Tesina Distributed FS (main)

alessiocorrado99

September 2020

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*.
Il *numero di repliche minimo* è: 2.

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...).

.

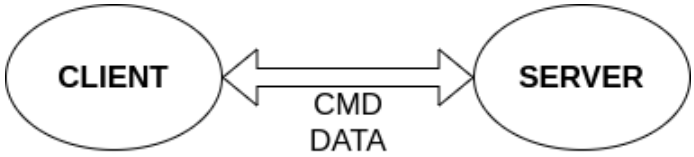


Figure 1: Single-server

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

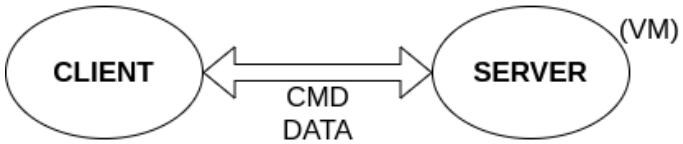


Figure 2: Single-server con virtualizzazione

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

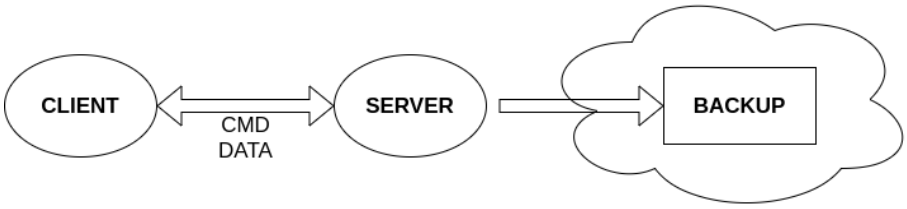


Figure 3: Aggiunta del backup nel cloud

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.

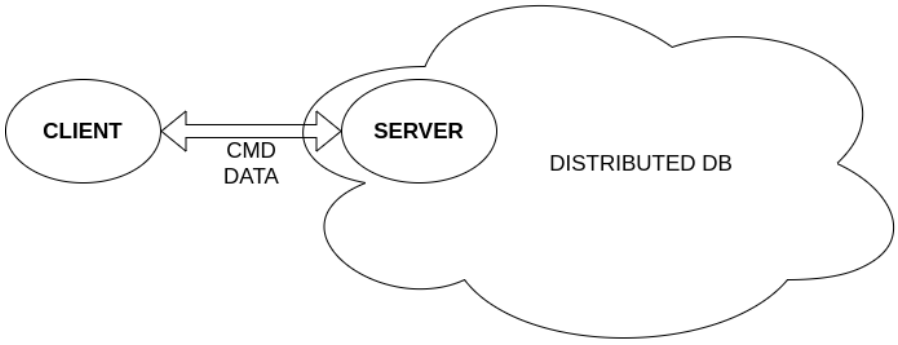


Figure 4: Utilizzo di un database distribuito

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)

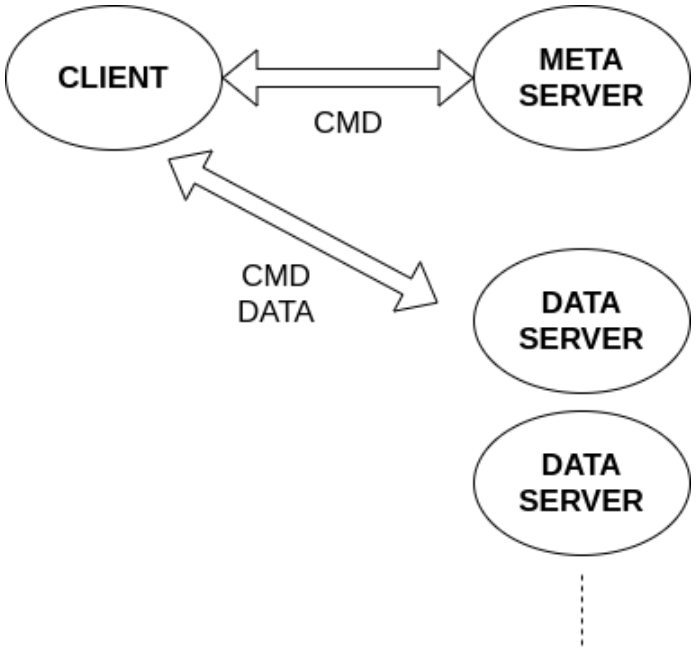


Figure 5: Separazione di meta e data server

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 complessita nel gestire operazioni *concorrenti* ed *asincrone* in nodi diversi

