)

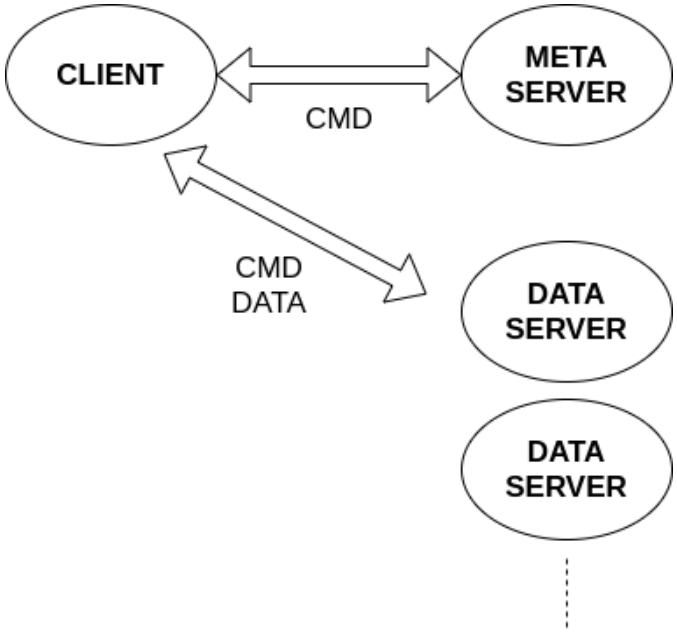


Figure 5: Separazione di meta e data server

2.6 Meta+Data server 2

Facendo dialogare meta e data server, è possibile:

1. Spostare l'*onere di configurazione dei data server* da client a meta server
2. Gestire in qualsiasi momento la *replicazione dei dati*, ad esempio per bilanciare il carico o far fronte alla perdita di un nodo
3. Il meta server può monitorare *prestazioni* e *stato di salute* dei data sever

Il client quindi si limita a trasferire i dati da/verso data server utilizzando token forniti dal meta server.

Vantaggi

- Il sistema può *riconfigurarsi* in ogni momento per far fronte a qualsiasi evenienza
- Ruolo del client semplificato
- Maggiore sicurezza (minore esposizione) perchè il client non può inviare comandi ai data server

Svantaggi

- Maggiore complessità nel gestire operazioni *concorrenti* ed *asincrone* in nodi diversi

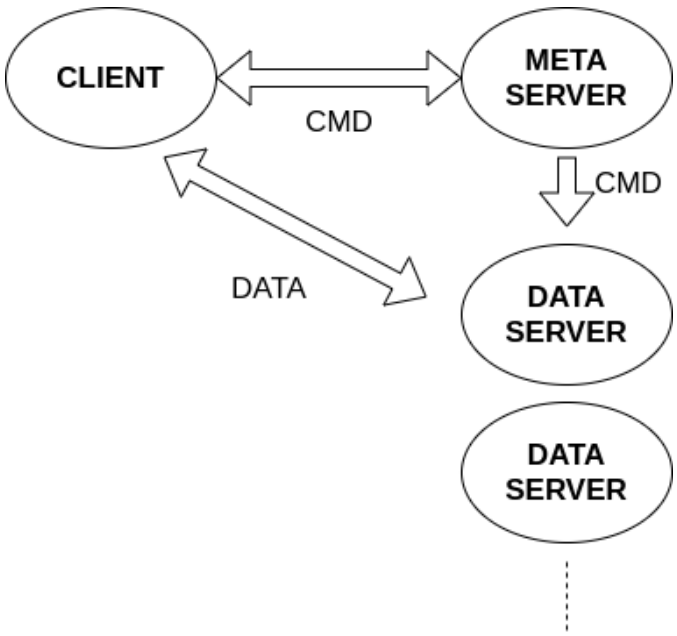


Figure 6: Meta + data server, il meta server dialoga con i data server

2.7 Meta server as distributed system

In questa soluzione non è presente un unico meta server, ma viene utilizzato un sistema distribuito. In generale è preferibile utilizzare un dbms distribuito presente in commercio per gestire i dati a basso livello.

Come dimostrato dal teorema CAP bisogna rinunciare alla disponibilità per garantire la coerenza del fs.

Vantaggi

- Possibilità di utilizzo di software in commercio
- Maggiore fault tolerance dei metadati
- Minore tempo di down in caso di malfunzionamenti

Svantaggi

- Maggiore difficoltà nell'implementazione dei meta server e dei protocolli di comunicazione tra i vari componenti.
- Maggiore overhead (e quindi minori prestazioni) nel caso di sistemi di piccole/medie dimensioni

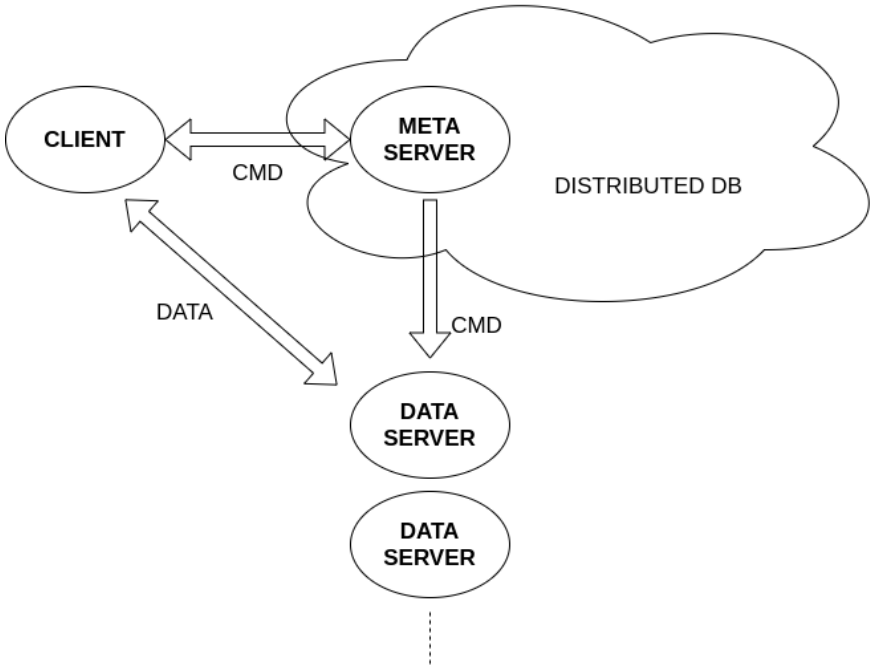


Figure 7: Il meta server è implementato come database distribuito

3 Struttura del filesytem

Il filesystem ha una struttura non convenzionale. Descriveremo quindi la semantica dei path e i metadati ad essi associati.

3.1 Semantica dei path

Nei *sistemi tradizionali* il filesystem è formato da *cartelle, file e link*. La *struttura* è ad albero oppure grafo, con o senza la possibilità di cicli. La cartella "base" è root (nei sistemi unix-like denotata con la stringa "/"). Un *path* può essere associato ad uno delle tre tipologie sopra menzionate; il filesystem memorizza questa associazione oltre che ad altri metadati.

Il simbolo '/' viene usato per separare i *componenti di un path*: ad esempio il path "/dir1/dir2/file1" identifica il file "file1" figlio della directory "dir2" che a sua volta è figlia della directory "dir1" che è figlia di root. Per questo '/' compare tra i caratteri vietati per la nominazione di un oggetto.

Nel *sistema presentato* esistono soltanto i **documenti**. Essi sono *contenitori di dati*, ed hanno associati alcuni *metadati* (in versione semplificata rispetto unix).

Un path che termina con un carattere diverso da '%' rappresenta un *singolo documento* (es: "documento1", "home/lavoro#foto/mare-foto1"). Un path che termina con '%' rappresenta l'*insieme di documenti* che ha come prefisso i caratteri antecedenti al simbolo finale (es. "home/lavoro#foto%" include tutti i path che rispettano l'espressione regolare "home/lavoro#foto*"). Il carattere '%' può essere usato solo come terminatore. Il carattere ';' è vietato in quanto usato come separatore degli argomenti.

Questa organizzazione possiede intrinsecamente una *struttura ad albero*. I path che rappresentano insiemi di documenti sono chiamati *repository*. Le operazioni effettuate su repository hanno effetto su tutti i documenti (anche non ancora creati) appartenenti alla repository. La repository root è naturalmente identificata con il path "%". "" è un documento valido: si consiglia di usarlo per salvare le informazioni sul sistema (es proprietario, descrizione, contatti) in formato testuale con codifica utf.

3.2 Metadati di un documento

I metadati associati ad un documento sono:

- uid: identificatore univoco dell'oggetto, in formato UUID Type 1
- path: deve rappresentare un documento, stringa utf
- created: timestamp di creazione in formato UTC
- size: dimensione in byte, 0 per le directory, formato unsigned int
- owner: utente proprietario, stringa utf
- priority: numero minimo di copie, coincide con la priorità, formato unsigned int

- checksum: checksum sha1, formato esadecimale
- deleted: timestamp di eliminazione (vuoto se non eliminato), formato UTC

4 Comandi gestiti client

Di seguito l'elenco e descrizione dei comandi gestiti dal client, su cui deve basarsi l'implementazione del sistema.

- list: lista i metadati di un path (documento o documenti attualmente in una repository)
- get: download di un documento
- push: upload di un documento
- rem: eliminazione di path (documento o repository)
- lock: lock e unlock di un path (documento o repository)
- setPriority: aggiornare la priorità di un path (documento o repository)

5 Protocollo di comunicazione

Il protocollo di comunicazione tra nodi diversi si basa sullo scambio di messaggi. Il flusso di dati è binario. Viene utilizzata una connessione di tipo TCP per garantire ordine di consegna, affidabilità, gestione della connessione.

Il formato dei comandi è testuale (codifica utf). Un comando è formato dalla parola chiave che lo identifica e dalla lista degli argomenti. Il separatore utilizzato è il punto e virgola; i comandi sono terminati dal newline.

