# Tesina Distributed FS (main)

alessiocorrado99

September 2020

## 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 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 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 a seguito di problemi.

**Numero di repliche** Il numero di repliche di un file deve essere proporzionale sia all'importanza che al numero di accessi di esso: file 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 i file vengono copiati o spostati in modo da non sovraccaricare un singolo nodo. Inoltre, per il trasferimento di un file 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...).

.

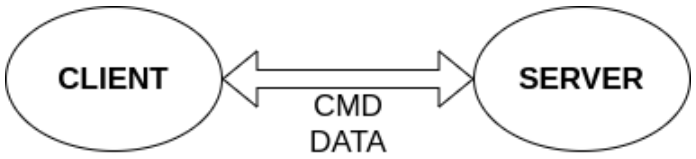



Figure 1: Single-server

## 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.

**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*

**Svantaggi**

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

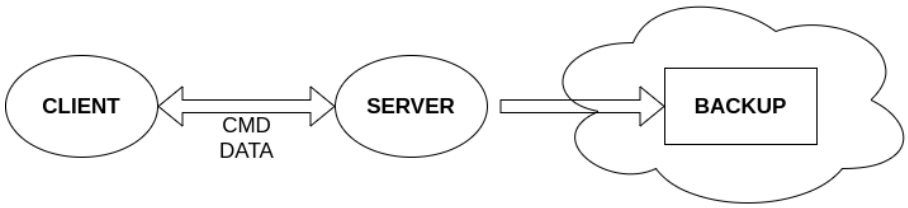



Figure 3: Aggiunta del backup nel cloud

## 2.4 Server as distributed database

Al posto di utilizzare il file system 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 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 può essere doppio rispetto al necessario: nel caso i dati vadano trasferiti dal client al server (gateway) al server di destinazione o vice versa.

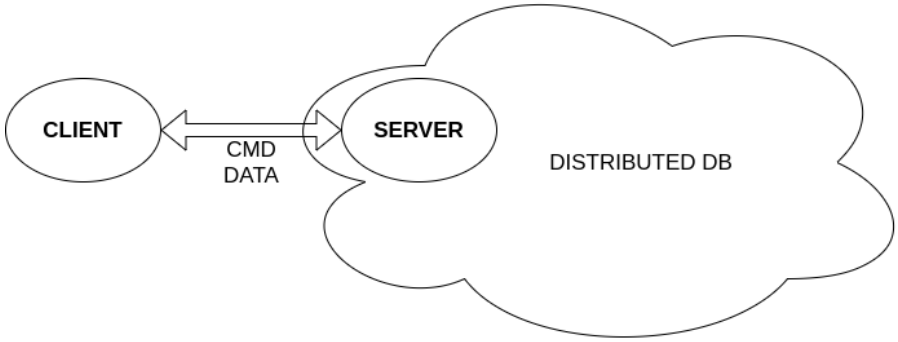



Figure 4: Utilizzo di un database distribuito

## 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 file system (metadati), mentre i data server memorizzano soltanto i dati conenuti nei file.

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 file 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
- Un *unico meta server* permette di mantenere in modo semplice la consistenza del file system