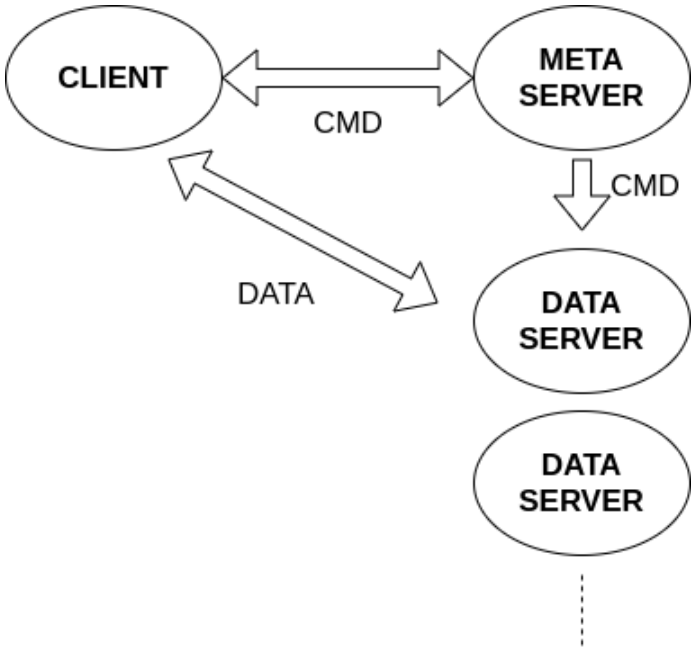


Figure 6: Meta + data server, il meta server dialoga con i data server

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

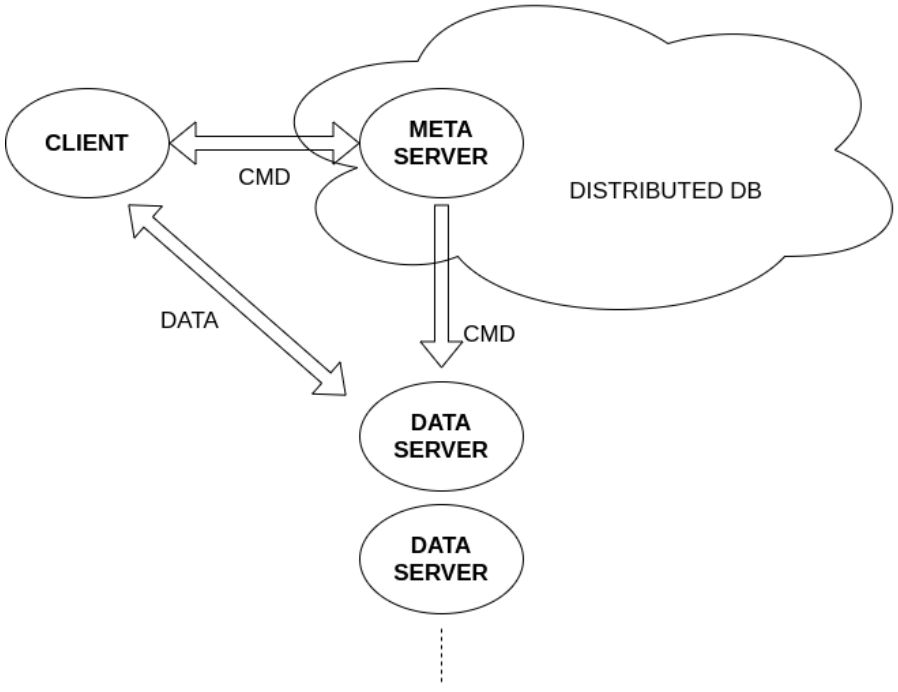


Figure 7: Il meta server è implementato come database distribuito

3 Comandi del client

Di seguito l'elenco e descrizione dei comandi gestiti dal client.