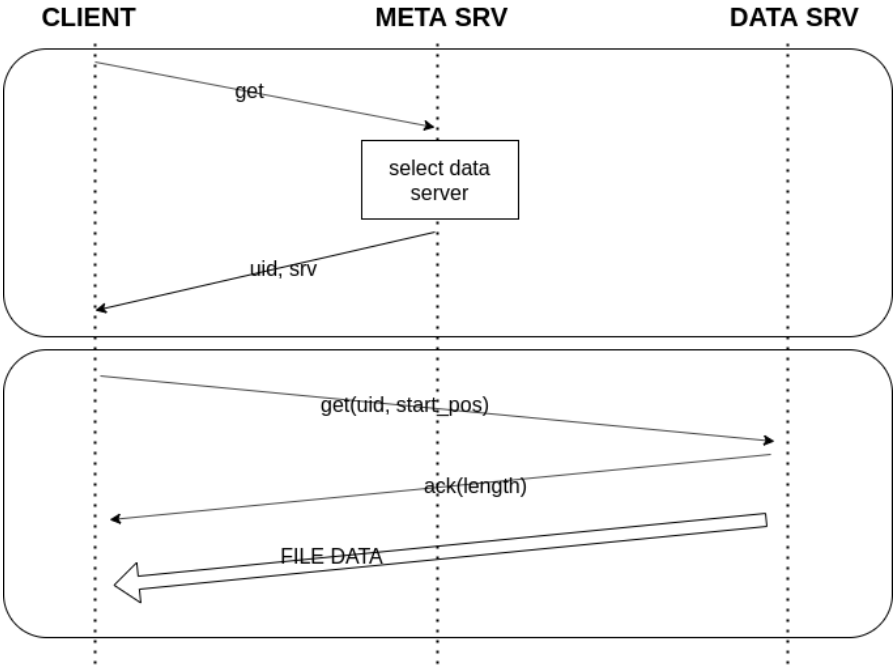
Di seguito i flussi di dati coinvolti nelle diverse tipologie di richieste.

5.1 get

La richiesta di tipo *get* richiede il trasferimento in entrata di un documento.

Inizialmente il *client* contatta il *meta server*, richiedendo un particolare *documento* (mediante path o uid). Il *meta server* seleziona il *data server* ottimale per gestire la richiesta (deve essere online, contenere la risorsa e non essere sovraccarico). Il *meta server* risponde quindi al client specificando l'uid della risorsa (richiesto per identificarla) e l'indirizzo del *data server* a cui collegarsi. Il *client* si disconnette dal *meta server*.

Il *client* si connette al *data server* inviando una richiesta *get(uid,start_pos)*. L'argomento *start_pos* indica da che byte iniziare a trasferire il documento (indice 0-based). Se la risorsa è disponibile (come dovrebbe essere) il *data server* invia un *ack* seguito dal flusso di dati del documento. Altrimenti risponde con *err* seguito dai dettagli.



5.2 push

La richiesta di tipo *push* richiede il trasferimento in uscita di un documento.

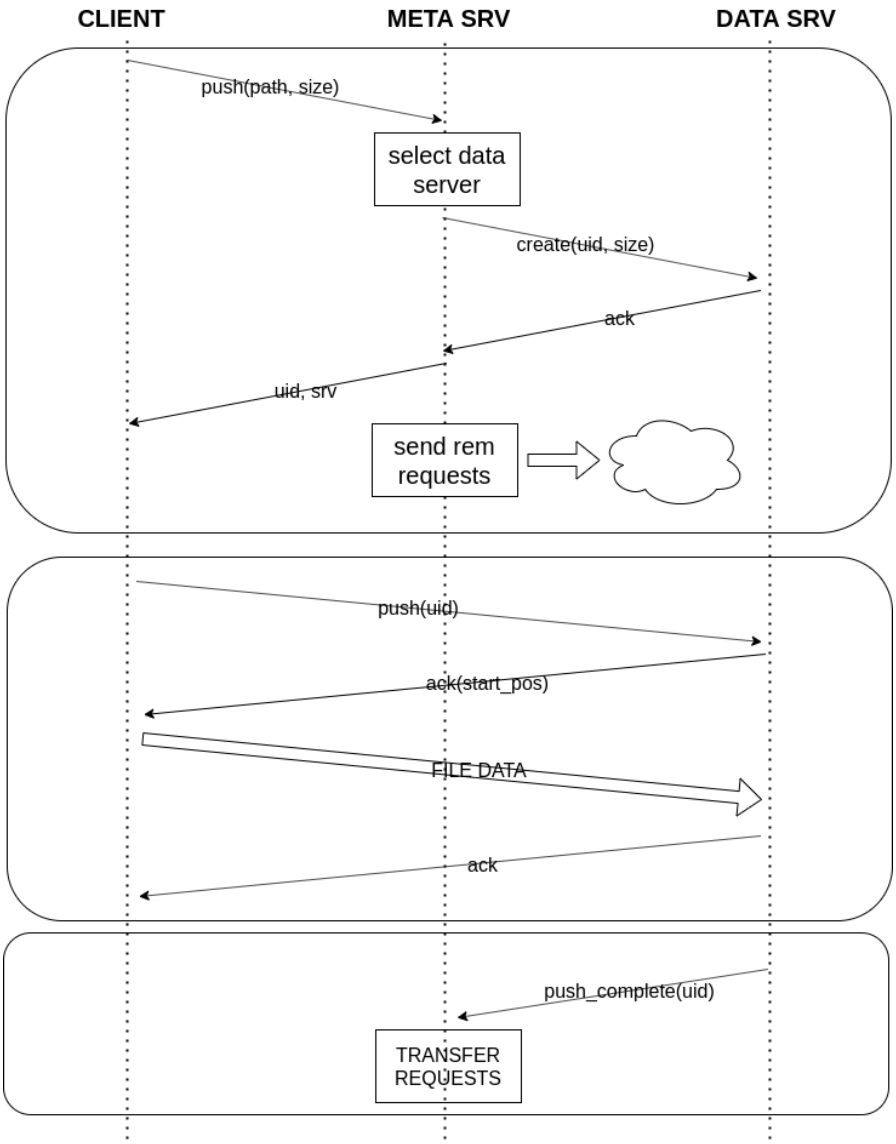
Inizialmente il *client* contatta il *meta server*, inviando *path* e *size*. Il *meta server* seleziona il *data server* ottimale per gestire la richiesta (deve essere online, poter contenere la risorsa e non essere sovraccarico). Il *meta server* si occupa anche di generare un nuovo *uid* da assegnare alla risorsa.

Il *meta server* si collega al *data server* dove verrà inizialmente salvata la risorsa, inviando un comando *create(uid, size)*. Questo comando predispone il *data server* per ospitare un nuovo documento con tale uid e dimensione. Se non ci sono errori, il *data server* risponde al *meta server* con *ack*. Altrimenti con *err* seguito dai dettagli.

Il *meta server* risponde quindi al client specificando l'uid della risorsa (richiesto per identificarla) e l'indirizzo del *data server* a cui collegarsi. Il *meta server* si disconnette dal *client* e invia le eventuali richieste di eliminazione ai data server (se il documento è una nuova versione di uno esistente).

Inizia quindi il caricamento del documento. Il *client* si collega al *data server* e invia una *push(uid)*. Se non ci sono errori, il *data server* risponde con *ack* e invia la posizione (indice 0-based) *last_pos* del primo byte non trasferito (equivalente al numero di byte già trasferiti). A quel punto il *client* trasferisce la porzione rimanente di documento. Al termine del trasferimento, se non ci sono errori, il *data server* risponde con *ack*. La connessione viene chiusa.

Quando il trasferimento è completato con successo, inizia la terza fase. Il *data server* invia un messaggio del tipo *push_complete(uid)* al *meta server*. Il meta server inizia quindi ad inviare ai *data server* interessati le richieste di trasferimento, per raggiungere il grado di replicazione voluto.

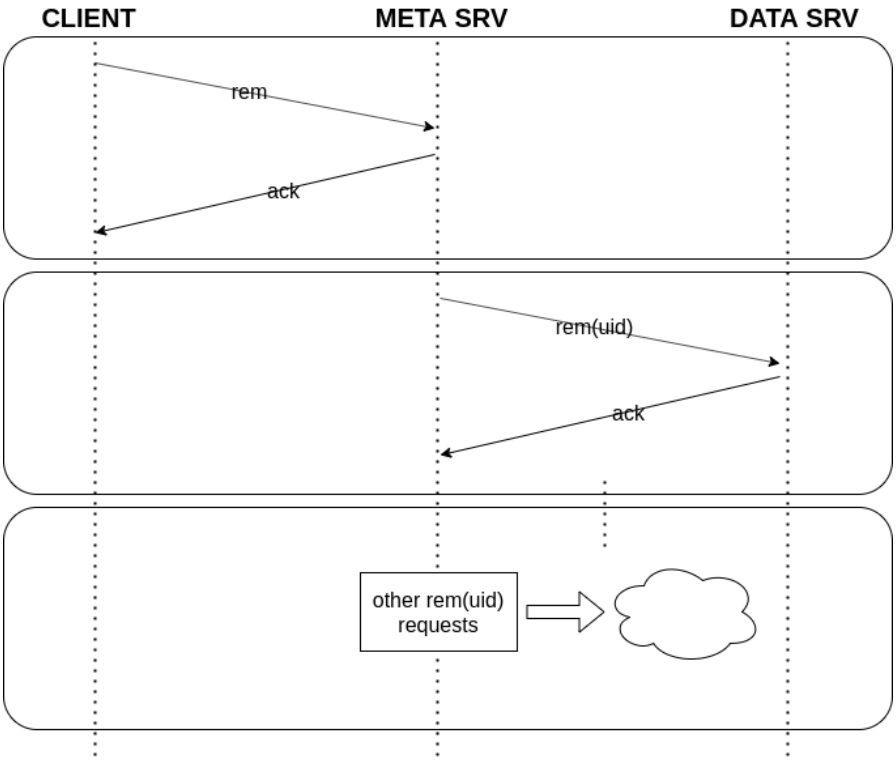


5.3 rem

La richiesta di tipo *rem* richiede l'eliminazione di un intero path o singolo uid dallo storage.

Il *client* invia al *meta server* una richiesta *rem*. Se non ci sono errori, il *meta server* risponde con *ack* e chiude la connessione.

Nel caso la richiesta di eliminazione sia di tipo fisico, il *meta server* contatta separatamente tutti i *data server* contenenti dati relativi a quel particolare *uid* e invia una richiesta *rem(uid)*.

