SISTEMI DISTRIBUITI

Apache ZooKeeper

Luca Mengozzi (luca.mengozzi4@studio.unibo.it)
Elia Calisesi (elia.calisesi@studio.unibo.it)

Anno accademico 2020-2021 Cesena

INDICE

INTRODUZIONE

ZooKeeper è un progetto Apache open source che fornisce un servizio centralizzato per fornire informazioni di configurazione, denominazione, sincronizzazione e servizi di gruppo su cluster di grandi dimensioni nei sistemi distribuiti. L'obiettivo è rendere questi sistemi più facili da gestire con una propagazione delle modifiche migliorata e più affidabile.

Per le applicazioni, ZooKeeper fornisce un'infrastruttura per la sincronizzazione tra nodi mantenendo in memoria sui server Zookeeper le informazioni sul tipo di stato. Un server ZooKeeper conserva una copia dello stato dell'intero sistema inserendo queste informazioni nei file di registro locali. I cluster Hadoop di grandi dimensioni sono supportati da più server ZooKeeper, con un server principale che sincronizza i server di livello superiore.

All'interno di questo servizio, un'applicazione può creare quello che viene chiamato uno znode, che è un file che persiste in memoria sui server ZooKeeper. Lo znode può essere aggiornato da qualsiasi nodo nel cluster e qualsiasi nodo può registrarsi per essere informato delle modifiche a quello znode.

In parole povere, le applicazioni possono sincronizzare le proprie attività nel cluster distribuito aggiornando il proprio stato in uno znode. Lo znode informa quindi il resto del cluster del cambiamento di stato di un nodo specifico. Questo servizio di centralizzazione dello stato a livello di cluster è fondamentale per le attività di gestione e serializzazione su un ampio set di server distribuiti.

APACHE ZOOKEEPER

2.1 ZNode

Ogni nodo in un albero ZooKeeper è chiamato znode. Ognuno di questi ha una struttura statica che include:

1. il numero di versione causato dal cambio di dati o delle ACL (Access Control List), quindi ad ogni aggiornamento dei dati il numero di versione cresce:
2. timestamp;

Questi due campi assieme danno la possibilità a ZooKeeper di validare la cache e coordinare gli aggiornamenti. Un esempio può essere che un client richiede dati al server, il quale glieli comunica insieme a una versione, quella più recente. Quando il client vuole eseguire un aggiornamento o una cancellazione

deve fornire il numero di versione, se questo è diverso da quello dei dati attuali nello znode l'operazione fallisce.
I client possono anche tenere sotto controllo uno znode in modo tale da intercettare degli eventi quando vengono eseguiti dei cambiamenti sui suoi dati.
Ogni znode contiene dei dati che possono essere letti o riscritti in blocco, inoltre tiene memorizzata un'ACL che indica le possibili operazioni che possono eseguire gli utenti sul nodo stesso.
La caratteristica principale di ZooKeeper è che non è stato progettato per gestire grandi quantità di dati, ma per gestire la coordinazione dei dati. I dati negli znode possono essere al massimo di 1KB per garantire velocità. Nel caso vengano trasferite grosse quantità di dati, ZooKeeper dovrebbe essere utilizzato solo per salvare i puntatori collegati agli indirizzi dove sono salvati i dati.
Uno znode può essere:

1. effimero, questi nodi vengono creati durante una sessione ed eliminati quando viene terminata. Questo tipo di znode non può avere figli;
2. sequenziale, questo tipo di nodo alla fine del percorso ha un contatore a dieci caratteri composto di 0 e il numero del conteggio alla fine;
3. container, questo nodo è utile per eseguire mansioni come il leader, quando l'ultimo figlio di un container viene cancellato, questo viene preso in considerazione dal server come candidato alla sua eliminazione nel caso ce ne sia bisogno in futuro;
4. TTL, quando viene creato uno znode sequenziale o persistente, si può settare opzionalmente il TTL (Time To Live) in millisecondi. Se il nodo non ha TTL e non ha figli viene candidato come prossimo eliminato dal server.

2.2 Tempo in ZooKeeper

ZooKeeper ha diverse interpretazioni del tempo che sono:

1. Zxid (Zookeeper Transaction Id), ogni cambiamento nello stato del sistema riceve un timbro sotto forma di zxid che dà la possibilità di mantenere ordinati tutti i cambiamenti del sistema;
2. Version number, ogni cambiamento in un nodo aumenta il numero di versione di quel nodo, ci sono tre versioni in uno znode:
   (a) Versione, numero di versione causato dal cambiamento dello znode;
   (b) Versione-C, numero di versione causato dal cambiamento dei figli dello znode;
   (c) Versione-A, numero di versione causato dal cambiamento della ACL.

3. Ticks, quando si utilizza ZooKeeper con multi-server, i server utilizzano dei ticks per indicare le tempistiche degli eventi come aggiornamento di stato, time out sessione, etc. . . ;
4. Real Time, Zookeeper non utilizza il tempo fisico del clock, ma basa le sue operazioni sul timestamp nello stato dello znode per indicare le sue modifiche.

2.3 Struttura dello stato in ZNode

La struttura dello stato di uno znode è composta dai seguenti campi:

1. czxid, lo zxid assegnato alla creazione dello zonde;
2. mzxid, lo zxid assegnato all'ultima modifica dello znode;
3. pzxid, lo zxid assegnato all'ultima modifica dei figli dello znode;
4. ctime, il tempo in millisecondi dalla creazione dello znode;
5. mtime, Il tempo in millisecondi dall'ultima modifica del nodo;
6. version;
7. cversion;
8. aversion;
9. ephemeralOwner, l'id della sessione del proprietario dello znode effimero preso in considerazione, se non è effimero è settato a 0;
10. dataLength, la lunghezza del campo data di un nodo;
11. numChildren, il numero di figli di uno znode.

