Un archivio chiave-valore distribuito e affidabile per i dati critici di un sistema distribuito: etcd

Mattia Achilli
mattia.achilli@studio.unibo.it

Gennaio 2020

Il progetto nasce con lo scopo di studiare, analizzare e testare la tecnologia **etcd**. La tecnologia etcd consiste in un archivio distribuito chiave-valore **open source** che fornisce supporto alla memorizzazione dei dati in un sistema distribuito o cluster di macchine. Le applicazioni del cloud possono leggere e scrivere su etcd, mantenere tempi di attività più coerenti e continuare a funzionare, anche a fronte di guasti dei singoli server. etcd è un componente fondamentale di molti progetti come, ad esempio, il datastore principale di **Kubernetes**, la piattaforma standard de facto per l'orchestrazione dei container.

Indice

1 Obiettivi del progetto **4**
- 1.1 Piano di lavoro . . . 4
- 1.2 Terminologia . . . 4

2 Introduzione a etcd **5**
- 2.1 Cos'è etcd? . . . 5
- 2.2 Perché utilizzare etcd? . . . 5
- 2.3 Esempio di utilizzo con Docker . . . 6
- 2.4 L'algoritmo di consenso Raft . . . 7
- 2.5 Log Replication . . . 7
- 2.6 Leader Election . . . 13
- 2.7 Funzionalità di etcd . . . 18

3 Introduzione a Docker **20**
- 3.1 Cos'è Docker? . . . 20
- 3.2 Cosa sono i container? . . . 20
- 3.3 Docker Engine . . . 20
- 3.4 Per cosa può essere utilizzato Docker? . . . 21
- 3.5 Architettura di Docker . . . 21
- 3.6 Gli oggetti Docker . . . 22

4 Configurazione pre-test **23**
- 4.1 Ambiente e strumenti utilizzati . . . 23
- 4.2 Installare Go e etcd . . . 23
- 4.3 SSL/TLS cosa sono e come configurarli in etcd . . . 25
- 4.4 Installare Docker . . . 29

5 Testing **31**
- 5.1 Cluster etcd Bare metal . . . 31
- 5.2 etcdctl . . . 34
- 5.3 Monitorare lo stato del cluster . . . 34
- 5.4 Gestire le entry . . . 36
- 5.5 Gestire i nodi del cluster . . . 38
- 5.6 Snapshot . . . 41
- 5.7 Misurare le prestazioni di etcd . . . 42
- 5.8 Stress test . . . 43
- 5.9 etcd con container Docker . . . 46

6 etcd in Java **51**
- 6.1 Connettersi a etcd . . . 52
- 6.2 Gestione dei membri . . . 53
- 6.3 Stato e manutenzione del cluster . . . 55
- 6.4 Gestire lo spazio di chiavi . . . 56

6.5 Watch sulle entry 58
6.6 Stress test in Java 59

7 etcd vs Zookeeper **63**
7.1 Analisi finale di etcd 63
7.2 Zookeeper 63
7.3 ZNodes 64
7.4 Zookeeper Watches 64
7.5 Vantaggi e svantaggi di Zookeeper 64

8 Conclusioni **65**

9 Sviluppi futuri **65**

Bibliografia **65**

1 Obiettivi del progetto

Il progetto si basa su due principali scenari, uno teorico e uno di test della tecnologia: nello scenario teorico verrà approfondita la tecnologia, il suo funzionamento, gli algoritmi e i possibili scenari di utilizzo, in quello pratico verrà toccata con mano la tecnologia e il suo funzionamento attraverso test.

Gli obiettivi si possono riassumere nei seguenti punti:

- Studiare approfonditamente la tecnologia con il supporto di articoli scientifici, spiegandone le caratteristiche in maniera semplice ma approfondita.
- Concentrarsi sul funzionamento dell'algoritmo di consenso **Raft**, fondamentale in etcd.
- Creare degli scenari di test per l'utilizzo della tecnologia utilizzando anche **Docker**.
- Rilevare attraverso i test i punti forti e carenti della tecnologia, verificando se quest'ultima si comporta come promesso anche in situazioni al di fuori della zona di comfort.
- Sottolineare infine differenze con tecnologie simili.

1.1 Piano di lavoro

Riassumendo gli obiettivi, il piano di lavoro si concretizza in queste due fasi:

- **Fase teorica**: studiare e analizzare la tecnologia con il supporto di articoli scientifici.
- **Fase pratica**: realizzare ambienti di test con il supporto (laddove necessario) di altre tecnologie, con lo scopo di spingere la tecnologia al di fuori della sua zona di comfort.