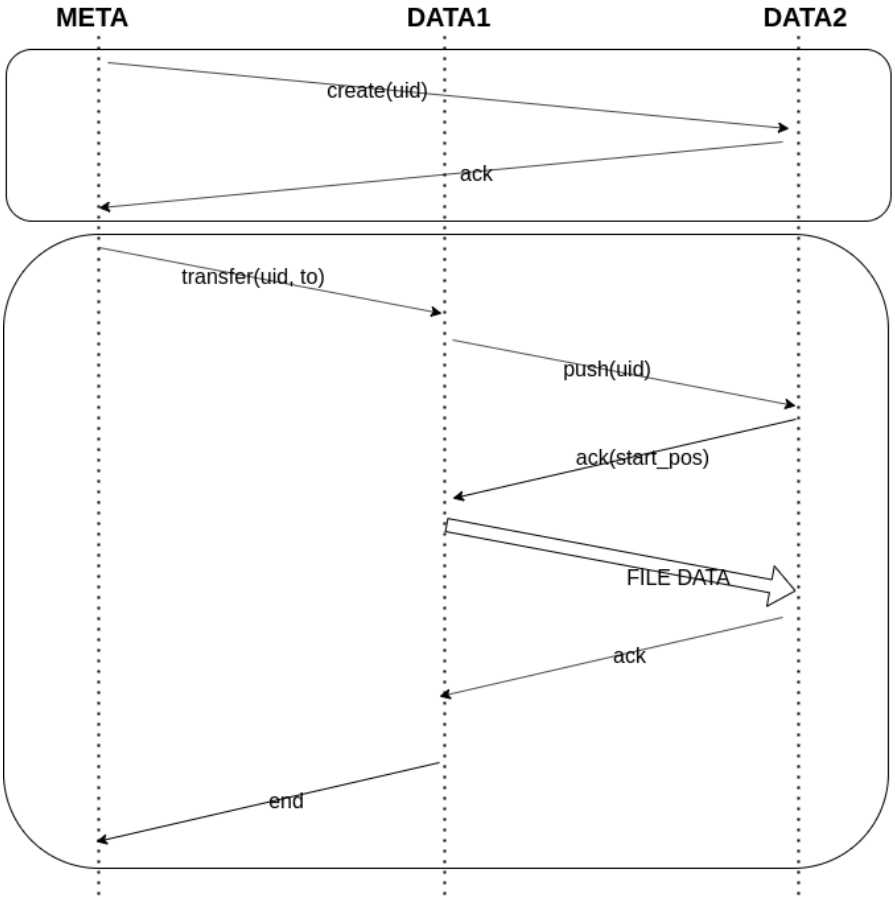


5.4 transfer

La richiesta di tipo *transfer*, effettuata esclusivamente dal metaserver, richiede la copia di un documento da un *data server* ad un altro. Viene usata per trasferire i dati tra i nodi in modo da gestire la ridondanza.

Inizialmente, come nel caso di una *push*, il meta server contatta il *data server* di destinazione (*DATA2*) tramite una richiesta *create(uid)*, per inizializzarlo a ricevere il documento. Se non ci sono errori il *data server* risponde con *ack*. La connessione viene chiusa.

A questo punto il *meta server* invia al *data server* di origine (*DATA1*) una richiesta *transfer(uid, to)* per ordinare il trasferimento. Se *DATA1* riesce a connettersi con successo a *DATA2*, risponde a *META* con *begin_transfer(uid)*. *META1* invia a *META2* una richiesta di tipo *push(uid)* e da luogo al trasferimento. Al termine *DATA1* invia un messaggio *end_transfer(uid)* a *META* per notificare l'avvenuto trasferimento.



6 Logging

Tutte le operazioni prima di essere eseguite vengono registrate in appositi database di log. In questo modo, se un'operazione viene interrotta per qualsiasi motivo, può essere analizzata e rieseguita, in modo da non lasciare spazzatura.

Ciò è molto utile anche in fase di debugging/testing: analizzare lo stacktrace ed il flusso di operazioni avvenute precedentemente ad un problema è il modo più efficace per comprenderne le cause. Il modulo *logging*, parte della libreria standard di Python3, permette di organizzare le operazioni di log in modo compatto ed efficace.

7 Sicurezza ed autenticazione

Tutte le connessioni vengono cifrate a livello intermedio tramite protocollo TLS over TCP. In questo modo la cifratura è invisibile alla logica dell'application e può in ogni momento essere modificata dal programmatore. Inoltre, essendo una tecnologia ampiamente diffusa e collaudata, presenta una vulnerabilità minima.

L'autenticazione da client a server avviene tramite token all'inizio della connessione. La password di ciascun utente può essere cambiata in ogni momento dall'utente stesso, dopo essersi loggato.

L'autenticazione da server a server avviene tramite certificato digitale.

Per garantire sia l'integrità che la sicurezza (per evitare modifiche volute) viene conservato e verificato il checksum (sha1) di tutti i documenti in ingresso.

8 Generazione di una chiave identificativa

Il problema della generazione di una chiave identificativa è spesso sottovalutato. Consiste nella creazione di un codice che identifichi univocamente una risorsa, un oggetto o un sistema. Questa chiave deve avere la caratteristica di unicità, ovvero non devono esistere codici duplicati: altrimenti non sarebbe più possibile identificare correttamente la risorsa. Inoltre, per ragioni di efficienza, la chiava deve avere lunghezza limitata, in modo da limitarne il tempo di comparazione e lo spazio occupato. In genere le chiavi non hanno un significato, quindi possono essere interpretate indifferentemente come interi o stringhe esadecimali.

Strategie di generazione di una chiave Alcune possibili strategie di generazione di chiavi sono:

- Contatore intero
- Funzione di hash applicata ad un insieme di proprietà variabili con il tempo
- Numero casuale
- UUID tipo 1

Il caso più semplice è quello del contatore intero. Solitamente si implementa con un contatore che parte da 0 e viene incrementato di un unità ad ogni generazione. I vantaggi sono: semplicità di implementazione e garanzia di unicità nel contesto del generatore, ovvero il generatore garantisce una sequenza di valori senza duplicati. Questa soluzione può essere implementata efficacemente soltanto all'interno di un sistema che disponga di un solo generatore, quindi di un solo nodo. Inoltre ciò espone all'esterno il numero

di chiavi generate sin ora, informazione che potrebbe essere sfruttata per attacchi al sistema.

Per superare il problema dell'unicità del nodo una soluzione è prependere al codice l'identificativo del nodo. Ciò garantisce l'unicità della chiave nell'insieme generato dall'intero cluster. È necessario che gli identificativi dei nodi siano univoci, pena il fallimento del sistema. Questo potrebbe essere un problema nel caso venga effettuato un merge tra cluster diversi: se due nodi avevano lo stesso identificativo, almeno uno di essi deve essere rinominato, assieme a tutte le chiavi da esso generate. Ciò potrebbe non essere possibile.

Una delle soluzioni più comunemente usate per la generazione di identificativi è l'utilizzo degli *UUID*. Gli **UUID** (Universally Unique Identifier) sono stringhe binarie di 128 cifre generate con precise regole, dipendenti dalla versione. Spesso vengono rappresentati in esadecimale ed compaiono tra i tipi primitivi di molti linguaggi (tra cui CQL).

Gli UUID di tipo 4 sono generati casualmente. Se viene utilizzato un generatore con distibuzione uniforme, per avere il 50% di probabilità di registrare almeno una collisione è necessario generare $2.71 * 10^{18}$ valori. Questa probabilità, seppur non nulla, quasi sempre viene trascurata. Quando però la generazione di un duplicato potrebbe portare il sistema in uno stato inconsistente, è necessario adottare accorgimenti specifici per riportare il sistema ad uno stato safe.

Gli UUID di tipo 1, o *TimeUUID*, sono invece generati a partire da istante temporale (UTC), un identificativo del nodo e un numero di sequenza. Come identificativo del nodo si adotta l'indirizzo MAC della scheda di rete del pc, il quale è garantito essere univoco dal costruttore. La presenza dell'istante temporale permette di ordinare gli identificativi per ordine cronologico. L'identificativo del nodo impedisce collisioni tra nodi diversi. Il numero di sequenza elimina le collisioni (fino a 2^{12}) per elementi generati dallo stesso nodo nello stesso millisecondo: la capacità di generazione garantita è quindi di 4096 UUID tipo 1 al millisecondo.

Il problema principale degli UUID tipo 1 è causata dalla presenza di due nodi associati alla medesima scheda di rete, per i quali la probabilità di conflitto è estremamente elevata. Ciò si risolve imponendo l'esecuzione di una sola istanza del software per host, dato un cluster.

https://tools.ietf.org/html/rfc4122.html

9 Problema della consistenza in un sistema distribuito

I sistemi distribuiti sono per loro natura soggetti a frequenti errori, ritardi di comunicazione e situazioni di concorrenza. È quindi fondamentale assicurarsi di mantenere la consistenza dello stato globale in modo che eventuali vincoli di integrità non siano violati.

Un esempio lampante è quello delle richieste del tipo "INSERT .. IF NOT EXISTS". Tale operazione non può essere valutata in un solo nodo, in quanto l'identificativo cercato potrebbe essere stato inserito in un altro nodo ma non ancora propagato all'intero sistema.

Per verificare l'esito di tale richiesta è quindi necessario adottare una politica specifica, quale potrebbe essere il meccanismo del **QUORUM**. Essa stabisce che se in un istante la maggioranza assoluta ($Q = floor(N/2) + 1$, con N numero di nodi) dei nodi concorda su una proprietà, allora essa in quell'istante è vera. Capiamo meglio con un esempio: viene inserito il valore 1 all'interno di un nodo. Esso propaga il valore ad almeno $Q - 1$ nodi distinti. Al termine dell'operazione almeno la Q nodi contengono il valore 1. Per verificare la presenza del valore 1, viene effettuata una richiesta ad almeno Q nodi distinti qualsiasi. Per il principio della piccionaia, almeno uno di essi conterrà il valore specificato e quindi risponderà "yes", dando il risultato corretto.