**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)

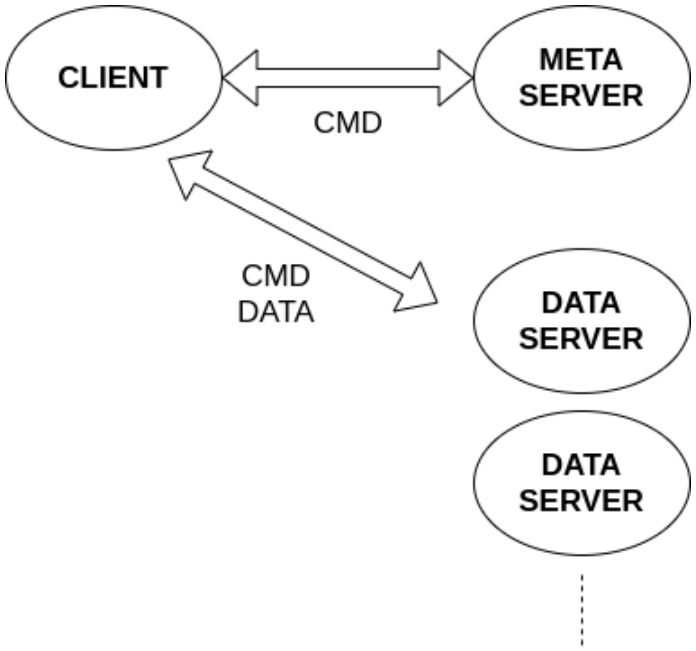



Figure 5: Separazione di meta e data server

## 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*, 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

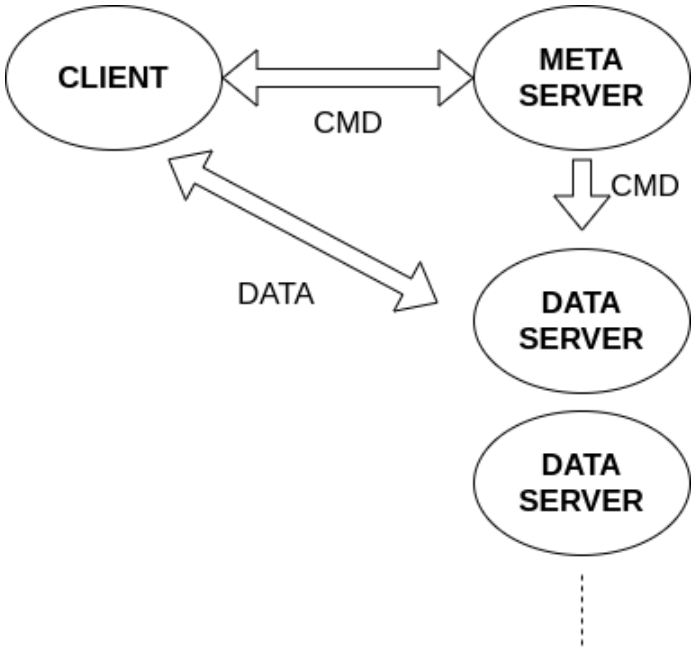



Figure 6: Meta + data server, il meta server dialoga con i data server

## 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.

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

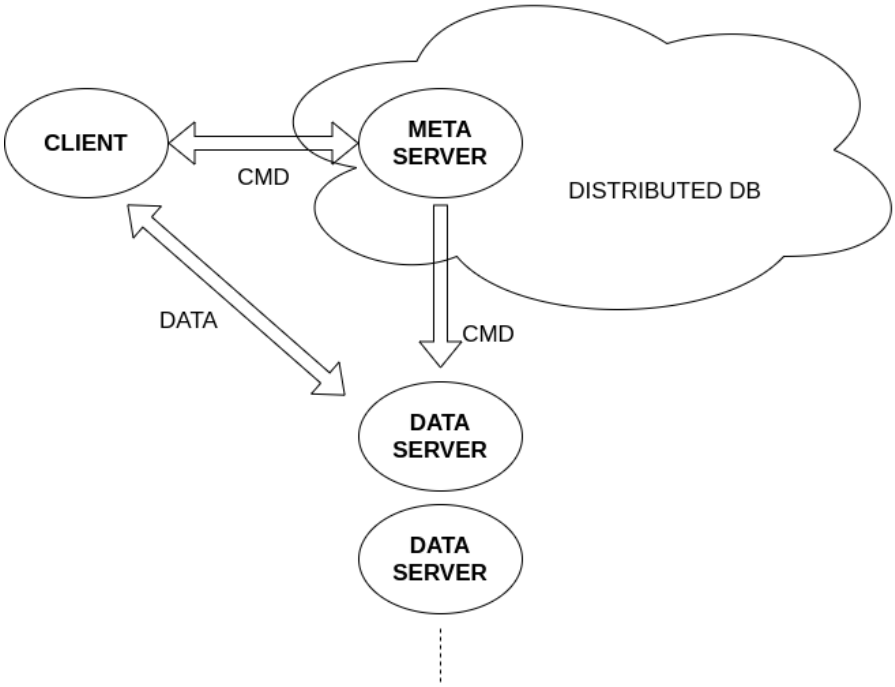



Figure 7: Il meta server è implementato come database distribuito

# 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 (associabili a file). 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 il simbolo '/'). Un path può essere associato ad uno delle tre tipologie sopra menzionate; il filesystem memorizza questa associazione oltre che agli 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 un documento con le informazioni sul sistema (es proprietario, descrizione, contatti).

### 3.2 Metadati di un path

I metadati associati ad un path sono:

- uid: identificatore univoco dell'oggetto
- path: deve rappresentare un documento
- created: timestamp di creazione
- size: dimensione in byte, 0 per le directory
- owner: utente proprietario, default chi l'ha creato
- priority: numero minimo di copie, coincide con la priorità
- checksum: checksum sha1
- deleted: timestamp di eliminazione (vuoto se non eliminato)

## 4 Comandi del 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 documenti attualmente in una repository)
- lock: lock e unlock di un path (documento o documenti attualmente in una repository)
- setPriority: aggiornare la priorità di un path (documento o documenti attualmente in una repository)

# 5 Protocollo comunicazione

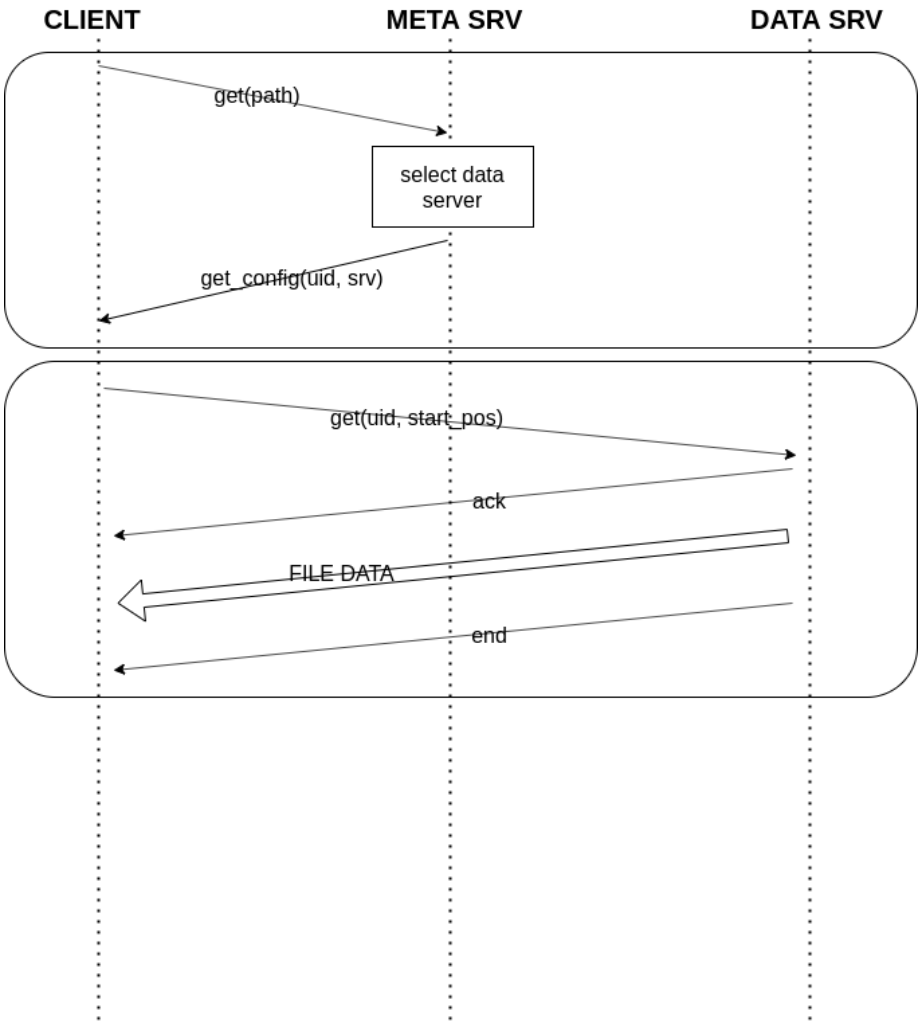
Di seguito i flussi di dati coinvolti nelle diverse tipologie di richieste.