1.2 Terminologia

In questa sezione viene data una breve panoramica della terminologia più utilizzata durante tutto lo svolgimento del progetto. I termini più utilizzati riguardano:

- **Cluster**: insieme di macchine/host connessi tramite la rete.
- **Nodo**: macchina/host elaborativa fisica o virtuale all'interno di un cluster. In questo progetto un nodo contiene un'istanza di **etcd**.
- **Client-Server**: architettura di rete nella quale genericamente un client (come un terminale ad esempio) si connette ad un server per la fruizione di un certo servizio.
- **Peer-to-peer**: architettura di rete informatica in cui i nodi non hanno una gerarchia sotto forma di client o server fissi, ma anche sotto forma di nodi detti **peer**. Nel cluster di nodi etcd, i nodi rappresentano i peer.

2 Introduzione a etcd

Il nome "etcd" ha origine da due idee, la cartella Unix **"/etc"** e i sistemi distribuiti **"d"**. La cartella "/etc" rappresenta il percorso in cui memorizzare i dati di configurazione per un singolo sistema, mentre etcd memorizza le informazioni di configurazione per i sistemi distribuiti su larga scala, da qui "etcd".

2.1 Cos'è etcd?

etcd [3] è un archivio **chiave-valore** distribuito open source scritto in **Go** [2], utilizzato per memorizzare e gestire le informazioni critiche necessarie ai sistemi distribuiti per continuare a funzionare. In particolare, gestisce i dati di configurazione, i dati di stato e i metadati per Kubernetes, la popolare piattaforma di orchestrazione dei container.

L'archivio etcd garantisce aggiornamenti automatici più sicuri, coordina le attività da eseguire sugli host e agevola la configurazione di reti overlay per i container.

Grazie a etcd le applicazioni possono garantire un uptime più costante e continuare a funzionare anche in caso di errore dei singoli server. Le applicazioni leggono e scrivono dati in etcd, il quale distribuisce i dati di configurazione garantendo ridondanza e resilienza nella la configurazione dei nodi.

2.2 Perché utilizzare etcd?

Non è un compito da poco fungere da backbone di dati che mantiene in esecuzione un carico di lavoro distribuito, inoltre etcd garantisce le seguenti proprietà:

- **Replicazione (Fully replicated)**: ogni nodo in un cluster etcd ha accesso all'archivio dati.
- **Disponibilità (Highly available)**: etcd è progettato per non avere un single point of failure e per tollerare guasti hardware e di rete.
- **Affidabilità (Reliably consistent)**: ogni dato rappresenta i dati più recenti in tutti i nodi.
- **Velocità**: utilizzando dischi allo stato solido (SSD), etcd garantisce 10.000 scritture al secondo .
- **Sicurezza**: etcd supporta l'autenticazione automatica del certificato del client TLS (Transport Layer Security) e SSL (Secure Socket Layer) opzionale. Poiché etcd archivia dati di configurazione altamente sensibili, gli amministratori dovrebbero implementare i controlli di accesso basati sui ruoli all'interno della distribuzione e assicurarsi che i membri del team, che interagisce con etcd, siano limitati al livello di accesso meno privilegiato necessario a svolgere il proprio lavoro.
- **Semplicità**: qualsiasi applicazione, dalle semplici app Web ai motori di orchestrazione di container complessi come Kubernetes può leggere o scrivere dati su etcd utilizzando strumenti **HTTP / JSON** standard.

2.3 Esempio di utilizzo con Docker

I container Docker semplificano la creazione di un nuovo server che si desidera avviare, come, ad esempio, un nuovo server per un servizio Web. Ciò comporta connettere tutti i server insieme tramite file di configurazione quando si avviano i container. Ciò è particolarmente problematico quando si dispone di un cluster in esecuzione esistente e si desidera aggiungere, ad esempio, un'altra istanza del server Web, senza fermare l'intero sistema per aggiungere un nuovo nodo server.

L'approccio più efficace è, quindi, il seguente: ogni volta che un nuovo server viene aggiunto, si registra il nodo in un registro come etcd. Altri server possono controllare i servizi appena creati e modificare la propria configurazione in modo che punti al nuovo servizio appena creato. Allo stesso modo, possono guardare i servizi che non servono più all'interno del sistema per rimuoverli.

Figure 1: Possibile utilizzo di etcd con containers Docker.

2.4 L'algoritmo di consenso Raft

etcd si basa sull'algoritmo di consenso **Raft** [5] per garantire la consistenza (**consistency**) dell'archivio dati su tutti i nodi di un cluster.

L'algoritmo Raft raggiunge questa coerenza tramite un nodo **leader** eletto, che gestisce la replica per gli altri nodi del cluster, chiamati **follower**.