Il nuovo problema da affrontare è che l'algoritmo precedente vale se e solo se tutte le richieste vengono valutate allo stesso istante, e il risultato ottenuto è garantito solo per lo stesso istante. Questo però è impossibile, in quanto comunicazioni e computazioni non sono istantanei. La soluzione adottata da Cassandra è l'utilizzo del protocollo *PAXOS* per effettuare *Lightweight transactions*. Le LWT sono utilizzate in particolare nell'esecuzione di comandi "INSERT ... IF NOT EXISTS" e "UPDATE ... IF CONDITION", in modo da garantire la consistenza.
https://en.wikipedia.org/wiki/Paxos_(computer_science)

Inoltre, Cassandra permette di stabilire il livello di consistenza per ogni query di tipo select. Esso può essere ANY, ONE, TWO, THREE, ..., QUORUM, ..., ALL, in base alle necessità del sistema. Questa scelta consente di bilanciare il tradeoff tra *disponibilità* e *consistenza*. ANY fornisce la maggiore disponibilità (basta che un nodo qualsiasi risponda), ALL la maggiore consistenza (tutti i nodi devono concordare). Per questo progetto è stata scelta l'opzione QUORUM, in quanto garantisce forte consistenza ma tolleranza ai guasti fino a $floor(N)$ nodi.
https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/dml/dmlConfigConsistency.html

10 Implementazione

Di seguito verranno mostrati i punti salienti dell'implementazione demo da me sviluppata. Questa implementazione non è commercializzabile, in quanto incompleta, ma ha lo scopo di provare tutte le funzionalità del sistema sopra descritto.

10.1 Strumenti software

Il linguaggio scelto per implementare client, dataserver e metaserver è **Python 3**. Questo linguaggio ha numerosi vantaggi: è *platform-independent*, è *semplice* ma *efficace* da utilizzare, permette la scrittura di codice abbastanza *compatto*, possiede una collezione

di *librerie* praticamente illimitata, si adatta bene alla programmazione *concorrente*. Unico punto debole è l'*efficienza*, minore rispetto al C++, ma comunque più che sufficiente per lo scopo.

Come dbms per il metaserver ho scelto **Apache Cassandra**, uno dei più famosi sistemi *NoSQL* distribuiti in commercio. Gestisce automaticamente la *replicazione* e *concorrenza* all'interno del cluster, con efficacia, in modo da assicurare *disponibilità* e *affidabilità*. Le *prestazioni* sono ottime, come il supporto della *community* e la *documentazione*. *Cql* (Cassandra Query Language) una sintassi molto simile a SQL, cosa che agevola molto l'intervento di programmatori non specializzati.

Dato che i dataserver devono essere più leggeri possibile, come dbms per i dataserver ho scelto **SQLite3**. E' completamente integrato nella libreria standard di Python 3, quindi non necessita dell'installazione di software aggiuntivo. Tutti i dati vengono memorizzati all'interno di un *unico file compresso*. Le *performances* sono più che adatte per le operazioni richieste. Sia buona *documentazione* che *facilità* di utilizzo sono punti a favore. Il linguaggio è un *dialetto SQL*, molto simile a quello degli altri sistemi (l'unica differenza importante stà nei tipi di dato, che sono semplificati).

I *file di configurazione* sono in formato **YAML**: un formato *open*, *human-readable* tra i più diffusi al mondo e intuitivi.

10.2 Struttura del codice

Il codice sia di dataserver che metaserver si articola in 5 parti:

- Caricamento della cofigurazione
- Interfaccia di connessione e scambio dati
- Interfaccia al database
- Operazioni periodiche
- ServerSocket e relativo handler

Il client dispone soltanto dell'interfaccia di connessione/scambio dati e dell'implementazione delle operazioni.

10.3 Caricamento della configurazione

Sia per metaserver che dataserver è obbligatorio fornire il percorso del file di configurazione come primo argomento.

10.3.1 Metaserver

Codice

```
if len(sys.argv) > 1:
    configFile = sys.argv[1]
    print("load config", configFile)
    config = yaml.full_load(open(configFile, "r"))
    print("config", config)
else:
    logging.error("Please provide config file")
    exit(1)

database = Database()

for dataserver in config["dataservers"]:
    database.addDataServer(dataserver)
    print("add dataserver", dataserver)
```

Esempio di configurazione

```
dataservers:
    - localhost:10010
    - localhost:10011
    - localhost:10012
```

10.3.2 Dataserver

Codice

```
if len(sys.argv) > 1:
    configFile = sys.argv[1]
    print("load config", configFile)
    config = yaml.full_load(open(configFile, "r"))
    print("config", config)
    NAME = config["name"]
    HOST = config["host"]
    PORT = config["port"]
    METASERVER = config["metaserver"]
    workingDir = config["workingDir"]
else:
    logging.error("Please specify config file")
    exit(1)

if not os.path.exists(workingDir):
    os.makedirs(workingDir)
    os.chdir(workingDir)
    logging.info("work on %s", os.getcwd())

#create db if not exists
if not os.path.exists("database.db"):
    with sqlite3.connect('database.db') as conn:
```

```
23          with open("../../dataserver/create_db.sqlite3", "r") as sql:
24              conn.executescript(sql.read())
25
26  database = Database()
27  database.setStats(config["storage"], config["downspeed"], config["upspeed"
        ])
28
29  SERVER = str(HOST) + ":" + str(PORT)
```

Esempio di configurazione

```
1  name: dataserver10012
2  workingDir: /home/ale/Dropbox/tesina iot/code/data/dataserver10012
3  host: localhost
4  port: 10012
5  storage: 100000000
6  downspeed: 50
7  upspeed: 10
8  metaserver: localhost:10000
```

10.4 Interfaccia di connessione e scambio dati

L'interfaccia è unica per client, metaserver e dataserver. E' implementata come *decorator* per la classe socket e StreamRequestHandler. Utilizza gli oggetti *rfile* (lettura) e *wfile* (scrittura), ottenuti dal socket, per effettuare i trasferimenti.

Il trasferimento dati avviene in *modalità binaria*, per ragioni di efficienza.

I file vengono scritti dal metodo ad alta efficienza *socket.sendfile*, il quale permette di evitare il *double buffering*. Vengono letti a blocchi di 1024 byte con il metodo *socket.read*.

Codice

```
1  def CustomConnection(cls):
2      def addrFromString(self, addr):
3          ip, port = addr.split(":")
4          return (ip, int(port))
5
6      def write(self, *args):
7          res = ";".join([str(i) for i in args]).strip() + "\n"
8          logging.info("write %s", res)
9          self.wfile.write(res.encode())
10
11     def readline(self):
12         line = self.rfile.readline().strip().decode().split(";")
13         logging.info("readline %s", line)
14         return line
15
16     def readFile(self, size, outFile):
17         size = int(size)
18         pos = 0
```

```
19          while pos != size:
20              chunk = min(1024, size - pos)
21              # print("read", chunk, "bytes")
22              data = self.rfile.read(chunk)
23              outFile.write(data)
24              pos += len(data)
25
26      def writeFile(self, filepath, startIndex):
27          if type(startIndex) != int:
28              startIndex = int(startIndex)
29          with open(filepath, "rb") as file:
30              self.sendfile(file, startIndex)
31
32      setattr(cls, "writeFile", writeFile)
33      setattr(cls, "addrFromString", addrFromString)
34      setattr(cls, "readFile", readFile)
35      setattr(cls, "write", write)
36      setattr(cls, "readline", readline)
37
38      return cls
39
40
41  @CustomConnection
42  class Connection(socket.socket):
43      def __init__(self, endpoint):
44          super(Connection, self).__init__(socket.AF_INET, socket.SOCK_STREAM
                )
45          self.connect(self.addrFromString(endpoint))
46          self.wfile = self.makefile('wb', 0)
47          self.rfile = self.makefile('rb', -1)
48
49
50  @CustomConnection
51  class DataServerHandler(StreamRequestHandler):
52      def sendfile(self, *args, **kwargs):
53          self.connection.sendfile(*args, **kwargs)
54
55      ...code...
```

10.5 Interfaccia al database

Per interfacciarsi al database metaserver e dataserver hanno degli oggetti dedicati, chiamati *Database*. Essi espongono tutte le query sotto forma di *API*: in questo modo gli utilizzatori del database sono svincolati dall'effettiva implementazione sottostante. Questo è un vantaggio sia in termini di *incapsulamento*, che rende più semplice la progettazione e manutenzione del software, che nel caso in cui si voglia cambiare dbms, perchè l'utilizzatore non deve essere modificato.

10.5.1 Metaserver

Struttura

```
1  //il fattore di replicazione deve essere <= al numero di server a
       disposizione
2  create keyspace metaserver with replication = {'class' : 'SimpleStrategy',
       'replication_factor' : 1};
3
4
5  use metaserver;
6
7
8  //un metaserver
9  create table dataserver(
10     server text,      //indirizzo in formato host:port del dataserver
11     online boolean,     //se in questo momento (all'ultima rilevazione) e'
           online
12     capacity bigint,     //capacita' in byte totale
13     remaining_capacity bigint,     //capacita' in byte disponibile (totale-
           utilizzata)
14     available_down float,     //banda in Mb/s disponibile per il download
15     available_up float,       //banda in Mb/s disponibile per l'upload
16     primary key(server)
17 );
18
19 //un documento
20 create table object(
21     uid uuid,
22     path text,        //path associato al documento
23     created timestamp,     //istante di creazione (nel metaserver)
24     owner text,     //proprietario (colui che l'ha creato)
25     size bigint,     //dimensione in byte
26     priority int,     //livello di priorita' (di replicazione)
27     checksum text,
28     deleted timestamp,       //istante di eliminazione (NULL se non
           eliminato)
29     primary key(uid)
30 );
31
32 //lock attivi
33 create table object_lock(
34     path text,    //path del documento
35     user text,      //utente che detiene il lock
36     primary key(path)
37 );
38
39 //log delle performance dei dataserver
40 create table performance_log(
41     server text,            //indirizzo del dataserver nel formato host:
           port
42     time timestamp,     //istante di registrazione del record
43     online boolean,
44     capacity bigint,
45     remaining_capacity bigint,
46     available_down float,
```

```
47      available_up float,
48      primary key(server, time)
49  );
50
51  //oggetti che vanno verificati
52  create table pending_object(
53      uid uuid,           //uid dell'oggetto
54      enabled boolean,        //se false, l'oggetto non puo essere attualmente
            processato perche' non e' disponibile o non sono disponibili
           server in cui replicarlo
55      primary key(uid)
56  );
57
58  //documenti salvati nei dataserver
59  create table stored_object(
60      uid uuid,       //documento
61      server text,        //indirizzo del server in cui e' salvato, nel
           formato host:port
62      created timestamp,
63      complete boolean,
64      primary key(uid, server)
65  );
66
67  //utilizzato per le ricerche di prefissi del tipo "where path like '<prefix
       >%'"
68  create custom index object_path_idx on object(path)
69  using 'org.apache.cassandra.index.sasi.SASIIndex';
70
71  //utilizzato per selezionare l'ultima versione di un documento, dato il
       path (con cql non e' possibile ordinare nella query)
72  create materialized view pathToObject as select * from object where path is
        not null
73  primary key (path, uid)
74  with clustering order by (uid desc);
```