2.4 Sessione in ZooKeeper

Un client stabilisce una connessione con il servizio di ZooKeeper utilizzando una sequenza ben precisa di passi. Inizialmente si trova allo stato CONNECTING, dove prova a connettersi a un server del servizio. Una volta avvenuto il collegamento passa allo stato CONNECTED. Questi due stati sono quelli utilizzati principalmente ma possono essercene molti altri illustrati nella seguente immagine:

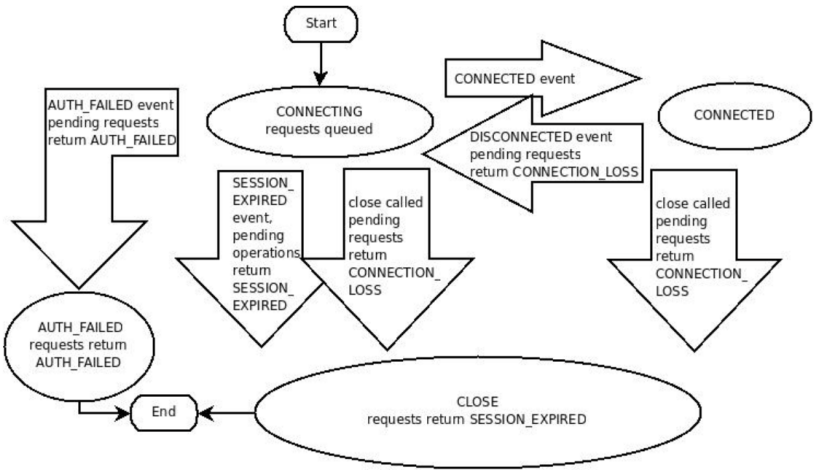


Figura che illustra i possibili stati e le transizioni di statodi un client Zookeeper

Per creare la sessione il client deve fornire una stringa di connessione composta da

<host:porta>

corrispondente al server esercente il servizio ZooKeeper. Si può aggiungere anche il percorso a uno znode specifico aggiungendolo dopo la porta del server, ad esempio "127.0.0.1:2781/app/myapp", in questo modo il client eseguirà operazioni su quel nodo e su tutti i suoi figli. Questa feature dà la possibilità di gestire al meglio le risorse in ambienti molto grandi in cui si vuole assegnare solo particolari servizi di ZooKeeper a determinati utenti.
Quando un client si collega al server ZooKeeper, viene creata una sessione, identificata da un numero a 64-bit che viene assegnato al client, quest'ultimo se vuole collegarsi a un nuovo server del servizio utilizzerà questo id nell'handshake iniziale. Per rendere più sicuro questo processo di cambio di sessione, il primo server, una volta creata la sessione, invia al client una password che dovrà comunicare ogni volta ai nuovi server a cui si connetterà.
Un altro parametro che il client può settare in una sessione è il timeout espresso in millisecondi, al quale il server può rispondere con il timeout che è disposto a concedere al client. Questo parametro serve per terminare la sessione quando un client non invia richieste al server entro il tempo prefissato. La chiusura di una sessione è gestita dal cluster ZooKeeper e non dal client, ed eliminerà tutti i nodi effimeri creati nella sessione e comunicherà al client che la sessione è terminata quando quest'ultimo proverà a riconnettersi al servizio.
Oltre ai due parametri appena illustrati c'è anche il watcher che notifica al client ogni possibile cambio di stato della connessione con il server e del client stesso, per esempio quando un client perde la connessione con un server, il watcher manda una notifica al client.

La connessione rimane attiva fino a quando il client interroga il server e nel caso in cui non abbia richieste specifiche può inviare delle istanze PING che servono a mantenere la connessione attiva e a verificare che la connessione con il server sia ancora funzionante, la stessa cosa vale per il server.
Una volta che viene stabilita la connessione tra server e client ci possono essere solo due motivi che possono portare alla chiusura della sessione:

1. l'applicazione richiede una funzione che non è più attiva o valida;
2. Il client si disconnette dal server ZK nel caso in cui abbia delle operazioni in attesa.

Il client può aggiornare la stringa di connessione fornendo una nuova lista di coppie del tipo: host:porta. Quest'azione invoca un algoritmo probabilistico che può disconnettere il client dal server con una probabilità del 40% per poi riconnetterlo ai nuovi server forniti. Se i nuovi server non sono disponibili il client viene riconnesso ai vecchi.
La creazione e la cancellazione di sessioni sono il collo di bottiglia di ZooKeeper. Questo problema nelle nuove versioni è stato migliorato con l'introduzione delle sessioni locali, che non hanno però le stesse funzionalità che hanno le sessioni normali. La sessione locale viene creata attivando localSessionEnabled.
Inoltre, se localSessionUpgradingEnabled è disabilitato:

1. La sessione locale non può creare nodi effimeri;
2. Una volta persa la sessione un client non può inizializzarne una nuova con altri server;
3. Quando una sessione locale si connette a un servizio ZooKeeper, tutte le informazioni sono mantenute sul server a cui è connesso;
4. Il PING, la chiusura della connessione e tutte le operazioni sulla sessione sono gestiti dal server a cui il client è connesso.

2.5 ZooKeeper Watches

Tutte le operazioni di lettura in ZooKeeper possono essere affiancate a un watch event che segnala al client quando i dati a cui è collegato cambiano. Le tre caratteristiche fondamentali di un watch sono:

1. One-time trigger, il watch segnala una sola volta il cambiamento, a meno che alla lettura non consegua l'inizializzazione di un nuovo watch;
2. Risposta al client, la notifica è inviata a un solo client e gestisce la sincronizzazione con il server; quindi, una modifica avverrà sempre dopo un aggiornamento dei dati;

3. Watch sui dati e sui nodi, grazie a questa funzionalità ZooKeeper riesce a mantenere separati i watch dei dati da quelli dei figli (child). Quindi i cambi di stato possono influire sui dati, sui nodi e sul watch dello znode padre del nodo che ha subito il cambiamento.

I watch sono mantenuti localmente nei server a cui il client è connesso, nel caso in cui un client venga disconnesso e riconnesso a un nuovo server, i watch verranno settati in maniera trasparente.
I watch possono eseguire fondamentalmente tre tipi di azioni:

1. Esistenza (Exists);
2. Ottenere i dati (getData);
3. Ottenere i figli (getChildren).

Gli eventi che vengono rilevati da un watch sono:

1. Evento di creazione, nato da una richiesta di esistenza;
2. Evento di cancellazione, attivato da una richiesta di Exists, getData e getChildren;
3. Evento di cambiamento, scatenato da una richiesta di esistenza o di getData;
4. Evento di un figlio, insorge tramite richiesta getChildren.