## 5.1 get

La richiesta di tipo *get(path)* richiede il trasferimento in entrata di un file.

Inizialmente il *client* contatta il *meta server*, richiedendo un particolare *path*. 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 file (indice 0-based). Se la risorsa è disponibile (come dovrebbe essere) il *data server* invia un *ack* seguito dal flusso di dati del file. Altrimenti risponde con *err* seguito dai dettagli.




## 5.2 push

La richiesta di tipo *push(path, data)* richiede il trasferimento in uscita di un file da parte del client.

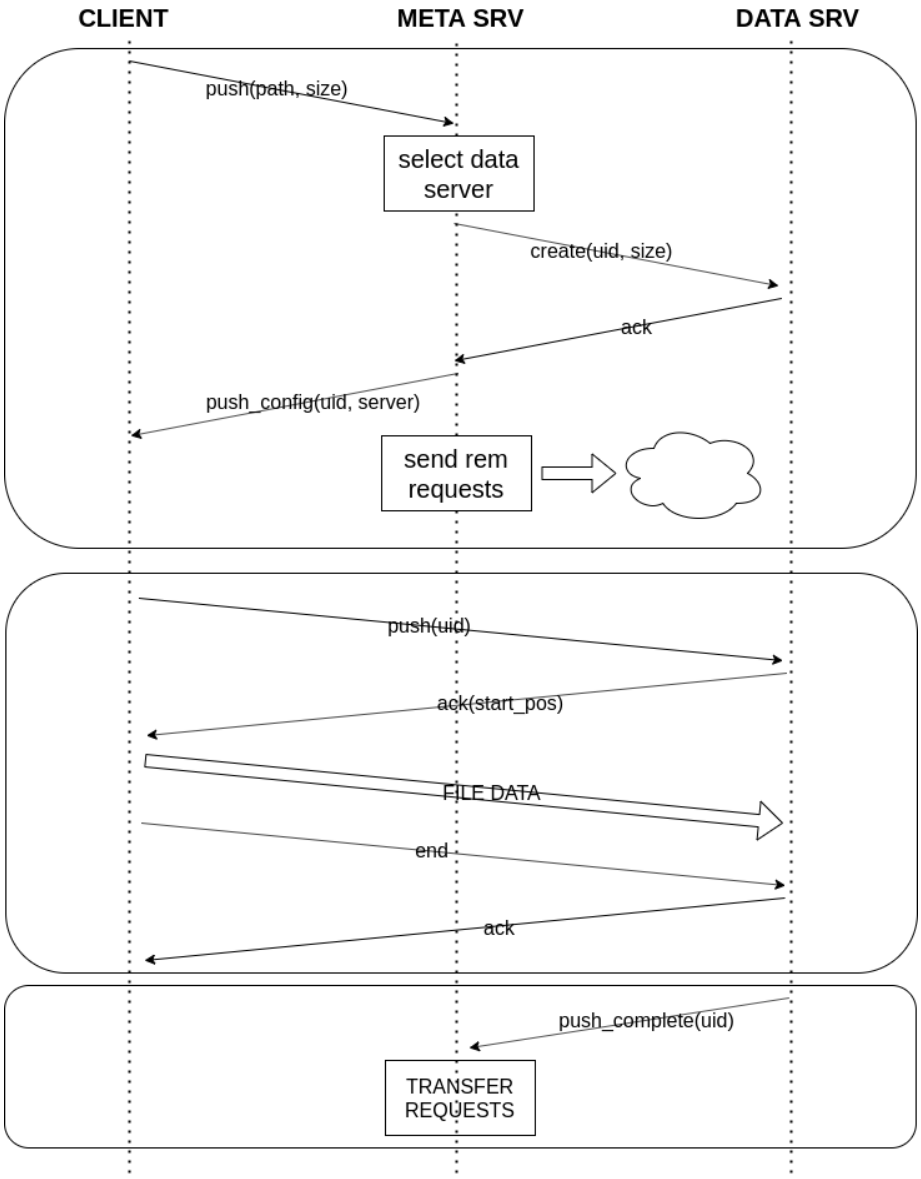
Inizialmente il *client* contatta il *meta server*, inviando *path* e *size*. Il *meta server* seleziona il *data server* ottimale per gestire la richiesta (deve essere online, poter contenere la risorsa e non essere sovraccarico). Il *meta server* si occupa anche di generare un nuovo *uid* da assegnare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