Codice

```
1   class Database:
2       def __init__(self):
3           from cassandra import ConsistencyLevel
4           from cassandra.cluster import Cluster, ExecutionProfile,
                EXEC_PROFILE_DEFAULT
5           from cassandra.policies import WhiteListRoundRobinPolicy,
                DowngradingConsistencyRetryPolicy
6           from cassandra.query import tuple_factory
7
8           profile = ExecutionProfile(
9               consistency_level=ConsistencyLevel.QUORUM,
10              serial_consistency_level=ConsistencyLevel.SERIAL
11          )
12
13          cluster = Cluster(execution_profiles={EXEC_PROFILE_DEFAULT: profile
                })
```

```
14          self.cluster = Cluster(protocol_version=4)
15
16          self.session = self.cluster.connect("metaserver")
17
18
19      def addDataServer(self, addr):
20          self.session.execute("insert into dataserver(server, online) values
                (%s, false)", (addr, ))
21
22      def addObject(self, uid, path, owner, size, priority, checksum):
23          if type(uid) == "str":
24              uid = UUID(uid)
25          created = datetime_from_uuid1(uid)
26          size = int(size)
27          priority = int(priority)
28          self.session.execute("insert into object(uid, path, created, owner,
                size, priority, checksum) "
29                               "values (%s, %s, %s, %s, %s, %s, %s)"
30                               , (uid, path, created, owner, size, priority,
                                    checksum))
31          return uid
32
33      def addStoredObject(self, uid, server, complete=False, created=None):
34          uid = makeUUID(uid)
35          if created is None: created = datetime.now()
36          self.session.execute("insert into stored_object(uid, server,
                created, complete) values (%s, %s, %s, %s)",
37                               (uid, server, created, complete))
38          return id
39
40      def lockPath(self, path, user="root"):
41          return self.session.execute("insert into object_lock(path, user)
                values (%s, %s) if not exists",
42                                      (path, user)).one().applied
43
44      def unlockPath(self, path):
45          self.session.execute("delete from object_lock where path = %s", (
                path, ))
46
47      def getPathLock(self, path):
48          r = self.session.execute("select * from object_lock where path = %s
                ", (path, )).one()
49          if r is None:
50              return False, ""
51          else:
52              return True, r.user
53
54      def getObjectByUid(self, uid):
55          uid = makeUUID(uid)
56          return self.session.execute("select * from object where uid = %s",
                (uid, )).one()
57
58      def list(self, path):
```

```
59          if path == "" or path == '%':
60              return self.session.execute("select * from object").all()
61          else:
62              return self.session.execute("select * from object where path
                    like %s", (path, )).all()
63
64      def getDataServers(self, onlyOnline=True):
65          if onlyOnline:
66              return self.session.execute("select * from dataserver where
                    online=true allow filtering").all()
67          return self.session.execute("select * from dataserver").all()
68
69      def updateDataServerStatus(self, addr, online, remaining_capacity=0,
            capacity=0, available_down=.0, available_up=.0):
70          self.session.execute("update dataserver "
71                               "set capacity=%s, remaining_capacity=%s,
                                    available_down=%s, "
72                               "available_up=%s, online=%s where server=%s",
73                               (capacity, remaining_capacity,
74                                available_down, available_up, online, addr))
75          self.session.execute("insert into performance_log(server, time,
                capacity, remaining_capacity, available_down,"
76                               "available_up, online) values (%s, %s, %s, %s,
                                    %s, %s, %s)",
77                               (addr, datetime.now(), capacity,
                                    remaining_capacity,
78                                available_up, available_down, online))
79
80      def removeStoredObject(self, uid, server):
81          uid = makeUUID(uid)
82          self.session.execute("delete from stored_object where uid = %s and
                server = %s", (uid, server))
83
84      def removeObject(self, uid):
85          uid = makeUUID(uid)
86          self.session.execute("delete from object where uid = %s", (uid, ))
87
88      def getServersForUid(self, uid, complete=True, online=True):
89          uid = makeUUID(uid)
90          if complete:
91              res = self.session.execute(
92                  "select server from stored_object where uid = %s and
                        complete = true allow filtering", (uid, )).all()
93          else:
94              res = self.session.execute(
95                  "select server from stored_object where uid = %s allow
                        filtering", (uid, )).all()
96          if online:
97              def f(srv):
98                  return self.session.execute("select online from dataserver
                        where server = %s", (srv.server, )).one().online
99              res = list(filter(f, res))
100         print("getServersForUid", uid, res)
```

```
101             return res
102
103         def isServerOnline(self, server):
104             return self.session.execute("select online from dataserver where
                    server = %s", (server, )).one().online
105
106         def setComplete(self, uid, server):
107             uid = makeUUID(uid)
108             self.session.execute("update stored_object set complete = true
                    where uid = %s and server = %s", (uid, server))
109
110         def markDeleted(self, path):
111             res = self.getUidForPath(path)
112             timestamp = datetime.now()
113             if res is not None:
114                 self.session.execute("update object set deleted = %s where uid
                        = %s",
115                                      (timestamp, res.uid))
116
117         def markUidDeleted(self, uid):
118             uid = makeUUID(uid)
119             timestamp = datetime.now()
120             self.session.execute("update object set deleted = %s where uid = %s
                    ",
121                                  (timestamp, uid))
122
123         def getUidForPath(self, path):
124             print("path", path)
125             path = str(path)
126             return self.session.execute("select * from pathToObject where path
                    = %s", (path, )).one()
127
128         def getUidsForPath(self, path):
129             print("path", path)
130             path = str(path)
131
132             return self.session.execute("select * from object where path like %
                    s", (path, )).all()
133
134         def getUidsForServer(self, server):
135             return self.session.execute("select uid from stored_object where
                    server = %s allow filtering", (server, ))
136
137         def addPendingUid(self, uid):
138             uid = makeUUID(uid)
139             self.session.execute("insert into pending_object(uid, enabled)
                    values (%s, True)", (uid, ))
140
141         def updatePriority(self, uid, priority):
142             uid = makeUUID(uid)
143             self.session.execute("update object set priority=%s where uid=%s",
                    (priority, uid))
144
```

```
145     def getPendingUids(self, onlyEnabled=True):
146         if onlyEnabled:
147             return self.session.execute("select uid from pending_object
                    where enabled=True allow filtering").all()
148         return self.session.execute("select uid from pending_object").all()
149 
150     def disablePendingUid(self, uid):
151         uid = makeUUID(uid)
152         self.session.execute("update pending_object set enabled=False where
                uid = %s", (uid, ))
153 
154     def removePendingUid(self, uid):
155         uid = makeUUID(uid)
156         self.session.execute("delete from pending_object where uid = %s", (
                uid, ))
157 
158 
159 database = Database()
```

10.5.2 Dataserver

Struttura

```
1 
2 //un oggetto
3 create table object(
4     uid text primary key,
5     local_path text,    //percorso del file system dov'e' salvato
6     size integer,    //dimensione in byte (totale)
7     complete integer,     //true se e' completo, false se deve ancora essere
          totalmente trasferito
8     created text,        //istante di creazione (nel dataserver)
9     checksum text
10 );
11 
12 //configurazione del dataserver
13 create table stats(
14     storage int,         //storage totale in byte
15     downspeed int,       //banda totale in download, in Mb/s
16     upspeed int          //banda totale in upload, in Mb/s
17 );
18 
19 //documenti che vanno eliminati
20 create table toBeDeleted(
21     uid text primary key         //uid del documento
22 );
```

Codice

```
1 class Database:
2     def __init__(self):
3         try:
```

```
4               self.connection = sqlite3.connect('database.db')
5               self.cursor = self.connection.cursor()
6               print("connected to database")
7
8           except sqlite3.Error as error:
9               print("error while connecting to database", error)
10
11      def getObject(self, uid):
12          print("getObject ------>", (str(uid), ))
13          return self.cursor.execute("select * from object where uid = ?", (
                str(uid), )).fetchone()
14
15
16      def deleteUid(self, uid):
17          self.cursor.execute("delete from object where uid = ?", (uid, ))
18          self.cursor.execute("delete from transfer where uid = ?", (uid, ))
19          self.cursor.execute("delete from toBeDeleted where uid = ?", (uid,
                ))
20          self.connection.commit()
21
22      def nodeStats(self):
23          return self.cursor.execute("select * from stats").fetchone()
24
25      def getStoredData(self):
26          return self.cursor.execute("select uid, created, complete from
                object").fetchall()
27
28      def reservedSpace(self):
29          res = self.cursor.execute("select sum(size) from object").fetchone
                ()[0]
30          if res is None:
31              res = 0
32          return res
33
34      def addToBeDeleted(self, uid):
35          self.cursor.execute("insert into toBeDeleted(uid) values (?)", (uid
                , ))
36          self.connection.commit()
37
38      def getToBeDeleted(self):
39          return self.cursor.execute("select * from toBeDeleted").fetchall()
40
41      def addObject(self, uid, local_path, size, checksum):
42          complete = 0
43          created = datetime.now()
44
45          self.cursor.execute("insert into object(uid, local_path, size,
                complete, checksum, created) values (?, ?, ?, ?, ?, ?)",
46                              (uid, local_path, size, complete, checksum,
                                    created))
47          self.connection.commit()
48
49          return self.cursor.lastrowid
```

```
50
51      def setComplete(self, uid):
52          self.cursor.execute("update object set complete=1 where uid = ?", (
                uid, ))
53          self.connection.commit()
54
55      def setStats(self, storage, downspeed, upspeed):
56          self.cursor.execute("delete from stats")
57          self.cursor.execute("insert into stats(storage, downspeed, upspeed)
                values (?, ?, ?)",
58                              (storage, downspeed, upspeed))
59          self.connection.commit()
60
61  database = Database()
```

10.6 Operazioni periodiche

Le operazioni che non possono essere eseguite subito (es: eliminazione di un documento mentre sta venendo trasferito) oppure che vanno eseguite regolarmente (es: monitoraggio delle prestazioni del dataserver) sono valutate *periodicamente* ed eseguite nel primo momento utile.