L'utente può eliminare i watch dallo znode manualmente con la funzione removeWatches, mentre il client elimina automaticamente il watch quando perde connessione con il server.
Dopo aver rimosso il watch verrà inviato lo stesso un trigger nel caso in cui ci sia:

1. Evento di rimozione figlio, nel caso in cui il watch fosse stato settato con la richiesta di trigger sul getChildren;
2. Evento di rimozione dei dati, se il watch era stato aggiunto con la funzione di controllare l'esistenza o il getData dei dati.

Vantaggi dei watch:

1. I watch sono ordinati rispetto agli eventi e agli altri watch;
2. Viene assicurato che il client riceverà la notifica di modifica dei dati prima che ci acceda;
3. L'ordine degli eventi scatenati dai watch è ordinato rispetto agli aggiornamenti del servizio ZooKeeper.

Svantaggi e cose da ricordare sui watch:

1. I watch standard inviano solo la notifica di un evento e poi vengono eliminati, per tenere sotto controllo altri cambiamenti occorre settarne dei nuovi;

2. La latenza tra l'invio della notifica al client e il settaggio di un nuovo watch può fare perdere al client delle modifiche avvenute il quel lasso di tempo;

3. Un watch anche se è settato per più funzioni manda solo una notifica per il primo evento che si verifica;

4. Quando un client si disconnette da un server, vengono eliminati anche i watch fino a quando la connessione non è stata ristabilita. Nel frattempo, tutte le notifiche dei watch vengono gestite da un handler che le invierà al client non appena si riconnetterà.

2.6 ZooKeeper ACLs

ZooKeeper utilizza delle Access Control Lists, molto simili a quelle di UNIX, per controllare l'accesso agli znode. Ogni funzione che un utente può svolgere è associata a un bit che indica se quell'utente ha il permesso oppure no di svolgerla. A differenza dello standard UNIX, i nodi ZooKeeper non gestiscono il tipo di utente che accede al dato (proprietario, gruppo, resto del mondo) ma mantiene delle ACL che specificano per determinati id quali permessi hanno a disposizione.
Ogni risorsa di ZK ha la propria ACL, quindi un utente che ha determinati permessi su uno znode, non è detto che abbia lo stesso tipo di permessi sui suoi figli. Tutti i permessi vengono assegnati una volta che l'utente si autentica al servizio ZooKeeper fornendo gli id che gli sono stati assegnati. La struttura delle ACL è del tipo:

<schema:espressione, permesso>

Un esempio potrebbe essere:

ip:19.22.0.0/16, READ

indica che per tutti gli IP che iniziano con 19.22 gli viene dato il permesso di lettura.
I permessi che possono dare le ACLs sono:

1. CREATE, permesso di creare figli in un nodo;

2. READ, si possono leggere i dati da un nodo e ottenere la lista dei figli;

3. WRITE, si possono modificare i dati di un nodo;

4. DELETE, si possono cancellare i figli di un nodo;

5. ADMIN, facoltà di gestire i permessi di un nodo, simile al creatore della risorsa dei sistemi UNIX. Grazie a questi permessi si possono creare gerarchie tra i nodi, ad esempio un client può creare uno znode dal quale chiunque altro può accedere e creare nuovi zonde figli, ma non può cancellarli.

Gli schemi delle ACL sono:

1. World, ha un unico ID che rappresenta chiunque;
2. Auth, è uno schema che dà la possibilità di accedere a uno znode solo al proprietario;
3. Digest, composto da user:password serve per autenticare un client;
4. Ip, utilizza l'ip dell'host per identificare il client;
5. X509, utilizza X500 Principal come ID di un cliente.

ZooKeeper ha molti metodi di autenticazione che possono essere utilizzati in ambienti diversi collegabili tra loro. Questa versatilità è data dalla suddivisione dell'autenticazione in due fasi:

1. Autenticazione del client, è fatta appena il client si connette al server. Il client invia delle sue informazioni che devono essere validate e vengono associate alla connessione;
2. ZooKeeper gestisce l'associazione tra il client e i suoi permessi nell'ACL. Le entries delle ACL sono formate da coppie <idspec, permessi> dove idspec può essere una semplice stringa associata alle informazioni di autenticazione relative alla connessione o potrebbe essere un'espressione valutata su sulle informazioni di autenticazione.

2.7 Garanzia di consistenza

ZooKeeper è uno servizio scalabile ad alte prestazioni in quanto sia le scritture che le letture sono state pensate per essere veloci, quest'ultime però sono più serrate rispetto alle scritture. La motivazione è legata al fatto che ZooKeeper fornisce vecchie informazioni alle quali sono garantite:

1. Consistenza sequenziale, gli aggiornamenti da un client sono applicati nell'ordine in cui sono stati effettuati;
2. Atomicità, gli aggiornamenti possono fallire o riuscire, non ci sono risultati parziali;
3. Immagine singola di sistema, un client vede sempre lo stato attuale del servizio, non vedrà mai una versione precedente;

4. Affidabilità, una volta che gli aggiornamenti sono stati applicati questi esistono fino quando non vengono aggiornati a loro volta. Questo garantisce:

   (a) Se un client riceve una risposta di successo significa che l'aggiornamento è stato effettuato. Se ci sono eventuali errori non viene notificato nulla al client. Quindi un aggiornamento si ritiene effettuato solo quando si riceve la risposta di successo;

   (b) Ogni aggiornamento inviato non verrà mai annullato dal server.

5. Tempestività, la visione che ha il client del sistema ZK è aggiornata entro un breve periodo di tempo (10 secondi circa) se non vengono notificati nuovi stati del sistema il client rileverà che il servizio è stato interrotto.