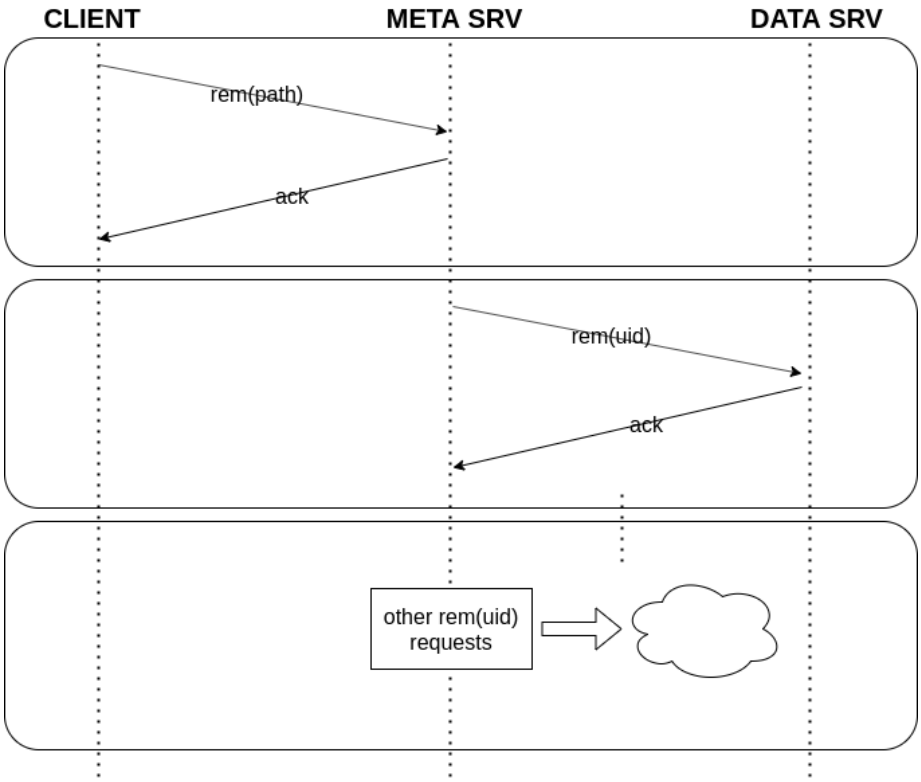



## 5.3 rem

La richiesta di tipo *rem(path)* richiede l'eliminazione di un file/directory dallo storage.

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

Il *meta server* contatta separatamente tutti i *data server* contenenti dati relativi a quel particolare *uid* e invia una richiesta *rem(uid)*.

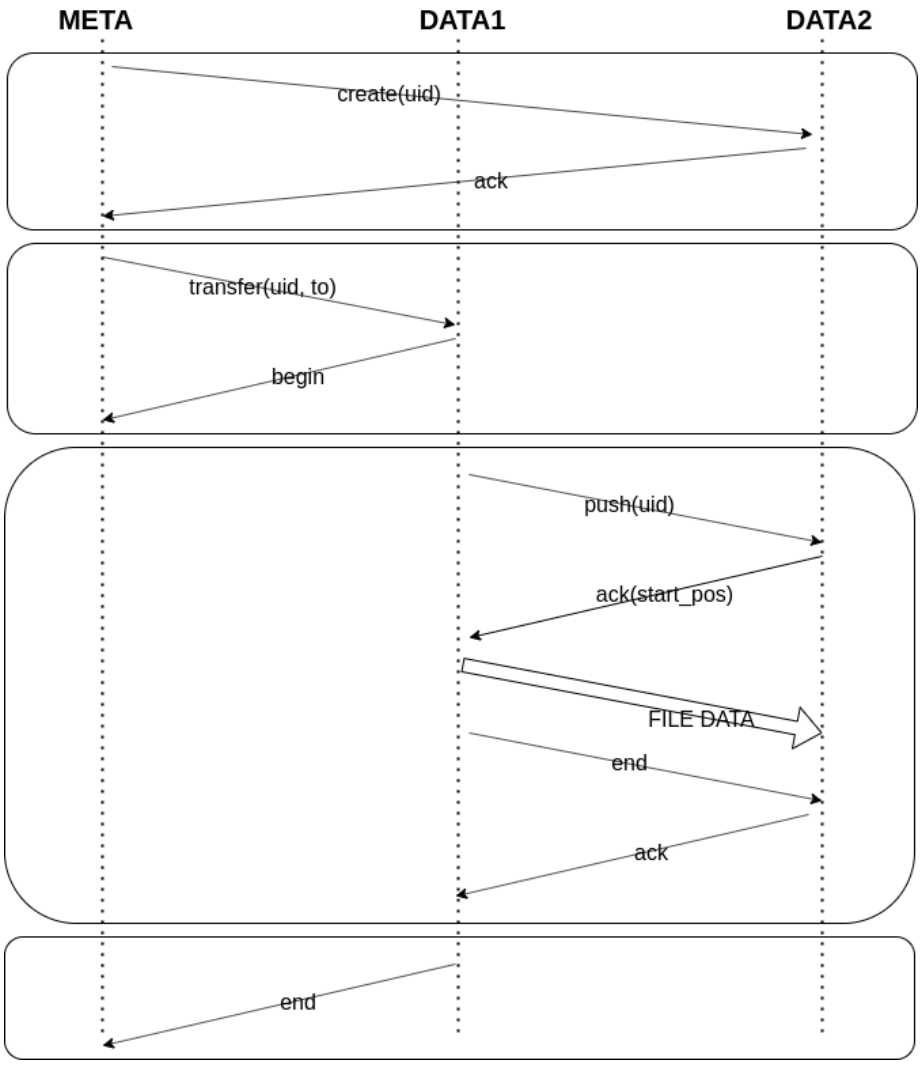



## 5.4 transfer

La richiesta di tipo *transfer* richiede la copia di un file (uid) da un *data server* ad un altro. Viene usata dal *meta server* per trasferire i dati tra i nodi e 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 file. 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.

# 7 Sicurezza ed autenticazione

Tutte le connessioni vengono cifrate 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 coppia *(username, password)* 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 file 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, dato che non esisterebbe più una funzione che va dal dominio delle chiavi a quello delle risorse. 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

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.