Lo schema di esecuzione è molto semplice: ogni operazione periodica è *schedulata ad intervalli regolari* (es 10 secondi) e viene eseguita da un *looper thread*.

10.6.1 Metaserver

Nel metaserver le operazioni periodiche sono:

- Controllo dello stato dei dataserver
- Mantenimento del grado di replicazione dei documenti

Controllo dello stato dei dataserver

```
1   def onDataServerConnect(addr):
2       print("server", addr, "connected")
3
4       for elem in database.getUidsForServer(addr):
5           database.addPendingUid(elem.uid)
6
7       for elem in database.getPendingUids(onlyEnabled=False):
8           database.addPendingUid(elem.uid)
9
10  def onDataServerDisconnect(addr):
11      if database.isServerOnline(addr):
12          database.updateDataServerStatus(addr, False)
13
14          for elem in database.getUidsForServer(addr):
15              database.addPendingUid(elem.uid)
```

```
16
17          print("server", addr, "disconnected")
18
19
20  def checkDataServerStatus(addr):
21      wasOnline = database.isServerOnline(addr)
22
23      try:
24          with Connection(addr) as conn:
25              conn.write("status")
26              res = conn.readline()
27
28          status, reservedCapacity, totCapacity, downSpeed, bandDown, upspeed
                , bandUp = res
29          reservedCapacity = int(reservedCapacity)
30          totCapacity = int(totCapacity)
31          downSpeed = float(downSpeed)
32          bandDown = float(bandDown)
33          upspeed = float(upspeed)
34          bandUp = float(bandUp)
35          database.updateDataServerStatus(addr, True, totCapacity -
                reservedCapacity, totCapacity,
36                                          downSpeed - bandDown, upspeed -
                                                bandUp)
37
38          if not wasOnline:
39              onDataServerConnect(addr)
40      except ConnectionRefusedError as err:
41          print(err)
42          if wasOnline:
43              onDataServerDisconnect(addr)
44
45  def monitorDataServers():
46      for server in database.getDataServers(onlyOnline=False):
47          checkDataServerStatus(server.server)
```

Mantenimento del grado di replicazione dei documenti

```
1   def processPendingUids():
2       for elem in database.getPendingUids():
3           processPendingUid(database, elem.uid)
4
5   def processPendingUid(database, uid):
6       res = database.getObjectByUid(uid)
7       print(res)
8       priority = res.priority
9       size = res.size
10      checksum = res.checksum
11      print("priority", priority)
12
13      serversContaining = {x.server for x in database.getServersForUid(uid)}
14      numCopies = len(serversContaining)
15
```

```
16      print("serversContaining", serversContaining)
17      availableServers = [x.server for x in database.getDataServers() if x.
            server not in serversContaining
18                          and x.remaining_capacity > size]
19
20      print("availableServers", availableServers)
21
22      serversContaining = list(serversContaining)
23
24      if numCopies == priority:
25          database.removePendingUid(uid)
26          return
27
28      if numCopies > priority:
29          random.shuffle(serversContaining)
30          target = serversContaining[0]
31
32          print("remove", uid, "target", target)
33
34          try:
35              with Connection(target) as conn:
36                  conn.write("deleteUid", uid)
37                  res = conn.readline()
38                  print(res)
39
40              database.removeStoredObject(uid, target)
41          except:
42              pass
43
44          return
45
46      if numCopies < priority:
47
48          if len(serversContaining) == 0:
49              database.disablePendingUid(uid)
50              return
51
52          if len(availableServers) > 0:
53              random.shuffle(availableServers)
54              random.shuffle(serversContaining)
55
56              try:
57                  source = serversContaining[0]
58                  target = availableServers[0]
59
60                  database.addStoredObject(uid, target)
61
62                  def process():
63                      try:
64                          with Connection(target) as conn:
65                              conn.write("createUid", uid, size, checksum)
66                              print(conn.readline())
67
```

```
68                              with Connection(source) as conn:
69                                  conn.write("transfer", uid, target)
70                                  print(conn.readline())
71                          except:
72                              database.removeStoredObject(uid, target)
73                              database.addPendingUid(uid)
74
75                      _thread.start_new_thread(process)
76
77                      #database.removePendingUid(uid)
78                  except:
79                      print("ERROR")
80              else:
81                  database.disablePendingUid(uid)
```

Scheduling ed esecuzione

```
1  schedule.every(5).seconds.do(monitorDataServers)
2  schedule.every(5).seconds.do(processPendingUids)
3  schedule.run_all()
4
5  def repeatedActions(_):
6      while True:
7          schedule.run_pending()
8          time.sleep(0.1)
9
10 _thread.start_new_thread(repeatedActions, (None,))
```

10.6.2 Dataserver

Nel dataserver le operazioni periodiche sono:

- Controllo dello *stato interno* (utilizzo della banda e della memoria di massa)
- Eliminazione dei documenti in coda

Controllo dello stato interno

```
1  #bandup and banddown in MB/s
2  performances = {"sent": 0, "recv": 0, "lastTime" : 0, "bandup" : 0, "
       banddown" : 0}
3
4  def getPerformance():
5      res = psutil.net_io_counters()
6      curtime = time.time()
7      delta = curtime - performances["lastTime"]
8      performances["lastTime"] = curtime
9      performances["bandup"] = (res.bytes_sent - performances["sent"]) /
           delta / 1000000
10     performances["banddown"] = (res.bytes_recv - performances["recv"]) /
           delta / 1000000
11     performances["sent"] = res.bytes_sent
```

```
12      performances["recv"] = res.bytes_recv
13      #print(performances)
```

Per tenere traccia dei documenti attualmente in trasferimento, che quindi non possono essere eliminati, viene utilizzato il set *processing*. Esso contiene in ogni momento tutti gli *uid* relativi a documenti in caricamento e in trasferimento.

Eliminazione dei documenti in coda

```
1  def processDeleteUids():
2      logging.info("processDeleteUids...")
3
4      database = Database()
5      uids = database.getToBeDeleted()
6
7      with processingLock:
8          logging.info("uids %s", uids)
9
10         for uid, in uids:
11             if uid not in processing:
12                 logging.info("delete %s", uid)
13                 uid, localPath, size, complete, created, checksum =
                       database.getObject(uid)
14                 if os.path.exists(localPath): os.remove(localPath)
15                 database.deleteUid(uid)
```

Scheduling ed esecuzione

```
1  schedule.every(5).seconds.do(getPerformance)
2  schedule.every(15).seconds.do(processDeleteUids)
3
4  def runPeriodicTasks(_):
5      while True:
6          schedule.run_pending()
7          time.sleep(0.1)
8
9  _thread.start_new_thread(runPeriodicTasks, (None,))
```

10.7 ServerSocket e relativo handler

Il punto di contatto di metaserver è il metodo *handle*. Esso si occupa del *dispatching delle richieste*, con l'ausilio di una mappa delle operazioni. Si ricorda che, secondo il protocollo di comunicazione, una richiesta è formata dal nome ed eventuali parametri, separati dal punto e virgola e terminati da newline. Una soluzione analoga è presente per il dataserver.

Una eventuale *richiesta non riconosciuta* viene automaticamente scartata.

Metaserver e relativo handler

```
    ...

    def handle(self):
        self.database = Database()

        switcher = {
            "getPath": self.getPath,
            "pushPath": self.pushPath,
            "list": self.list,
            "pushComplete": self.pushComplete,
            "test": self.test,
            "deletePath": self.deletePath,
            "addDataServer": self.addDataServer,
            "getUid": self.getUid,
            "lockPath": self.lockPath,
            "getPathLock": self.getPathLock,
            "unlockPath": self.unlockPath,
            "updatePriorityForPath": self.updatePriorityForPath,
            "updatePriorityForUid": self.updatePriorityForUid,
            "permanentlyDeletePath": self.permanentlyDeletePath
        }

        print("handle request from " + str(self.client_address))

        data = self.readline()
        print(data)

        switcher[data[0]](data[1:])

with MetaServer((HOST, PORT), MetaServerHandler, bind_and_activate=True) as
    server:
    server.serve_forever()
```

In questa sezione vengono analizzate le implementazioni di tutti i metodi/operazioni che possono essere eseguite. L'analisi è suddivisa per operazioni, in modo da poter apprezzare meglio le interazioni tra i componenti.

10.7.1 Caricamento di un documento

Questa azione viene usata dal client per caricare un documento.

Inizialmente il client si collega al dataserver inviando *path, dimensione, checksum, priorità, utente.*

Client

```
def sendFile(localPath = "small_file.txt", remotePath="testfile.txt",
    priority=1, user="default"):
    with Connection(METASERVER) as conn:
```

```
3           size = getsize(localPath)
4           print("file size", size)
5
6           checksum = hashFile(localPath)
7
8           conn.write("pushPath", remotePath, size, checksum, priority, user)
9           res = conn.readline()
10
11          if res[0] != "ok":
12              print("ERR", res)
13              return False
14
15          state, uid, addr = res
16      ...
```

Il metaserver controlla che il path non sia bloccato da un altro utente, cerca un dataserver che possa ospitare il documento e gli assegna un nuovo uid.

Metaserver

```
1   def pushPath(self, args):
2       path, size, checksum, priority, user = args
3       size = int(size)
4
5       lock, lockUser = self.database.getPathLock(path)
6       if lock and lockUser != user:
7           self.write("err", "path locked by " + lockUser)
8           return False
9
10      dataservers = [x for x in self.database.getDataServers() if x.
            remaining_capacity > size]
11
12      if len(dataservers) == 0:
13          self.write("err", "no data servers available")
14          return False
15
16      random.shuffle(dataservers)
17      target = dataservers[0]
18
19      uid = uuid4()
20      addr = target.server
21      ...
```

Il metaserver contatta il dataserver di destinazione creando la risorsa.