La cosa di cui non tengono conto molti sviluppatori è che due client possono avere visioni differenti del sistema in un determinato momento, probabilmente a causa di un ritardo di rete. Quindi può capitare che un utente invii un aggiornamento e un altro client non lo rilevi per un breve periodo di tempo. Quindi è giusto invocare il metodo sync() dalle API di ZooKeeper per sincronizzare il client con il server prima di effettuare una lettura.

2.8 Bindings

Le librarie dei client in ZooKeeper sono scritte in due linguaggi:

1. Java
2. C.

Java Binding

In Java ci sono due package che gestiscono il binding di ZooKeeper e sono:

1. Org.apache.zookeeper;
2. Org.apache.zookeeper.data;

Il resto dei package viene utilizzato internamente dal servizio.
La classe principale che utilizza un client è la classe ZooKeeper composta da due costruttori che differiscono l'uno dall'altro in base a due campi opzionali, id di sessione, password
Tramite Java è possibile salvare l'id di sessione e la password per poterli riutilizzare e recuperare la sessione precedente.
Quando viene creato un oggetto ZooKeeper, vengono creati anche due thread:

1. I/O thread, dove vengono eseguiti tutti gli input, gli output e si occupa anche di mantenere attiva la sessione, gestire l'eventuale riconnessione al servizio, la gestione degli Heartbeat e della sincronia;

2. Event thread, gestisce tutte le risposte di metodi asincroni, gestisce e notifica eventuali eventi.

L'Heartbeat, in italiano battito cardiaco, viene inviato da un client al server per mantenere viva la connessione.
Gli aspetti su cui focalizzarsi sono:

1. Tutte le chiamate asincrone e i callback dei watch vengono effettuati in ordine, uno alla volta;
2. I callback non bloccano l'elaborazione del thread I/O e l'elaborazione delle richieste sincrone;
3. Le risposte alle chiamate sincrone possono non essere restituite nell'ordine corretto.

Infine, le regole associate all'arresto sono semplici: una volta che un oggetto ZooKeeper viene chiuso o ha un'eccezione (SESSION_EXPIRED e AUTH_FAILED) diventa non valido. Alla chiusura, i due thread si chiudono e qualsiasi ulteriore accesso all'handler del servizio ZooKeeper viene rifiutato.
Le proprietà che possono essere settate per il client Java sono:

1. Zookeeper.sasl.client, booleano che abilita autenticazione SASL;
2. Zookeeper.sasl.clientconfig, specifica la chiave di contesto nel file di login JAAS;
3. Zookeeper.server.pricipal, specifica il server principale a cui fare riferimento per l'autenticazione, nel caso venga specificato non è necessario settare i parametri per il login (sasl.client.username, client.canonicalized.hostname, server.realm);
4. Zookeeper.sasl.client.username, è diviso in tre parti: primaria, istanza e dominio. Il formato tipico è: primaria/istanza@dominio, dove viene specifcata la parte principale del server. L'istanza è derivata dall'IP del server. Il server invece gestirà username/IP@dominio, dove username è client.username, l'IP è l'IP del server e il dominio è il valore di zookeeper.server.realm;
5. Zookeeper sasl.client.canonicalize.hostname, se server.principal non è settato serve a determinarlo, può essere settato a TRUE o FALSE;
6. Zookeeper.server.realm, il dominio che fa parte del server principale;
7. Zookeeper.disableAutoWatchReset, booleano utilizzato per abilitare o disabilitare automaticamente i watch al momento della riconnessione;
8. Zookeeper.client.secure, se il client vuole connettersi con una connessione sicura oppure no, nel caso venga utilizzata quella sicura verrà utilizzato SSL;

9. Zookeeper.clientCnxnSocket, specifica quale ClientCnxnSocket utilizzare, i valori possibili sono:
   (a) Org.apache.zookeeper.ClientCnxnSocketNIO, per connessione normale;
   (b) Org.apache.zookeeper.ClientCnxnSocketNetty, per instaurare una connessione sicura;
10. Zookeeper.ssl.keyStore.location e zookeeper.ssl.keyStore.password specifica il percorso a un file JKS che contiene le credenziali per l'accesso;
11. Zookeeper.maxbuffer, determina la dimensione massima dei dati provenienti dal server, la dimensione deve riamanere sotto 1MB altrimenti verrà lanciata un'eccezione;
12. Zookeeper.kinit, definisce il percorso al kinit binario, quello predefinito è "/usr/bin/kinit".

2.9 Linea di comando

Per utilizzare ZooKeeper da linea di comando Windows occorre settare le variabili di sistema, dopodiché sarà possibile svolgere le funzioni che verranno illustrate di seguito.

Avvio server zookpeer

Per avviare un server zookeper si utilizza:

"bin/zkServer.sh start" - LINUX "zkServer.cmd" - WINDOWS

Successivamente dovrà essere creato il client che si connetterà al servizio eseguendo da linea di comando:

"bin/zkClient.sh" DA LINUX
"zkCli" DA WINDOWS

Dal client ora si possono eseguire le seguenti operazioni:

1. Creare uno znode;
2. Ottenere dei dati (GET);
3. Osservare uno znode per tenere sotto controllo i suoi cambiamenti (WATCH);
4. Inserire dei dati (SET);
5. Creare un figlio di uno znode;
6. Verificare lo stato di uno znode;

7. Rimuovere o eliminare uno znode.

Creare znode

Si possono creare znode con un percorso specifico inserendo in attributo il tipo di znode che può essere:

1. Effimero (-e), si elimina automaticamente al termine della sessione;
2. Sequenziale (-s), garantisce che il percorso dello znode sia unico;
3. Persistente (-p), aggiunge un numero di sequenza di dieci caratteri al percorso dello znode, ad esempio path/myapp viene convertito in /myapp0000000001, poi 02 etc.