- list(path="root", recursive=true, inclue_meta=false, format=json)
  Lista il contenuto di una cartella.
  Argomenti:
  - path: directory target. Default: "root"
  - recursive: solo cartella o tutto il sottoalbero. In ogni caso non segue i link. Default: true
  - include_meta: include i metadati relativi ad ogni entry. Default: false
  - format: formato di output. valori: csv, json. Default: json.
- mkdir(path)
  Crea una directory
- meta(path)
  Metadati di un path
- get(path)
  Download di un file
- push(file, path)
  Upload di un file
- rem(path)
  Delete di un path

4 Metadati dei path

I metadati associati ad un path sono:

1. uid: identificatore univoco dell'oggetto
2. path
3. creation_date
4. size: dimensione in byte, 0 per le directory
5. owner: utente proprietario, default chi l'ha creato
6. visible: true/false. Indica se è visibile agli utenti diversi dall'owner
7. min_copies: numero minimo di copie, coincide con la priorità

5 Protocollo comunicazione

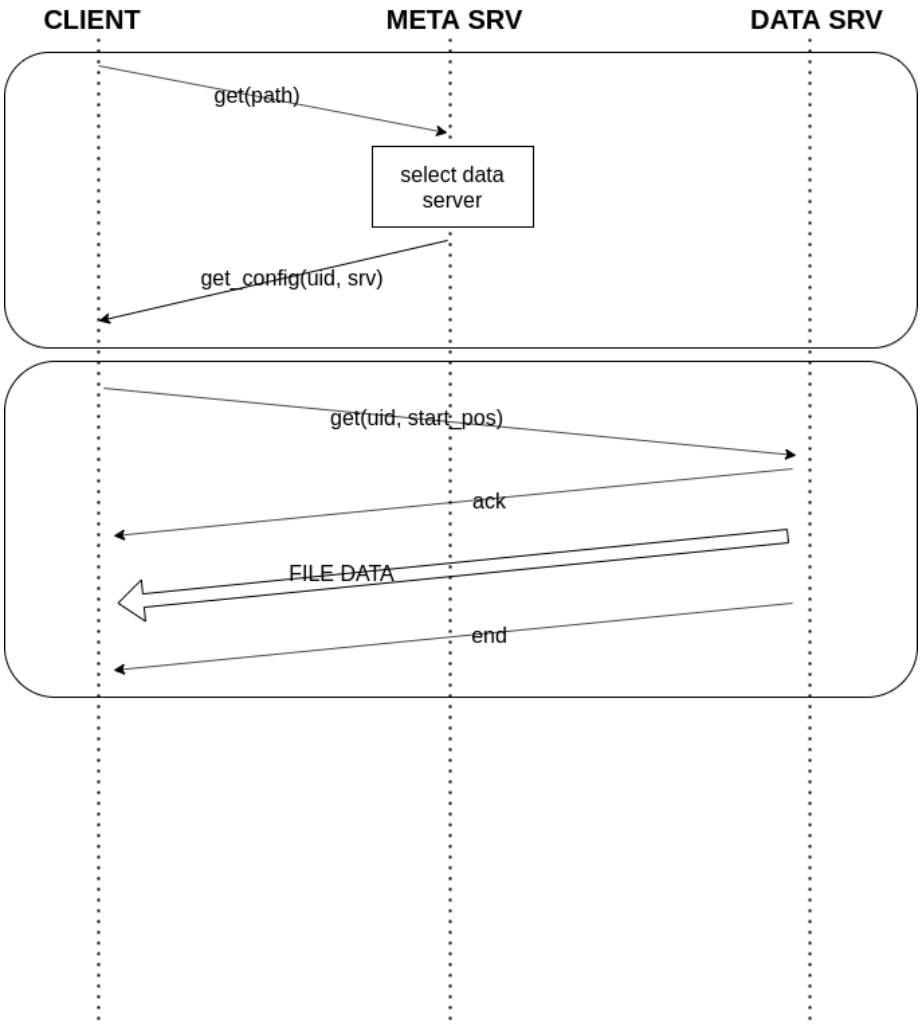
Di seguito i flussi di dati coinvolti nelle diverse tipologie di richieste.

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 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.