Metaserver

```
1       ...
2       with Connection(addr) as dataServer:
3           dataServer.write("createUid", uid, size, checksum)
4           response = dataServer.readline()
5
```

```
6       if response[0] != "ok":
7           print("ERROR", response[0])
8           self.write("err", response)
9           return False
10      ...
```

Dataserver

```
1   def createUid(self, args):
2       uid, size, checksum = args
3       local_path = os.getcwd() + "/" + uid
4       print("local_path", local_path)
5       size = int(size)
6
7       with processingLock:
8           totCapacity, downSpeed, upspeed = self.database.nodeStats()
9           reservedCapacity = self.database.reservedSpace()
10
11          if reservedCapacity + size > totCapacity:
12              self.write("err", "storage capacity exceeded")
13              return False
14
15          self.database.addObject(uid, local_path, size, checksum)
16
17      self.write("ok")
```

Il metaserver marca come eliminato l'oggetto associato al path (se esiste) e aggiunge il nuovo oggetto al database. Risponde al client indicando *uid, addr*.

Metaserver

```
1       self.database.markDeleted(path)
2       self.database.addObject(uid, path, "root", size, priority, checksum)
3       self.database.addStoredObject(uid, addr)
4
5       self.write("ok", uid, addr)
```

Il client si collega al dataserver target e invia il documento. Se il documento era stato già parzialmente caricato nel dataserver (es. file di grosse dimensioni), il caricamento riprende da dover era rimasto. Se il documento era già stato caricato completamente (ovvero il client tenta di caricare un documento già completo), viene generato un errore. Se il checksum al termine del caricamento non corrisponde, il documento viene subito eliminato per essere ricaricato in futuro.

Client

```
1       ...
2       with Connection(addr) as conn:
3           conn.write("pushUid", uid)
4
```

```
5           res = conn.readline()
6           if res[0] != 'ok':
7               print(res)
8               return False
9
10          status, startIndex = res
11
12          print("send starting from", int(startIndex))
13
14          conn.writeFile(localPath, startIndex)
15          print(conn.readline())
```

Dataserver

```
1   def pushUid(self, args):
2       uid, = args
3
4       with processingLock:
5           objinfo = self.database.getObject(uid)
6
7           if objinfo is None:
8               self.write("ERR", "uid info not found")
9               return False
10
11          print(objinfo)
12
13          uid, localpath, size, complete, created, checksum,  = objinfo
14
15          if complete:
16              self.write("err", "file complete")
17              return False
18
19          processing.add(uid)
20
21      startIndex = 0
22      if os.path.exists(localpath):
23          startIndex = os.path.getsize(localpath)
24
25      assert startIndex < size   #altrimenti sarebbe complete
26
27      self.write("ok", startIndex)
28
29      with open(localpath, "a+b") as out:
30          self.readFile(size - int(startIndex), out)
31
32      file_checksum = hashFile(localpath)
33
34      with processingLock:
35          processing.remove(uid)
36
37          print("checksum", checksum, "file_checksum", file_checksum)
38
39          if file_checksum != checksum:
```

```
40                os.remove(localpath)
41                print("ERROR checksum do not match")
42                self.write("err", "checksum do not match")
43                return False
44
45            self.database.setComplete(uid)
```

Il dataserver invia al metaserver la notifica che il caricamento è completo, e quindi il documento può essere utilizzato.

Dataserver

```
1       ...
2       with Connection(METASERVER) as metaConn:
3           metaConn.write("pushComplete", uid, SERVER)
4
5       self.write("ok")
```

Il metaserver processa l'elemento per controllarne il grado di replicazione.

Metaserver

```
1   def pushComplete(self, args):
2       uid, addr = args
3
4       self.database.setComplete(uid, addr)
5       self.database.addPendingUid(uid)
6
7       self.write("ok")
```

10.7.2 Trasferimento di un documento

Il trasferimento viene usato dal metaserver per copiare un documento da un dataserver ad un altro, aumentandone così il grado di replicazione.

Il metaserver prima configura il server di destinazione per ricevere il documento, come nel caso del caricamento da parte del client. Poi invia la richiesta di trasferimento al server sorgente specificando *uid, indirizzo server destinazione*.

Metaserver

```
1   with Connection(target) as conn:
2       conn.write("createUid", uid, size, checksum)
3       print(conn.readline())
4
5   with Connection(source) as conn:
6       conn.write("transfer", uid, target)
7       print(conn.readline())
```

Il caricamento del documento dal server sorgente a destinazione avviene con lo stesso metodo usato nel caricamento da client a dataserver.

Dataserver sorgente

```
1  def transfer(self, args):
2      uid, server = args
3
4      with processingLock:
5          res = self.database.getObject(uid)
6          if res is None:
7              self.write("err", "uid not found")
8              return False
9
10         processing.add(uid)
11
12     uid, localPath, size, complete, created, checksum = res
13
14     with Connection(server) as target:
15         target.write("pushUid", uid)
16         status, startIndex = target.readline()
17         target.writeFile(localPath, startIndex)
18
19     with processing:
20         processing.remove(uid)
21
22     self.write("ok")
```

10.7.3 Scaricamento sul client di un documento (dato il path)

Questa è l'operazione con cui il client ottiene un documento dal sistema di storage.

Inizialmente il client contatta il metaserver indicando il path che vuole scaricare.

Client

```
1  def get(localPath = "testin.txt", remotePath = "ale/file1", newFile=True,
       user="default"):
2      if newFile and os.path.exists(localPath):
3          os.remove(localPath)
4
5      with Connection(METASERVER) as conn:
6          conn.write("getPath", remotePath, user)
7          res = conn.readline()
8
9      print(res)
10
11     if res[0] != 'ok':
12         print("ERR", res)
13         return False
14
15     status, uid, addr, checksum = res
```

Il metaserver verifca che il path non sia bloccato da un altro utente, cerca l'uid associato al path (quello che corrisponde all'ultima versione del documento), cerca un dataserver che lo contenga (possibilmente seleziona il migliore) e risponde al client indicando *uid, indirizzo dataserver*.

Metaserver

```
1  def _getServerForUid(self, uid):
2      res = self.database.getServersForUid(uid)
3      print("servers", res)
4
5      if len(res) == 0:
6          self.write("err", "no copies available")
7          return False
8
9      random.shuffle(res)
10
11     addr = res[0].server
12     return addr
13
14 def getPath(self, args):
15     path, user = args
16
17     lock, lockUser = self.database.getPathLock(path)
18     if lock and lockUser != user:
19         self.write("err", "path locked by " + lockUser)
20         return False
21
22     res = self.database.getUidForPath(path)
23
24     if not res:
25         self.write("err", "path not found")
26         return False
27
28     if res.deleted is not None:
29         self.write("err", "path deleted")
30         return False
31
32     uid, checksum = res.uid, res.checksum
33     print("uid", uid)
34
35     addr = self._getServerForUid(uid)
36
37     self.write("ok", uid, addr, checksum)
```

A quel punto il client si collega al dataserver sorgente, invia la posizione da cui vuole iniziare a leggere il documento (usato per riprendere il trasferimento interrotto di un documento di grandi dimensioni) e legge il documento. La chiamata usata in questo caso è *getUid*. Al termine viene verificato il checksum del documento per individuare eventuali errori o manomissioni.

Client

```
    startIndex = 0
    if os.path.exists(localPath):
        startIndex = os.path.getsize(localPath)

    with Connection(addr) as conn:
        conn.write("getUid", uid, startIndex)
        res = conn.readline()
        print(res)
        status, size = res

        with open(localPath, "a+b") as out:
            conn.readFile(size, out)

    if checksum != hashFile(localPath):
        logging.error("HASH don't match")
        return False

    return True
```

Codice

```
def getUid(self, args):
    uid, startIndex = args
    startIndex = int(startIndex)

    with processingLock:
        res = self.database.getObject(uid)

        if res is None:
            self.write("err", "specified uid not present")
            return False

        print(res)

        uid, localPath, size, complete, created, checksum = res

        if not complete:
            self.write("err", "not complete")
            return False

        processing.add(uid)

    self.write("ok", size-startIndex)
    self.writeFile(localPath, startIndex)

    with processingLock:
        processing.remove(uid)
```

10.7.4 Scaricamento di un documento dato l'uid

Nel caso in cui si voglia ottenere una particolare versione di un documento, si segue un meccanismo analogo al precedente. La differenza è che il client effettua una richiesta iniziale del tipo *getUid*, al posto che *getPath*, verso il metaserver.

Metaserver

```
1  def _getServerForUid(self, uid):
2      res = self.database.getServersForUid(uid)
3      print("servers", res)
4
5      if len(res) == 0:
6          self.write("err", "no copies available")
7          return False
8
9      random.shuffle(res)
10
11     addr = res[0].server
12     return addr
13
14 def getUid(self, args):
15     uid, user = args
16
17     res = self.database.getObjectByUid(uid)
18
19     if res is None:
20         self.write("err", "uid " + uid + " not found")
21         return False
22
23     checksum = res.checksum
24     addr = self._getServerForUid(uid)
25
26     self.write("ok", addr, checksum)
```

10.7.5 Eliminazione di un path

Questa operazione permette di eliminare logicamente un path. Tutti i documenti interessati vengono marcati come eliminati ma conservati, in modo che sia possibile reperirli tramite l'uid.

Il client effettua una richesta del tipo *deletePath* indicando *path, utente*. Remind: path può riferirsi ad un documento (es "doc1") o ad una repository (es "dir1%").

Client

```
1 def deletePath(path, user="default"):
2     conn = Connection(METASERVER)
3     conn.write("deletePath", path, user)
4     print(conn.readline())
5     conn.close()
```

Il metaserver ricerca tutti i documenti interessati e, se non sono bloccati, li marca come eliminati.