Sintassi per creare uno znode:

create -tipo /path/data

Get dei dati

Ritorna i dati e i metadati dello znode specificato. Verranno ottenute informazioni anche di quando risale l'ultima modifica, dove è avvenuta e informazioni sui dati.
Per eseguire il comando la sintassi è la seguente:

get /path

Watch dati

Serve per ricevere notifiche quando uno znode specificato o un suo figlio cambiano i dati, si può utilizzare solo con l'ausilio del comando GET:

get /path 1

Set dati

Viene utilizzato per settare dei dati su uno znode specifico, dopo aver settato i dati si possono verificare con il comando GET. La sintassi:

set /path /data

Crea figlio

Creare un figlio di uno znode è simile a creare uno znode, l'unica differenza è che il percosrso del figlio avrà il percorso dello znode padre.

create /parent/path/child/path/data

Lista dei figli

Serve per mostrare la liste dei figli di uno znode, la sintassi:

ls /path

Controllo dello stato

Utilizzato per controllare lo stato di uno znode, si esegue con il comando:

stat /path

Rimuovere/cancellare

Per rimuovere o cancellare uno znode si utilizzano le seguenti linee di codice:

rmr /path REMOVE
delete /path DELETE
deleteall /path DELETE ALL

Delete a differenza di delete all cancella uno znode esclusivamente se non ha figli, altrimenti per eliminare ricorsivamente tutto il sotto albero identificato da un nodo si deve utilizzare delete all.

2.10 API Java

Zookeeper mette a disposizione dell'utente delle API Java per poter sviluppare applicazioni sfruttando le sue funzionalità.
Utilizzando le API di ZooKeeper, un'applicazione può connettersi, interagire, manipolare dati, coordinare e infine disconnettersi da un insieme ZooKeeper.
L'API ZooKeeper fornisce metodi sia sincroni che asincroni.
La parte centrale delle API ZooKeeper è la classe ZooKeeper che ha i seguenti metodi:

1. connect – si connette a ZooKeeper
2. create – crea uno znode
3. exists – controlla se esiste uno znode e le sue informazioni
4. getData - ottiene dati da un particolare znode
5. setData - imposta i dati in un particolare znode
6. getChildren - ottiene tutti i sottonodi disponibili in un particolare znode
7. delete – ottiene un particolare znode e tutti i suoi figli
8. close – chiude una connessione

Connect

La classe ZooKeeper fornisce funzionalità di connessione tramite il suo costruttore
ZooKeeper (Stringa connectionString, int sessionTimeout, Watcher watcher)
Dove:

1. connectionString - Host ZooKeeper;
2. sessionTimeout - timeout della sessione in millisecondi;
3. watcher - un oggetto che implementa l'interfaccia "Watcher". ZooKeeper restituisce lo stato della connessione tramite l'oggetto watcher;
4. Viene creata una nuova classe helper ZooKeeperConnection e si aggiunge un metodo connect. Il metodo connect crea un oggetto ZooKeeper, si connette a ZooKeeper e quindi restituisce l'oggetto.

Qui CountDownLatch viene utilizzato per fermare (aspettare) il processo principale fino a quando il client non si connette con ZooKeeper.
ZooKeeper risponde allo stato della connessione tramite la richiamata di Watcher. Il callback di Watcher verrà chiamato una volta che il client si connette con ZooKeeper e il callback di Watcher chiama il metodo countDown di CountDownLatch per rilasciare il blocco, wait nel processo principale.

Create

La classe ZooKeeper fornisce il metodo create per creare un nuovo znode nell'ensemble ZooKeeper.

create(String path, byte[] data, List<ACL> acl, CreateMode createMode)

Dove:

1. path - percorso Znodo;
2. data - dati da memorizzare in un percorso znodo specificato;
3. acl – lista di controllo accessi del nodo da creare. L'API ZooKeeper fornisce un'interfaccia statica ZooDefs.Ids per ottenere alcuni elenchi ACL di base. Ad esempio, ZooDefs.Ids.OPEN_ACL_UNSAFE restituisce un elenco di acl per gli znodi aperti;
4. createMode - il tipo di nodo, effimero, sequenziale o entrambi. Questo è un enum.

Viene creata una nuova applicazione Java per verificare la funzionalità di creazione dell'API ZooKeeper. Nel metodo principale viene creato un oggetto di tipo ZooKeeperConnection e chiama il metodo connect per connettersi a ZooKeeper.
Il metodo connect restituirà un oggetto ZooKeeper che verrà poi composto con percorso e dati personalizzati.

Exists

La classe ZooKeeper fornisce il metodo exists per verificare l'esistenza di uno znode. Se lo znode specificato esiste restituisce i metadati di uno znode.

exists (percorso stringa, osservatore booleano)

Dove:

1. path – percorso Znode
2. watcher - valore booleano per specificare se guardare o meno uno znode specificato

Viene creata una nuova applicazione Java per verificare la funzionalità "exists" dell'API ZooKeeper. Crea un file "ZKExists.java". Nel metodo principale, crea l'oggetto ZooKeeper, "zk" usando l'oggetto "ZooKeeperConnection". Quindi, chiama il metodo "exists" dell'oggetto "zk" con il "percorso" personalizzato

getData

La classe ZooKeeper fornisce il metodo getData per ottenere i dati allegati in uno znode specificato e il suo stato

getData(String path, Watcher watcher, Stat stat)

Dove:

1. path - percorso Znode
2. watcher – Funzione di callback di tipo Watcher. ZooKeeper avviserà tramite la richiamata di Watcher quando i dati dello znode specificato cambiano. Questa è una notifica una tantum.
3. stat - Restituisce i metadati di uno znode.

Viene creata una nuova applicazione Java per comprendere la funzionalità getData dell'API ZooKeeper. Crea un file ZKGetData.java. Nel metodo principale, crea un oggetto ZooKeeper zk usando l'oggetto ZooKeeperConnection. Quindi, chiama il metodo getData dell'oggetto zk con un percorso personalizzato.

setData

La classe ZooKeeper fornisce il metodo setData per modificare i dati allegati in uno znode specificato

setData(String path, byte[] data, int version)

Dove:

1. path – percorso Znode

2. data - dati da memorizzare in un percorso znode specificato.

3. version - Versione corrente dello znode. ZooKeeper aggiorna il numero di versione dello znode ogni volta che i dati vengono modificati.