Non potendo garantire che tutti i nodi mondiali (quindi appartenenti ad organizzazioni diverse, che non si conoscono), una delle soluzioni maggiormente impiegate consiste nell'utilizzo di UUID. Gli UUID (Universally Unique Identifier) sono stringhe binarie di 128 cifre generate con precise regole, dipendenti dalla versione. Vengono usati generatori di numeri pseudocasuali (con distribuzione uniforme) e funzioni di hash per creare valori apparentemente casuali. Sebbene l'impiego di UUID non garantisca la certezza di generare un valore non visto in precedenza, per avere il 50% di probabilità di avere almeno una collisione è necessario generare $2.71 * 10^{18}$ valori.

Quando si lavora con UUID ci sono due grandi filosofie: controllare che il valore generato effettivamente non sia già presente o non fare nulla, contando sul fatto che la collisione non si verifichi. La prima soluzione è più dispendiosa e non sempre più essere implementata in modo efficace, la seconda necessita di meccanismi di recovery che permettono di recuperare il sistema nel caso una collisione si sia verificata.

# 9 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.

## 9.1 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 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.

## 9.2 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.

## 9.3 Caricamento della configurazione

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

### 9.3.1 Metaserver

Codice

```
1  if len(sys.argv) > 1:
2      configFile = sys.argv[1]
3      print("load config", configFile)
4      config = yaml.full_load(open(configFile, "r"))
5      print("config", config)
6  else:
7      logging.error("Please provide config file")
8      exit(1)
9
10 database = Database()
11
12 for dataserver in config["dataservers"]:
13     database.addDataServer(dataserver)
14     print("add dataserver", dataserver)
```

Esempio di configurazione

```
1  dataservers:
2      - localhost:10010
3      - localhost:10011
4      - localhost:10012
```

### 9.3.2 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:
        with open("../../dataserver/create_db.sqlite3", "r") as sql:
            conn.executescript(sql.read())

database = Database()
database.setStats(config["storage"], config["downspeed"], config["upspeed"])

SERVER = str(HOST) + ":" + str(PORT)
```

Esempio di configurazione

```
name: dataserver10012
workingDir: /home/ale/Dropbox/tesina iot/code/data/dataserver10012
host: localhost
port: 10012
storage: 100000000
downspeed: 50
upspeed: 10
metaserver: localhost:10000
```

## 9.4 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.

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.

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

```
def CustomConnection(cls):
    def addrFromString(self, addr):
        ip, port = addr.split(":")
        return (ip, int(port))

    def write(self, *args):
        res = ";".join([str(i) for i in args]).strip() + "\n"
        logging.info("write %s", res)
        self.wfile.write(res.encode())

    def readline(self):
        line = self.rfile.readline().strip().decode().split(";")
        logging.info("readline %s", line)
        return line

    def readFile(self, size, outFile):
        size = int(size)
        pos = 0
        while pos != size:
            chunk = min(1024, size - pos)
            # print("read", chunk, "bytes")
            data = self.rfile.read(chunk)
            outFile.write(data)
            pos += len(data)

    def writeFile(self, filepath, startIndex):
        if type(startIndex) != int:
            startIndex = int(startIndex)
        with open(filepath, "rb") as file:
            self.sendfile(file, startIndex)

    setattr(cls, "writeFile", writeFile)
    setattr(cls, "addrFromString", addrFromString)
    setattr(cls, "readFile", readFile)
    setattr(cls, "write", write)
    setattr(cls, "readline", readline)

    return cls


@CustomConnection
class Connection(socket.socket):
    def __init__(self, endpoint):
        super(Connection, self).__init__(socket.AF_INET, socket.SOCK_STREAM)
        self.connect(self.addrFromString(endpoint))
        self.wfile = self.makefile('wb', 0)
        self.rfile = self.makefile('rb', -1)

```

```
49
50 @CustomConnection
51 class DataServerHandler(StreamRequestHandler):
52     def sendfile(self, *args, **kwargs):
53         self.connection.sendfile(*args, **kwargs)
54
55     ...code...
```

## 9.5 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.

### 9.5.1 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, created)
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
        and created is not null
73 primary key (path, created, uid)
74 with clustering order by (created desc);
```