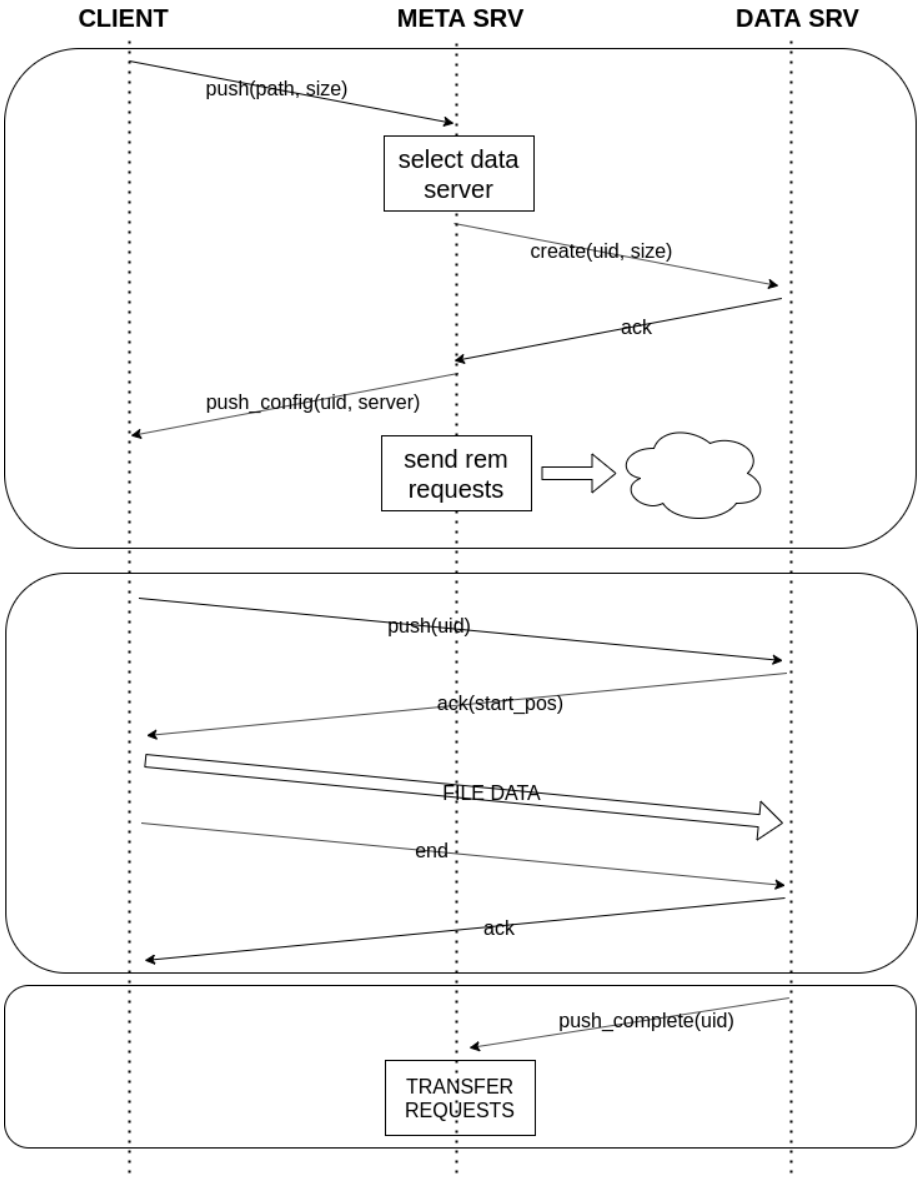
Inizialmente il *client* contatta il *meta server*, inviando *path* e *size*. Il *meta server* seleziona il *data server* ottimale per gestire la richiesta (deve poter contenere la risorsa e non essere sovraccarico). Il *meta server* si occupa anche di generare un nuovo *uid* da assegnare alla risorsa.

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.