Viene creata una nuova applicazione Java per comprendere la funzionalità setData dell'API ZooKeeper. Crea un file ZKSetData.java. Nel metodo principale, crea un oggetto ZooKeeper zk usando l'oggetto ZooKeeperConnection. Quindi, chiama il metodo setData dell'oggetto zk con il percorso specificato, i nuovi dati e la versione del nodo.

getChildren

La classe ZooKeeper fornisce il metodo getChildren per ottenere tutti i sottonodi di un particolare znode

getChildren(String path, Watcher watcher)

Dove:

1. path - percorso Znode

2. watcher – Funzione di callback di tipo "Watcher". ZooKeeper avviserà quando lo znode specificato viene eliminato o un bambino sotto lo znode viene creato/cancellato. Questa è una notifica una tantum.

Delete

La classe ZooKeeper fornisce il metodo delete per eliminare uno znode specificato

delete(String path, int version)

Dove:

1. path - percorso Znode

2. version - Versione corrente dello znode.

Viene creata una nuova applicazione Java per comprendere la funzionalità di eliminazione dell'API ZooKeeper. Crea un file ZKDelete.java. Nel metodo principale, crea un oggetto ZooKeeper zk usando l'oggetto ZooKeeperConnection. Quindi, chiama il metodo delete dell'oggetto zk con il percorso e la versione specificati del nodo.

ARCHITETTURA

Come anticipato nell'introduzione ZooKeeper permette di creare un servizio centralizzato al quale si connettono diversi client che possono creare, rimuovere o modificare uno znode. L'architettura si focalizzerà sulla struttura e sulle proprietà dei due componenti fondamentali: Server e Client.

3.1 Architettura server ZooKeeper

ZooKeeper mette a disposizione dell'utente un server già implementato in grado di gestire le connessioni con i client. Il server, chiamato ZKServer, è collegato ai package inseriti nella cartella /lib del servizio zookeeper.
Come anticipato il server mette a disposizione dell'utente la radice di un albero che potrà essere suddiviso in tanti sottoalberi con diverse funzioni. In particolare, si è pensato che per gestire al meglio i client connessi al servizio, il server debba essere suddiviso nei seguenti percorsi:

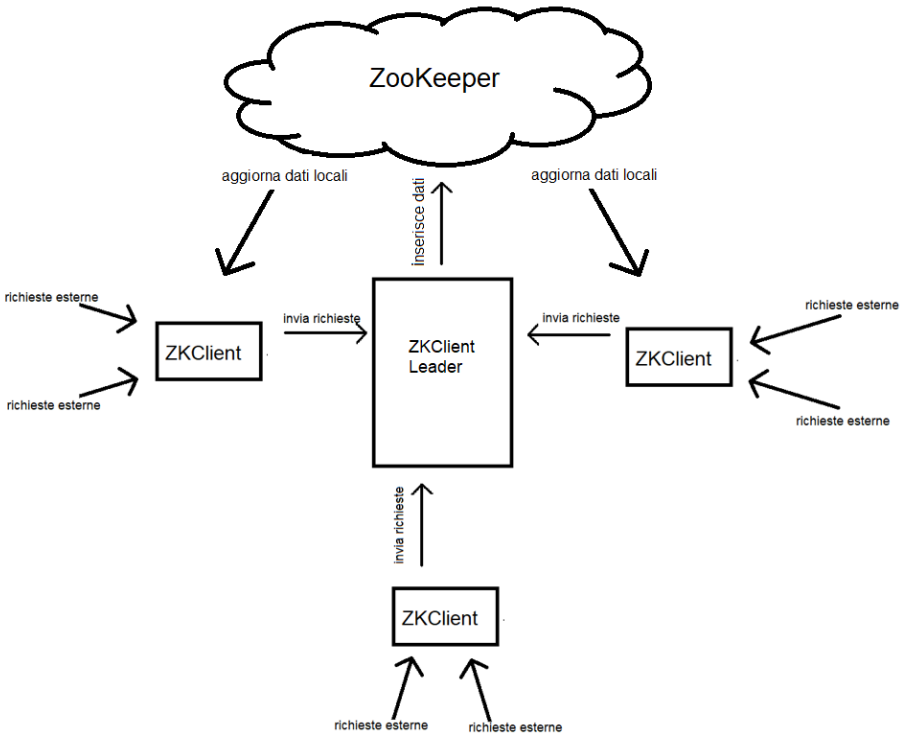
1. All nodes, in questo sottoalbero sarà presente la liste di tutti i client che si sono connessi al servizio dal primo avvio, mantenendone uno storico in "memoria";
2. Live nodes, è il percorso dove saranno presenti tutti gli ZKClient connessi attualmente al servizio in attesa di eseguire comandi;
3. Election, in questa directory verrà eletto il leader del servizio fra tutti i client connessi, ovvero colui che dovrà gestire tutti gli aggiornamenti e le richieste provenienti dagli altri client;
4. Lock, questo znode conterrà la lista dei client che vogliono aggiornare il leader con le richieste ricevute dall'ambiente esterno.

3.2 Architettura client ZooKeeper

Il client ZooKeeper, chiamato ZKClient, dovrà essere sviluppato sfruttando le API java e avrà tutte le funzionalità ottenibili tramite linea di comando.
I client sono stati pensati per essere divisi in due tipi:

1. Leader;
2. Slave;

Lo slave riceverà comandi dall'ambiente esterno e per coordinare le operazioni di inserimento, modifica o cancellazione dei dati dovrà reindirizzarli al leader del servizio che provvederà ad eseguirli in ordine di ricezione.
Per ogni richiesta eseguita il leader dovrà aggiornare i dati degli altri ZKClient connessi al servizio, ovvero quelli presenti nel percorso /live_nodes.
Di seguito un immagine riassuntiva dell'architettura descritta:



Questo tipo di architettura leader/slave unita a ZooKeeper tende ad eliminare l'inconsistenza dei dati in quanto ogni qualvolta che uno slave vorrà inserire o modificare dei dati nel servizio dovrà semplicemente comunicarlo al leader che prenderà in carico la gestione delle richieste, le elaborerà secondo un ordine prestabilito e provvederà ad aggiornare tutti gli ZKClient connessi.
Il problema da tenere conto in fase di implementazione è quello di evitare un collo di bottiglia nell'aggiornamento degli slave da parte del leader. Se il sistema fosse di grandi dimensioni potrebbe crearsi una congestione nell'elaborazione delle richieste dei client.