Codice

```
1 class Database:
2     def __init__(self):
3         self.cluster = Cluster(protocol_version=4)
4
5         self.session = self.cluster.connect("metaserver")
6
7
8     def addDataServer(self, addr):
```

```
9           self.session.execute("insert into dataserver(server, online) values (%s,
                false)", (addr, ))
10
11      def addObject(self, uid, path, owner, size, priority, checksum):
12          if type(uid) == "str":
13              uid = UUID(uid)
14          created = datetime.now()
15          size = int(size)
16          priority = int(priority)
17          self.session.execute("insert into object(uid, path, created, owner, size,
                priority, checksum) "
18                               "values (%s, %s, %s, %s, %s, %s, %s)"
19                               , (uid, path, created, owner, size, priority, checksum)
                                     )
20          return uid
21
22      def addStoredObject(self, uid, server, complete=False, created=None):
23          uid = makeUUID(uid)
24          if created is None: created = datetime.now()
25          self.session.execute("insert into stored_object(uid, server, created,
                complete) values (%s, %s, %s, %s)",
26                               (uid, server, created, complete))
27          return id
28
29      def lockPath(self, path, user="root"):
30          return self.session.execute("insert into object_lock(path, user) values (%s,
                 %s) if not exists",
31                                      (path, user)).one().applied
32
33      def unlockPath(self, path):
34          self.session.execute("delete from object_lock where path = %s", (path, ))
35
36      def getPathLock(self, path):
37          r = self.session.execute("select * from object_lock where path = %s", (path,
                 )).one()
38          if r is None:
39              return False, ""
40          else:
41              return True, r.user
42
43      def getObjectByUid(self, uid):
44          uid = makeUUID(uid)
45          return self.session.execute("select * from object where uid = %s", (uid, )).
                one()
46
47      def list(self, path):
48          if path == "":
49              return self.session.execute("select * from object").all()
50          else:
51              path = path + "%"
52              return self.session.execute("select * from object where path like %s", (
                    path, )).all()
53
54      def getDataServers(self, onlyOnline=True):
55          if onlyOnline:
56              return self.session.execute("select * from dataserver where online=true
                    allow filtering").all()
57          return self.session.execute("select * from dataserver").all()
```

```
58
59      def updateDataServerStatus(self, addr, online, remaining_capacity=0, capacity=0,
            available_down=.0, available_up=.0):
60          self.session.execute("update dataserver "
61                               "set capacity=%s, remaining_capacity=%s, available_down
                                     =%s, "
62                               "available_up=%s, online=%s where server=%s",
63                               (capacity, remaining_capacity,
64                                available_down, available_up, online, addr))
65          self.session.execute("insert into performance_log(server, time, capacity,
                remaining_capacity, available_down,"
66                               "available_up, online) values (%s, %s, %s, %s, %s, %s,
                                     %s)",
67                               (addr, datetime.now(), capacity, remaining_capacity,
68                                available_up, available_down, online))
69
70      def removeStoredObject(self, uid, server):
71          uid = makeUUID(uid)
72          self.session.execute("delete from stored_object where uid = %s and server =
                %s", (uid, server))
73
74      def removeObject(self, uid):
75          uid = makeUUID(uid)
76          self.session.execute("delete from object where uid = %s", (uid, ))
77
78      def getServersForUid(self, uid, complete=True, online=True):
79          uid = makeUUID(uid)
80          if complete:
81              res = self.session.execute(
82                  "select server from stored_object where uid = %s and complete = true
                        allow filtering", (uid, )).all()
83          else:
84              res = self.session.execute(
85                  "select server from stored_object where uid = %s allow filtering", (
                        uid, )).all()
86          if online:
87              def f(srv):
88                  return self.session.execute("select online from dataserver where
                        server = %s", (srv.server, )).one().online
89              res = list(filter(f, res))
90          print("getServersForUid", uid, res)
91          return res
92
93      def isServerOnline(self, server):
94          return self.session.execute("select online from dataserver where server = %s
                ", (server, )).one().online
95
96      def setComplete(self, uid, server):
97          uid = makeUUID(uid)
98          self.session.execute("update stored_object set complete = true where uid = %
                s and server = %s", (uid, server))
99
100     def markDeleted(self, path):
101         res = self.getUidForPath(path)
102         timestamp = datetime.now()
103         if res is not None:
104             self.session.execute("update object set deleted = %s where uid = %s and
                    created = %s",
```

```
105                                     (timestamp, res.uid, res.created))
106
107     def markUidDeleted(self, uid, created):
108         uid = makeUUID(uid)
109         timestamp = datetime.now()
110         self.session.execute("update object set deleted = %s where uid = %s and
                created = %s",
111                              (timestamp, uid, created))
112
113     def getUidForPath(self, path):
114         print("path", path)
115         path = str(path)
116         return self.session.execute("select * from pathToObject where path = %s", (
                path, )).one()
117
118     def getUidsForPath(self, path, noDeleted=False):
119         print("path", path)
120         path = str(path)
121         return self.session.execute("select * from object where path like %s", (path
                , )).all()
122
123     def getUidsForServer(self, server):
124         return self.session.execute("select uid from stored_object where server = %s
                 allow filtering", (server, ))
125
126     def addPendingUid(self, uid):
127         uid = makeUUID(uid)
128         self.session.execute("insert into pending_object(uid, enabled) values (%s,
                True)", (uid, ))
129
130     def updatePriority(self, uid, created, priority):
131         uid = makeUUID(uid)
132         self.session.execute("update object set priority=%s where uid=%s and created
                =%s", (priority, uid, created))
133
134     def getPendingUids(self, onlyEnabled=True):
135         if onlyEnabled:
136             return self.session.execute("select uid from pending_object where
                    enabled=True allow filtering").all()
137         return self.session.execute("select uid from pending_object").all()
138
139     def disablePendingUid(self, uid):
140         uid = makeUUID(uid)
141         self.session.execute("update pending_object set enabled=False where uid = %s
                ", (uid, ))
142
143     def removePendingUid(self, uid):
144         uid = makeUUID(uid)
145         self.session.execute("delete from pending_object where uid = %s", (uid, ))
146
147
148 database = Database()
```

### 9.5.2 Dataserver

Struttura

```

//un oggetto
create table object(
    uid text primary key,
    local_path text,   //percorso del file system dov'e' salvato
    size integer,   //dimensione in byte (totale)
    complete integer,    //true se e' completo, false se deve ancora essere
        totalmente trasferito
    created text,       //istante di creazione (nel dataserver)
    checksum text
);

//configurazione del dataserver
create table stats(
    storage int,        //storage totale in byte
    downspeed int,      //banda totale in download, in Mb/s
    upspeed int         //banda totale in upload, in Mb/s
);

//documenti che vanno eliminati
create table toBeDeleted(
    uid text primary key        //uid del documento
);
```

Codice

```
class Database:
    def __init__(self):
        try:
            self.connection = sqlite3.connect('database.db')
            self.cursor = self.connection.cursor()
            print("connected to database")

        except sqlite3.Error as error:
            print("error while connecting to database", error)

    def getObject(self, uid):
        print("getObject ------->", (str(uid), ))
        return self.cursor.execute("select * from object where uid = ?", (str(uid),
            )).fetchone()


    def deleteUid(self, uid):
        self.cursor.execute("delete from object where uid = ?", (uid, ))
        self.cursor.execute("delete from transfer where uid = ?", (uid, ))
        self.cursor.execute("delete from toBeDeleted where uid = ?", (uid, ))
        self.connection.commit()

    def nodeStats(self):
        return self.cursor.execute("select * from stats").fetchone()

    def getStoredData(self):
        return self.cursor.execute("select uid, created, complete from object").
            fetchall()

    def reservedSpace(self):
        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()
```