Metaserver

```
1  def deletePath(self, args):
2      path, user = args
3
4      res = self.database.getUidsForPath(path)
5
6      lockedPaths = []
7
8      for elem in res:
9          if elem.deleted is not None:
10             continue
11
12         lock, lockUser = self.database.getPathLock(elem.path)
13         if lock and lockUser != user:
14             lockedPaths.append(elem.path)
15         else:
16             uid = elem.uid
17             print("delete", uid)
18             if elem.deleted is None:
19                 self.database.markUidDeleted(uid)
20
21     if len(lockedPaths):
22         self.write("err", "paths locked :", lockedPaths)
23     else:
24         self.write("ok")
```

10.7.6 Eliminazione permanente di un path

L'eliminazione permanente di un path causa la cancellazione logica e fisica di tutti i dati relativi ad un path. In questo modo tutto lo spazio verrà liberato e non sarà più possibile recuperare i dati. Questa operazione è simile all'eliminazione di un'intera repository di un sistema di versionamento.

Il client invia una richiesta del tipo *permanentlyDeletePath* al metaserver indicando *path, utente.*

Codice

```
1  def permanentlyDeletePath(path, user="default"):
2      conn = Connection(METASERVER)
3      conn.write("permanentlyDeletePath", path, user)
4      print(conn.readline())
5      conn.close()
```

Come nel caso precedente, il metaserver verifica che i documenti non siano bloccati da altri utenti. In seguito marca ogni documento non bloccato come eliminato e gli setta

priorità nulla, di fatto schedulando l'eliminazione tutte le repliche.

Codice

```
def permanentlyDeletePath(self, args):
    path, user = args

    res = self.database.getUidsForPath(path)

    lockedPaths = []

    for elem in res:
        lock, lockUser = self.database.getPathLock(elem.path)
        if lock and lockUser != user and elem.priority > 0:
            lockedPaths.append(elem.path)
        else:
            uid = elem.uid
            print("delete", uid)
            if elem.deleted is None:
                self.database.markUidDeleted(uid)
            self.database.updatePriority(uid, 0)
            self.database.addPendingUid(uid)

    if len(lockedPaths):
        self.write("err", "paths locked :", lockedPaths)
    else:
        self.write("ok")
```

10.7.7 Lock

Mediante il meccanismo dei lock un client può "bloccare" un path (solo singolo documento). Quando un path è bloccato, tutte le richieste di altri utenti riguardanti quel path vengono respinte.

Per effettuare un lock, il client specifica il *path, utente.*

Client

```
def lockPath(path, user="default"):
    conn = Connection(METASERVER)
    conn.write("lockPath", path, user)
    res, = conn.readline()
    print("lockPath", path, "lock", res, "by", user)
    return res
```

Il metaserver risponde con True/False per indicare se il lock ha avuto successo. Fallisce quando si tenta di bloccare un path già bloccato da un altro utente.

Metaserver

```
def lockPath(self, args):
```

```
2       path, user = args
3       res = self.database.lockPath(path, user)
4       self.write(res)
```

Per verificare se un path sia bloccato si può effettuare una richiesta del tipo *getPathLock*. Essa restituisce un booleano e, nel caso sia bloccato, anche l'utente che detiene il blocco.

Metaserver

```
1  def getPathLock(self, args):
2      path, = args
3      lock, user = self.database.getPathLock(path)
4      self.write(lock, user)
```

Per sbloccare un path è sufficiente invocare la richiesta *unlockPath*. Essa non effettua controlli e si affida alla correttezza di comportamento dei client: questo per permettere a chiunque di sbloccare un path bloccato da un utente che è andato offline (al posto che lasciarlo bloccato per tempo arbitrario).

Metaserver

```
1  def unlockPath(self, args):
2      path, = args
3      self.database.unlockPath(path)
4      self.write("ok")
```

Gestire i lock su repository, al posto che singoli documenti, è tutt'altro che banale. Non avendo a disposizione all'interno di Cassandra l'operatore ILIKE, non ho trovato una soluzione efficiente per interrogare il database in modo da controllare se uno dei lock presenti sia prefisso del path che voglio controllare.

10.7.8 Modificare la priorità di un documento

È possibile in ogni momento modificare la priorità di un determinato path. Con *updatePriorityForPath* viene modificata la priorità di ogni documento relativo al path specificato, mentre con *updatePriorityForUid* è possibile effettuare l'operazione su una singola versione.

Il client contatta il metaserver indicando *path, utente, nuova priorità*. La nuova priorità deve essere >0.

Metaserver

```
1  def updatePriorityForPath(self, args):
2      path, priority, user = args
3      priority = int(priority)
4
```

```
5       if priority <= 0:
6           self.write("err", "priority must be >0")
7           return False
8
9       res = self.database.getUidsForPath(path)
10
11      lockedPaths = []
12
13      for elem in res:
14          if elem.priority <= 0:
15              continue
16
17          lock, lockUser = self.database.getPathLock(elem.path)
18          if lock and lockUser != user:
19              lockedPaths.append(elem.path)
20          else:
21              uid = elem.uid
22              print("delete", uid)
23              self.database.updatePriority(uid, priority)
24              self.database.addPendingUid(uid)
25
26      if len(lockedPaths):
27          self.write("err", "paths locked :", lockedPaths)
28      else:
29          self.write("ok")
30
31  def updatePriorityForUid(self, args):
32      uid, priority = args
33      priority = int(priority)
34
35      res = self.database.getObjectByUid(uid)
36
37      self.database.updatePriority(uid, priority)
38      self.database.addPendingUid(uid)
39
40      self.write("ok")
```

10.7.9 Aggiungere un dataserver

È possibile aggiungere un nuovo dataserver utilizzando la richiesta *addDataServer*, a cui va specificato l'indirizzo del dataserver.

Al momento della connessione il metaserver richiede al dataserver la lista dei contenuti in modo da aggiornare le proprie tabelle.

Metaserver

```
1   def addDataServer(self, args):
2       addr, = args
3
4       self.database.addDataServer(addr)
5
```

```
6       checkDataServerStatus(self.database, addr)
7
8       with Connection(addr) as conn:
9           conn.write("getStoredData")
10          len, = conn.readline()
11
12          for i in range(int(len)):
13              uid, created, complete = conn.readline()
14              created = dateutil.parser.parse(created)
15              complete = complete == "1"
16              self.database.addStoredObject(uid, addr, complete, created)
```

Dataserver

```
1   def getStoredData(self, args):
2       data = self.database.getStoredData()
3       self.write(len(data))
4       for uid, created, complete in data:
5           self.write(uid, created, complete)
```

11 Considerazioni finali

Perchè questo approccio è migliore rispetto ad usare direttamente un database distribuito?

L'utilizzo di un database distribuito è senza dubbio più semplice: basta installare il dbms nei nodi voluti e dialogarci tramite uno dei tanti client in commercio.

Ci sono però numerosi aspetti negativi:

- Non è possibile stabilire il livello di replicazione per un singolo documento. Un livello di replicazione unitario per file temporanei (es cache) riduce lo spreco di memoria, mentre un livello di replicazione elevato praticamente annulla la possibilità di perdita di dati di importanza inestimabile.
- Il dbms è generalmente dispendioso di risorse. Mentre la semplicità di un dataserver ne permette persino l'installazione in un dispositivo come Raspberry. E' così possibile sfruttare ad esempio un impianto IoT per immagazzinare dati.
- Il client si collega ad un nodo il quale inoltra i dati alla destinazione. Con questa architettura invece invia direttamente i dati al dataserver di destinazione (e viceversa). Si ha quindi un risparmio di banda complessiva.
- Il metaserver fornisce funzionalità quali mantenimento di un filesystem, operazioni di lock, bilanciamento del carico personalizzato.

Sviluppi futuri?

Alcuni sviluppi potrebbero essere la creazione di un client grafico, per l'utilizzo da parte del pubblico. Inoltre dovrebbe essere aggiunto un livello di protezione, sia dal punto di vista della cifratura che dei permessi (con eventuale login). Vanno effettuati test approfonditi e migliorata la gestione degli errori, soprattutto per quanto riguarda i corner case. Una compressione lato client permetterebbe un risparmio di risorse (memoria e rete) non indifferente.

Sistema di versionamento

È interessante notare che il sistema può benissimo essere utilizzato come sistema di versionamento. Dato che i documenti non vengono mai eliminati, ma la modifica presuppone la creazione di un nuovo uid (quindi vecchia e nuova versione sono effettivamente documenti a se stanti che condividono solo il path), è possibile andare a selezionare una particolare versione di ogni documento basandosi sulla data di creazione. L'unico accorgimento è quello di effettuare dal client una richiesta del tipo *getUid*, al posto di *getPath*.

La creazione di una repository, a differenza di *git*, è trasparente, in quanto ogni path è in sè una repository (oppure da un altro punto di vista l'intero filesystem è una grande repository). L'eliminazione di un documento tramite *deletePath* agisce soltanto sul fs, marcandolo come eliminato, ma non rimuove i dati dai dataserver. Quindi è sempre possibile recuperare i dati di un documento eliminato in questo modo. Invece *permanentlyDeletePath* elimina definitivamente un documento (o repository), permettendo la liberazione di spazio.

Cosa ho imparato con questo progetto?

Ho affinato la mia conoscenza del linguaggio Python 3. In particolare ho scoperto come utilizzare il pattern decorator per arricchire oggetti "di sistema" in modo da strutturare in modo efficace il codice.

Ho usato per la prima volta un database NoSQL, in particolare Apache Cassandra. Essendoci molti punti a favore, oltre che molte limitazioni, rispetto ai "classici" database SQL, non è stato facile capire come sfruttarne al meglio le potenzialità. Solo per citare alcuni esempi, operazioni come la generazione delle chiavi primarie ed effettuare filtri/join sono tutt'altro che scontate.

La gestione di un sistema distribuito conivolge, per sua natura, il dialogo di più programmi. Va posta particolare cura allo sviluppo dei protocolli di comunicazione, sincronizzazione e fault tolerance, in un ambiente in cui errori ed ritardi (sia interni che esterni al sistema) sono la norma. Tutte situazioni che in un sistema unitario non si presentano.

L'imprevedibilità elevata riduce drasticamente l'efficacia del testing. E' quindi ancora più fondamentale stabilire contratti, vincoli, invarianti, diagrammi di flusso, meccanismi di controllo, meccanismi di logging e criteri di scrittura del codice che aiutino a minimizzare sia la probabilità di errore che i danni a seguito di un errore.

Bibliography