IMPLEMENTAZIONE

Basandosi sull'architettura appena illustrata si è pensato di fare di sviluppare un'applicazione distribuita in grado di gestire i dati di persone, conservandone nome, cognome, indirizzo e età. L'implementazione prenderà in considerazione client e server ZooKeeper.

4.1 Implementazione server ZooKeeper

Lo ZKServer come anticipato nell'architettura verrà suddiviso nei seguenti percorsi:

1. /all_nodes, in questo percorso verrà mantenuto lo storico degli ZKClient che si sono connessi al servizio;
2. /live_nodes, serve per mantenere i dati degli ZKClient connessi al servizio in un preciso istante;
3. /election, all'interno della quale attraverso un semplice algoritmo gli ZKClient capiranno chi è il leader;
4. /lock, che conterrà una coda di tipo FIFO dove ogni slave si metterà in attesa di comunicare con il leader se ha ricevuto una richiesta esterna;
5. /config/data, conterrà i dati relativi al servizio, una lista di persone;
6. /config/done, questo znode servirà a tutti gli ZKClient successivi per capire se la configurazione dell'ambiente è stata effettuata.

Gli znode appena illustrati saranno d'appoggio ai vari ZKClient per gestirsi all'interno del servizio e sincronizzare i loro dati.
Una volta creato l'albero di directory il server rimarrà in attesa di connessioni dai client e gestirà tutte le richieste inoltrate dal leader del servizio.

4.2 Implementazione client ZooKeeper

Una volta configurato l'ambiente, ogni ZKClient può collegarsi ed eseguire le seguenti operazioni:

1. Inserisce uno znode persistente dentro al percorso "/all_nodes", questo nodo creato conterrà la data e l'ora della prima connessione al servizio;
2. Inserisce uno znode effimero dentro al percorso "/live_nodes" in questo modo il client sarà in grado di ricevere tutti gli aggiornamenti dei dati e degli stati del servizio. Nel caso la connessione tra client e ZKServer viene interrotta lo znode verrà cancellato, data la sua natura effimera, e lo ZKClient non sarà più in grado di riceverei aggiornamenti e comunicare con il servizio;

3. lo ZKClient inserisce nel percorso "/election" uno znode effimero sequenziale denominato "node-00000X" dove X è un numero incrementale il cui conteggio è mantenuto dal server ZooKeeper. La funzionalità di questo nodo è molto simile alla precedente ma si differenzia nel caso in cui lo znode associato al client venga eliminato. In questo nodo viene eletto il leader della rete utilizzando una coda di tipo FIFO, dove il primo sarà eletto a capo del servizio e nel caso la sua connessione venga interrotta, lo znode successivo contenente i dati del nuovo ZKClient sarà eletto il nuovo leader.

4. "/lock" a differenza dai precedenti viene utilizzato nel caso in cui un client riceva una richiesta dall'esterno. In particolare, verrà creato uno znode effimero sequenziale all'interno del percorso e inserito all'interno di una coda FIFO, dove il primo della lista avrà la possibilità di comunicare la propria richiesta al leader, che provvederà ad aggiornare i dati sul server e di conseguenza tutti gli altri ZKClient connessi al servizio.La denominazione degli znode sequenziali sarà "node-0000X" dove X è un contatore che viene incrementato ogni volta che viene creata una cartella figlio in questo nodo.

Quando lo ZKClient è connesso al servizio ed è sincronizzato, crea dei watch persistenti sui seguenti percorsi:

1. all_nodes, in questo percorso il watch lancia un evento quando lo storico degli ZKClient viene modificato;

2. live_nodes. il watch collegato a questo percorso serve a controllare quali client sono connessi al servizio, se viene aggiunto o modificato uno znode il watch avviserà il client, che provvederà ad aggiornare la sua lista locale contenente i client connessi;

3. election, il watch settato in questo percorso ha la funzione di aggiornare il leader. Se quest'ultimo perde connessione al servizio i watch settati dagli altri client provvederanno ad aggiornare il loro stato e ricalcoleranno il capo del servizio. Se un client slave perde la connessione, verrà lanciato un eventi dai watch degli altri client che aggiorneranno semplicemente la lista FIFO;

4. lock, quando un client vuole inviare una richiesta al leader ed è in attesa nella coda FIFO setterà un watch su questo znode. Ogni volta che un client esegue la sua richiesta, cancella il proprio znode e i watch degli altri client lanceranno un evento per il quale ogni ZKClient calcolerà se sarà il prossimo ad essere servito.

4.3 Implementazione algoritmo elezione leader

Nello znode "/election" ogni ZKClient inserisce una cartella del tipo node-0000X, ad esempio il primo client connesso avrà /node-0000001, il successivo /node-0000002 e così via. Il contenuto del percorso /election può essere visto come una coda di tipo FIFO dove il primo è quello con numero sequenziale più basso.Si è pensato di implementare il seguente algoritmo espresso in pseudocodice:

Algorithm 1 GetLeaderNodeData()

if *exists/election* **then**
  $res \leftarrow get(nodeInElection)$
  $order(res)$
  $first \leftarrow getFirst(res)$
  $data \leftarrow getData(first)$
end if

Pseudocodice algoritmo elezione, implementato in ZKClientImpl.java – metodo getLeaderNodeData()

Inizialmente lo ZKClient controlla che il nodo "/election" esista, nel caso ci sia:

1. Viene presa la lista dei nodi figli presenti all'interno del percorso;
2. Il risultato del passo precedente viene ordinato in ordine crescente, in modo tale da avere come primo elemento, il primo ZKClient che ha creato uno znode;
3. Il nome ottenuto è del tipo "node-00000X", per poter risalire al client a cui è collegato si esegue una funzione per ottenere i dati del nodo nella forma: Indirizzo IP : Porta.

Nel caso il nodo "/election" non esista il seguente algoritmo non viene eseguito.Inoltre il client fissa un watch nel percorso dell'elezione in modo tale che se il leader dovesse interrompere la connessione, ogni client nella rete verrebbe avvisato e provvederebbe a ricalcolare il leader.