## 9.6 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.

### 9.6.1 Metaserver

Nel metaserver le operazioni periodiche sono:

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

Controllo dello stato dei dataserver

```
def onDataServerConnect(addr):
    print("server", addr, "connected")

    for elem in database.getUidsForServer(addr):
        database.addPendingUid(elem.uid)

    for elem in database.getPendingUids(onlyEnabled=False):
        database.addPendingUid(elem.uid)

def onDataServerDisconnect(addr):
    if database.isServerOnline(addr):
        database.updateDataServerStatus(addr, False)

        for elem in database.getUidsForServer(addr):
            database.addPendingUid(elem.uid)

        print("server", addr, "disconnected")


def checkDataServerStatus(addr):
    wasOnline = database.isServerOnline(addr)

    try:
        with Connection(addr) as conn:
            conn.write("status")
            res = conn.readline()

        status, reservedCapacity, totCapacity, downSpeed, bandDown, upspeed, bandUp
            = res
        reservedCapacity = int(reservedCapacity)
        totCapacity = int(totCapacity)
        downSpeed = float(downSpeed)
        bandDown = float(bandDown)
        upspeed = float(upspeed)
        bandUp = float(bandUp)
        database.updateDataServerStatus(addr, True, totCapacity - reservedCapacity,
            totCapacity,
                                         downSpeed - bandDown, upspeed - bandUp)

        if not wasOnline:
            onDataServerConnect(addr)
    except ConnectionRefusedError as err:
        print(err)
        if wasOnline:
            onDataServerDisconnect(addr)

def monitorDataServers():
    for server in database.getDataServers(onlyOnline=False):
        checkDataServerStatus(server.server)
```

Mantenimento del grado di replicazione dei documenti

```
def processPendingUids():
    for elem in database.getPendingUids():
        processPendingUid(database, elem.uid)

```

```
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
19     print("availableServers", availableServers)
20
21     serversContaining = list(serversContaining)
22
23     if numCopies == priority:
24         database.removePendingUid(uid)
25         return
26
27     if numCopies > priority:
28         random.shuffle(serversContaining)
29         target = serversContaining[0]
30
31         print("remove", uid, "target", target)
32
33         with Connection(target) as conn:
34             conn.write("deleteUid", uid)
35             res = conn.readline()
36             print(res)
37
38         database.removeStoredObject(uid, target)
39         return
40
41     if numCopies < priority:
42
43         if len(serversContaining) == 0:
44             database.disablePendingUid(uid)
45             return
46
47         if len(availableServers) > 0:
48             random.shuffle(availableServers)
49             random.shuffle(serversContaining)
50
51             try:
52                 source = serversContaining[0]
53                 target = availableServers[0]
54
55                 database.addStoredObject(uid, target)
56
57                 with Connection(target) as conn:
58                     conn.write("createUid", uid, size, checksum)
59                     print(conn.readline())
60
61                 with Connection(source) as conn:
```

```
62                      conn.write("transfer", uid, target)
63                      print(conn.readline())
64
65              except:
66                  print("ERROR")
67          else:
68              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,))
```

### 9.6.2 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():
```

```
    logging.info("processDeleteUids...")

    database = Database()
    uids = database.getToBeDeleted()

    with processingLock:
        logging.info("uids %s", uids)

        for uid, in uids:
            if uid not in processing:
                logging.info("delete %s", uid)
                uid, localPath, size, complete, created, checksum = database.
                    getObject(uid)
                if os.path.exists(localPath): os.remove(localPath)
                database.deleteUid(uid)
```

Scheduling ed esecuzione

```
schedule.every(5).seconds.do(getPerformance)
schedule.every(15).seconds.do(processDeleteUids)

def runPeriodicTasks(_):
    while True:
        schedule.run_pending()
        time.sleep(0.1)

_thread.start_new_thread(runPeriodicTasks, (None,))
```

## 9.7 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.

### 9.7.1 Caricamento di un file

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:
        size = getsize(localPath)
        print("file size", size)

        checksum = hashFile(localPath)

        conn.write("pushPath", remotePath, size, checksum, priority, user)
        res = conn.readline()

        if res[0] != "ok":
            print("ERR", res)
            return False

        state, uid, addr = res
    ...
```

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

```
def pushPath(self, args):
    path, size, checksum, priority, user = args
```

```
3
4       lock, lockUser = self.database.getPathLock(path)
5       if lock and lockUser != user:
6           self.write("err", "path locked by " + lockUser)
7           return False
8
9       dataservers = self.database.getDataServers()
10
11      if len(dataservers) == 0:
12          self.write("err", "no data servers available")
13          return False
14
15      random.shuffle(dataservers)
16
17      size = int(size)
18      target = None
19      for target in dataservers:
20          print("option server", target.server, "rem capacity", target.
                remaining_capacity, "file size", size)
21          if target.remaining_capacity > size:
22              break
23          else:
24              target = None
25
26      if target is None:
27          self.write("err", "no dataserver with sufficient capacity")
28          return False
29
30      uid = uuid4()
31      addr = target.server
32      ...
```

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
6       self.database.addObject(uid, local_path, size, checksum)
7
8       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

```
    self.database.markDeleted(path)
    self.database.addObject(uid, path, "root", size, priority, checksum)
    self.database.addStoredObject(uid, addr)

    self.write("ok", uid, addr)
```