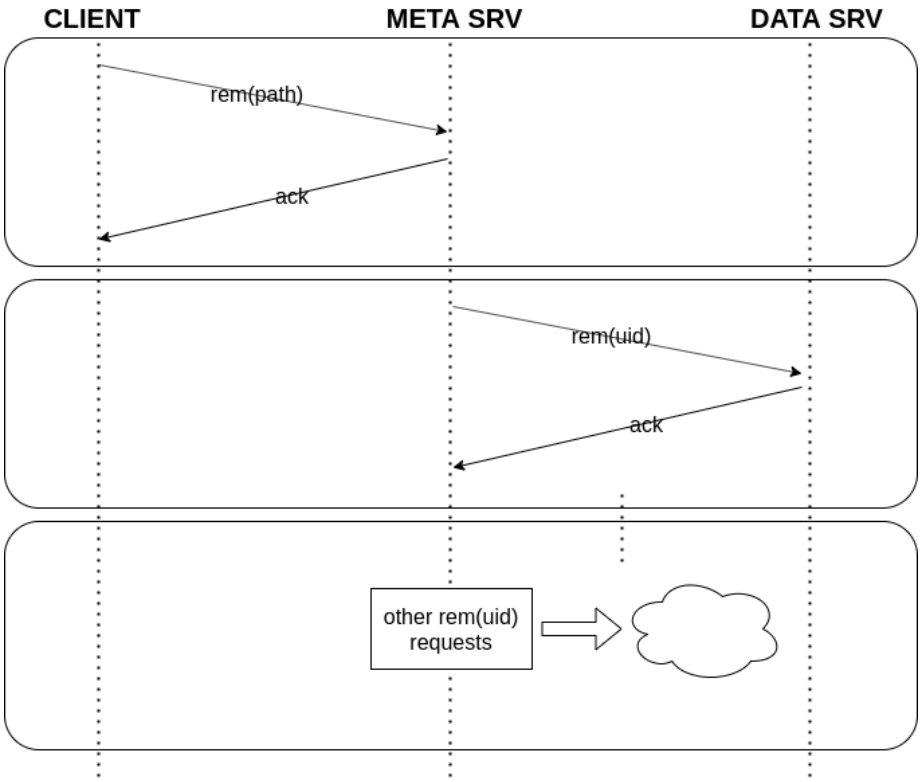


5.3 rem

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)*.

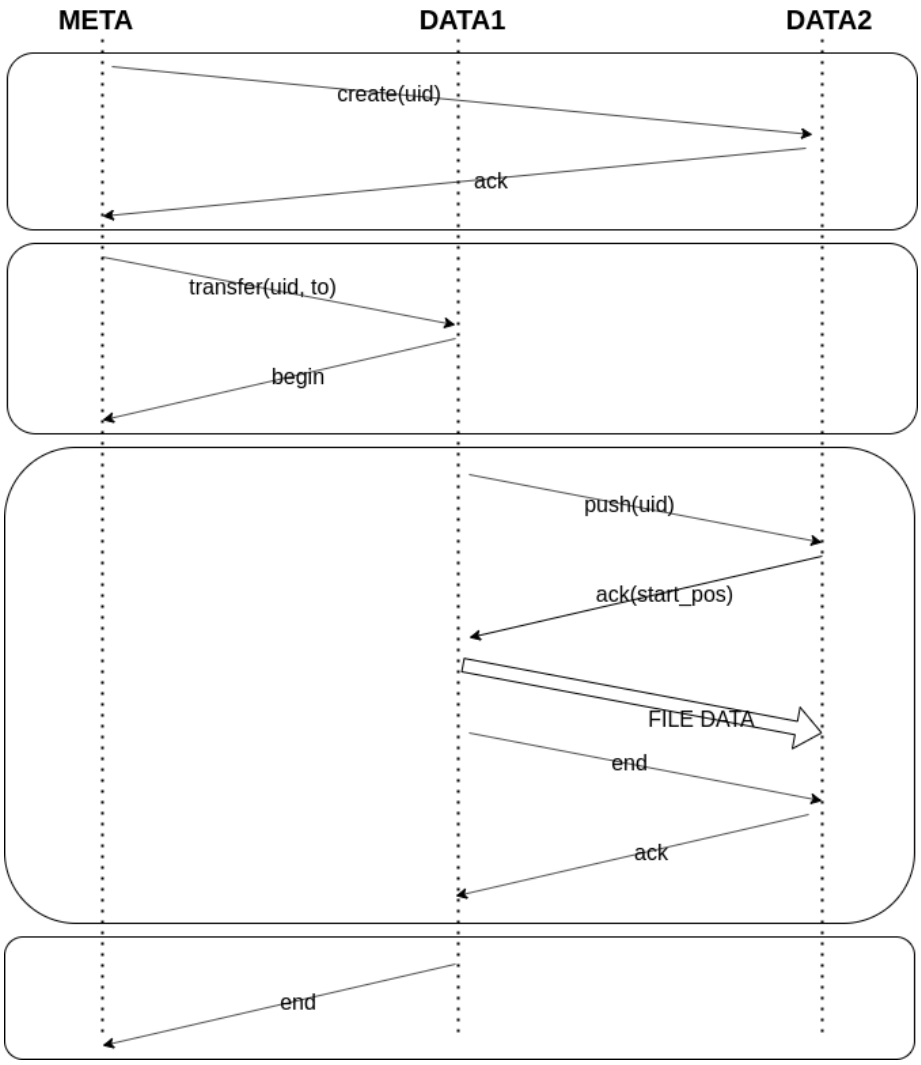


5.4 transfer

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.

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.

References