Il leader accetta le richieste dei client, che poi inoltra ai nodi follower. Una volta che il leader si è accertato che la maggior parte dei nodi follower abbia memorizzato ogni nuova richiesta come un **record/entry** di log, applica il record alla sua macchina a stati locale e restituisce il risultato di tale esecuzione, al client.

Se uno dei follower si blocca o i pacchetti di rete vengono persi, il leader riprova finché tutti i follower non hanno memorizzato tutti i record di log.

Se un nodo follower non riesce a ricevere un messaggio dal leader entro un intervallo di tempo specificato, si elegge un nuovo leader. Il follower si dichiara candidato e gli altri follower votano per esso, o per qualsiasi altro nodo in base alla sua availability.

Una volta eletto, il nuovo leader inizia a gestire la replica e il processo si ripete.

Questo processo consente a tutti i nodi etcd di memorizzare le stesse informazioni replicate allo stesso modo.

2.5 Log Replication

In questa sezione viene mostrato in modo dettagliato il processo di memorizzazione delle informazioni utilizzato da Raft tra i vari nodi di un cluster denominato **Log Replication**.

Supponendo di avere un client e un nodo che funge (ad esempio) da database, il client manda un valore al database che lo memorizza.

Figure 2: Scenario base con un client e un nodo.

Con un nodo è molto semplice. Ora supponiamo di avere più di un nodo, costruendo un sistema distribuito di nodi:

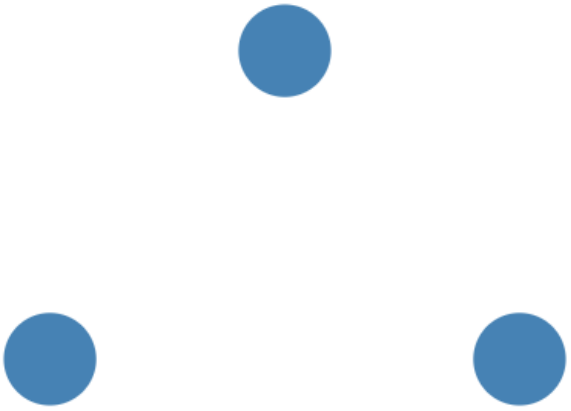


Figure 3: Scenario distribuito con più nodi.

I nodi possono assumere 3 diversi stati: **Leader**, **Candidate** e **Follower**.
Inizialmente tutti i nodi sono Follower.
Se i nodi non ricevono notizie da un leader, questi diventano candidati:

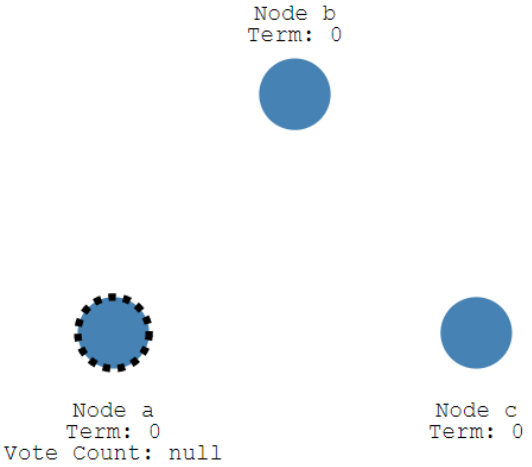


Figure 4: Il nodo A diventa un nodo candidato.

Il nodo candidato a questo punto richiede agli altri nodi follower di essere votato:

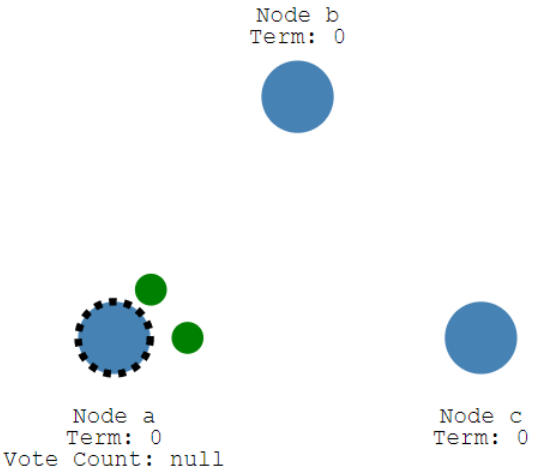


Figure 5: Il nodo A manda la richiesta agli altri nodi di essere votato.