Il client si collega al dataserver target e invia il file. Se il file era stato già parzialmente caricato nel dataserver (es. file di grosse dimensioni), il caricamento riprende da dover era rimasto. Se il file era già stato caricato completamente (ovvero il client tenta di caricare un file 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

```
    ...
    with Connection(addr) as conn:
        conn.write("pushUid", uid)

        res = conn.readline()
        if res[0] != 'ok':
            print(res)
            return False

        status, startIndex = res

        print("send starting from", int(startIndex))

        conn.writeFile(localPath, startIndex)
        print(conn.readline())
```

Dataserver

```
def pushUid(self, args):
    uid, = args

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

        if objinfo is None:
            self.write("ERR", "uid info not found")
            return False

        print(objinfo)

        uid, localpath, size, complete, created, checksum, = objinfo

        if complete:
            self.write("err", "file complete")
            return False

```

```
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")
```

### 9.7.2 Trasferimento di un file

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

```
with Connection(target) as conn:
    conn.write("createUid", uid, size, checksum)
    print(conn.readline())

with Connection(source) as conn:
    conn.write("transfer", uid, target)
    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

```
def transfer(self, args):
    uid, server = args

    with processingLock:
        res = self.database.getObject(uid)
        if res is None:
            self.write("err", "uid not found")
            return False

        processing.add(uid)

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

    with Connection(server) as target:
        target.write("pushUid", uid)
        status, startIndex = target.readline()
        target.writeFile(localPath, startIndex)

    with processing:
        processing.remove(uid)

    self.write("ok")
```

### 9.7.3 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

```
def get(localPath = "testin.txt", remotePath = "ale/file1", newFile=True, user="
    default"):
    if newFile and os.path.exists(localPath):
        os.remove(localPath)

```

```
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 = 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 getPath(self, args):
2       path, user = args
3
4       lock, lockUser = self.database.getPathLock(path)
5       if lock and lockUser != user:
6           self.write("err", "path locked by " + lockUser)
7           return False
8
9       res = self.database.getUidForPath(path)
10
11      if not res:
12          self.write("err", "path not found")
13          return False
14
15      if res.deleted is not None:
16          self.write("err", "path deleted")
17          return False
18
19      uid = res.uid
20      print("uid", uid)
21
22      addr = self._getServerForUid(uid)
23
24      self.write("ok", uid, addr)
```

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 file di grandi dimensioni) e legge il file. La chiamata usata in questo caso è *getUid*.

Client

```
1   startIndex = 0
2       if os.path.exists(localPath):
3           startIndex = os.path.getsize(localPath)
4
5       with Connection(addr) as conn:
6           conn.write("getUid", uid, startIndex)
```

```
7           res = conn.readline()
8           print(res)
9           status, size = res
10
11          with open(localPath, "a+b") as out:
12              conn.readFile(size, out)
```

Codice

```
1   def getUid(self, args):
2       uid, startIndex = args
3       startIndex = int(startIndex)
4
5       with processingLock:
6           res = self.database.getObject(uid)
7
8           if res is None:
9               self.write("err", "specified uid not present")
10              return False
11
12          print(res)
13
14          uid, localPath, size, complete, created, checksum = res
15
16          if not complete:
17              self.write("err", "not complete")
18              return False
19
20          processing.add(uid)
21
22      self.write("ok", size-startIndex)
23      self.writeFile(localPath, startIndex)
24
25      with processingLock:
26          processing.remove(uid)
```

### 9.7.4 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, = args
16
17      addr = self._getServerForUid(uid)
18
19      self.write("ok", addr)
```

### 9.7.5 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 directory (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      for elem in res:
7          lock, lockUser = self.database.getPathLock(elem.path)
8          if lock and lockUser != user:
9              self.write("err", "path locked by " + lockUser)
10         else:
11             uid = elem.uid
12             print("delete", uid)
13             if elem.deleted is None:
14                 self.database.markUidDeleted(uid, elem.created)
15
16     self.write("ok")
```

### 9.7.6 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.

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

Codice

```
def permanentlyDeletePath(path, user="default"):
    conn = Connection(METASERVER)
    conn.write("permanentlyDeletePath", path, user)
    print(conn.readline())
    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)

    for elem in res:
        lock, lockUser = self.database.getPathLock(elem.path)
        if lock and lockUser != user:
            self.write("err", "path locked by " + lockUser)
        else:
            uid = elem.uid
            print("permanentlyDelete", uid)

            if elem.deleted is None:
                self.database.markUidDeleted(uid, elem.created)
            self.database.updatePriority(uid, elem.created, 0)
            self.database.addPendingUid(uid)

    self.write("ok")
```

### 9.7.7 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

```
1  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.

### 9.7.8 Modificare la priorità di un documento

È possibile in ogni momento modificare la priorità di un determinato documento. Con *updatePriorityForPath* viene modificata la priorità dell'ultima versione non eliminata di un documento, mentre con *updatePriorityForUid* è possibile effettuare l'operazione su una versione specificata.

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 = 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.getUidForPath(path)
10
11      if res is None:
12          self.write("err", "path " + path + " not found")
13          return False
14
15      uid = res.uid
16      print(uid)
17
18      self.database.updatePriority(uid, res.created, priority)
19      self.database.addPendingUid(uid)
20
21      self.write("ok")
22
23  def updatePriorityForUid(self, args):
24      uid, priority = args
25      priority = int(priority)
26
27      res = self.getUid(uid)
28
29      self.database.updatePriority(uid, res.created, priority)
30      self.database.addPendingUid(uid)
31
32      self.write("ok")
```

### 9.7.9 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)
```

# 10 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 directory 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 file eliminato in questo modo. Invece *permanentlyDeletePath* elimina definitivamente un documento (o directory), 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.

# References