4.4 Gestione richieste

Ogni ZKClient è in grado di ricevere delle richieste dall'esterno attraverso l'utilizzo di socket e si possono verificare due casi:

1. Richiesta a slave;
2. Richiesta a leader.

Richiesta a slave

Una volta che lo slave riceve una richiesta, crea una cartella nel percorso "/lock", vi associa un watch e calcola il prossimo client che deve essere servito. La gestione della lista che si creerà all'interno della cartella sarà fatta attraverso una coda di tipo FIFO, dove il primo ad entrare in coda sarà anche il primo ad essere servito dal leader del servizio.
Se è il prossimo, provvederà ad inoltrare la richiesta al leader, altrimenti si metterà in attesa di essere servito. Grazie al watch ad ogni modifica della cartella "/lock" verrà lanciato un evento che farà ricalcolare ad ogni client chi è il successivo slave che potrà comunicare con il leader.
Di seguito l'algoritmo in pseudo codice per ottenere il prossimo client ad essere servito:

Algorithm 2 GetNextServed()

```
if exists/lock then
    res ← get(nodesInLock)
    order(res)
    first ← getFirst(res)
    data ← getData(first)
end if
```

Pseudocodice gestione del prossimo servito implementato in ZKClientImpl.java – metodo getNextServed()

Di seguito la gestione dei watch del client:

Algorithm 3 setWatch()

```
Require: watch
  path ← process(event)
  if path =' /election' then
    setwatch('/election')
  else if path =' /lock' then
    setwatch('/lock')
  end if
  if path =' /election' then
    insertWatch('/election', PERSISTENT)
    getLeader()
    if amILeader then
      iAmLeader ← true
    end if
  else if path =' /lock' then
    if AmINextServed then
      iAmNext ← true
    else
      insertWatch('/lock', PERSISTENT)
    end if
  end if
```

Pseudocodice gestione dei watch associati a "/lock" e "/election" implementato in ZKClientImpl.java metodo setWatch()

Richiesta a leader

Quando una richiesta viene ricevuta dal leader del servizio ZooKeeper, quest'ultimo provvederà semplicemente ad aggiornare i dati delle persone all'interno del percorso "/config/data". Ogni ZKClient, avendo predisposto un servizio di watch sui dati del servizio, riceverà la notifica della modifica effettuata dal leader e provvederà a sincronizzare i propri dati locali con quelli aggiornati.
La seguente gestione è stata pensata per evitare un collo di bottiglia nell'esecuzione delle richieste inoltrate al leader. In questo modo anche nel caso di un sistema distribuito di grandi dimensioni, non sarà il leader ad aggiornare tutti i client uno ad uno ma provvederà semplicemente ad aggiornare le risorse nel percorso specifico e a quel punto tutti gli slave connessi al servizio provvederanno ad aggiornare i loro dati e imposteranno un nuovo watch nello stesso percorso.

4.5 Spegnimento client ZooKeeper

Quando uno ZKClient viene disconnesso dal servizio vengono cancellati:

1. Tutti i watcher che sono stati predisposti dal client;
2. Tutti gli znode effimeri che sono stati creati durante l'esecuzione dello ZKClient, ovvero quelli in "/lock" se presenti, "/election" e "/live_nodes".

Si può verificare la situazione in cui lo ZKClient è il leader del servizio ed eliminando il nodo da "/election" verranno lanciati gli eventi di tutti i watch associati a quel percorso e ognuno provvederà a calcolare il nuovo leader del servizio.
Nel caso in cui invece il client è semplicemente uno slave, verranno aggiornati i watch nei percorsi "/election" e "/live_nodes".

TESTING

Il testing è stato effettuato utilizzando il framework di testing junit. Inizialmente verranno creati gli ZKClient che si connetteranno al servizio creando l'ambiente che gestirà ogni richiesta. Durante il testing i client verranno interrogati per capire se effettivamente i dati sono sincronizzati alle operazioni che vengono effettuate. I test principali riguardano:

1. Verifica del leader, ovvero controllare che ogni ZKClient sia sincronizzato sui dati aggiornati del leader;
2. Verifica dei client attivi, cioè prendere in esame il nodo "live_nodes" e controllare che i nodi presenti al suo interno siano effettivamente tutti i client attualmente connessi al servizio;
3. Verifica dei dati salvati, i dati gestiti dal servizio devono essere sempre sincronizzati al dato più recente.

Per verificare queste condizioni vengono utilizzate le condizioni:

1. assertFalse, verifica che la condizione inserita sia uguale a false;
2. assertTrue, controlla che una condizione sia vera;
3. assertEquals, viene utilizzato per il confronto dei dati.

CONCLUSIONI

Abbiamo trovato la tecnologia Apache ZooKeeper molto interessante, siamo riusciti rapidamente a mettere su un sistema distribuito e inoltre abbiamo capito le basi della materia.
Per quanto le sue API risultano tanto semplici quanto versatili possono nascere dei problemi. Prima di tutto, ZooKeeper non deve essere pensato come esistente su un unico nodo bensì su più nodi, con poi la nascita di problemi di gestione della replicabilità e sincronizzazione. Inoltre, l'elezione del leader è un problema molto più complesso se calato in un contesto di sistema distribuito. Infine, ci sono anche altre situazioni da tenere monitorate: ad esempio in fase di distribuzione di un sistema con ZooKeeper, cercando di avere un numero di znode dispari proprio per evitare problemi di pareggio per leader election o lock di risorse. Nel caso in cui parte della rete viene isolata, e quindi znode isolati, bisogna evitare che nelle parti isolate vengano eletti due leader distinti che quindi creerebbero delle situazioni di inconsistenza non appena tutta la rete torna disponibile. All'occhio dello sviluppatore però, questa complessità nascosta proprio grazie a queste API.
Insomma, è sicuramente una tecnologia a cui fare riferimento quando si ha bisogno di sincronizzare grandi sistemi distribuiti ed è per questo motivo per cui grandi società come Rackspace, Yahoo!, ebay e Solr ne fanno grande uso.