I nodi una volta aver ricevuto la richiesta, rispondono con il loro voto:

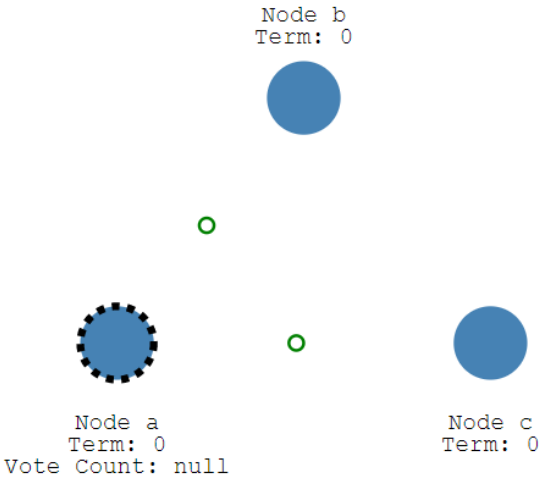


Figure 6: I nodi rispondono votando il nodo A.

A questo punto il nodo candidato che ha ricevuto più voti diventa leader (cambia di stato), questo processo viene chiamato: **Leader Election**.

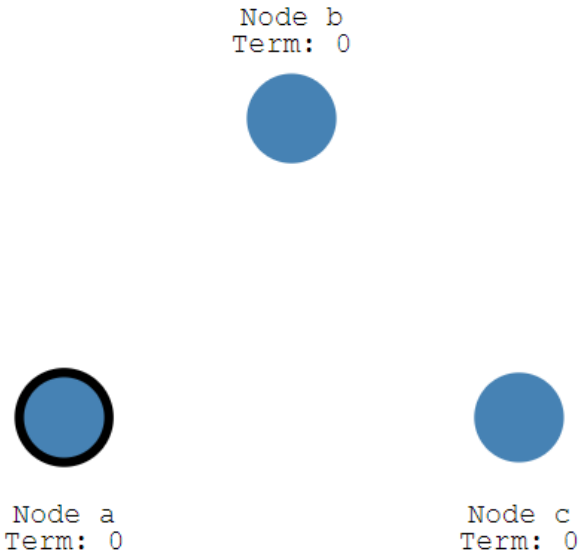


Figure 7: Il nodo A diventa un nodo leader.

Tutte le modifiche al sistema ora passano attraverso il leader.
A questo punto aggiungiamo nuovamente il nostro client di prima:

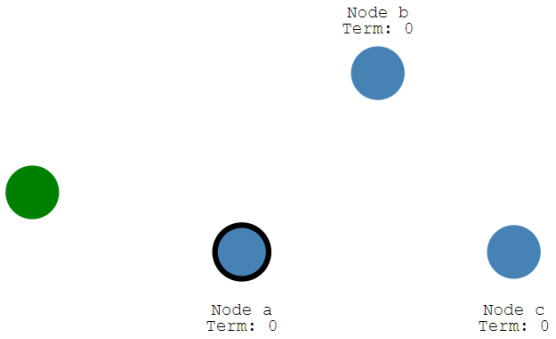


Figure 8: Il client viene aggiunto al sistema.

Come prima il client invia un valore al nodo Leader, ogni modifica viene aggiunta come **entry** (voce) nel registro del nodo Leader:

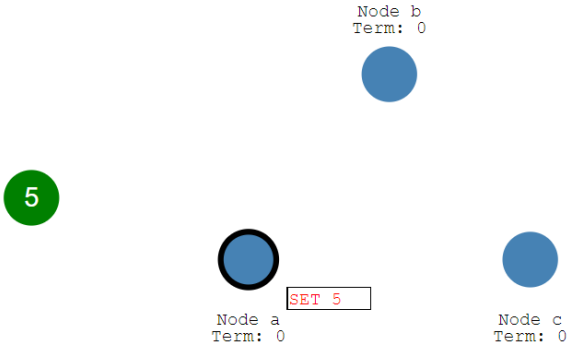


Figure 9: Il client invia un valore al nodo Leader.

La entry di log non viene subito salvata definitivamente.
Per salvare la entry, il nodo prima replica l'informazione sui nodi follower:

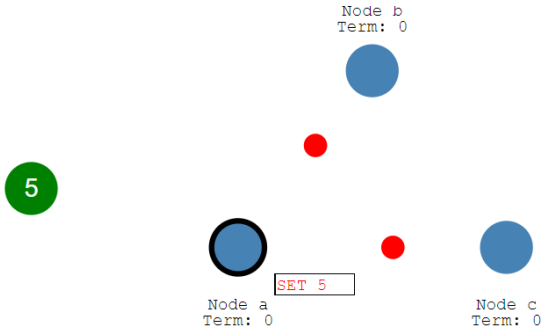


Figure 10: La modifica viene replicata ai nodi follower.

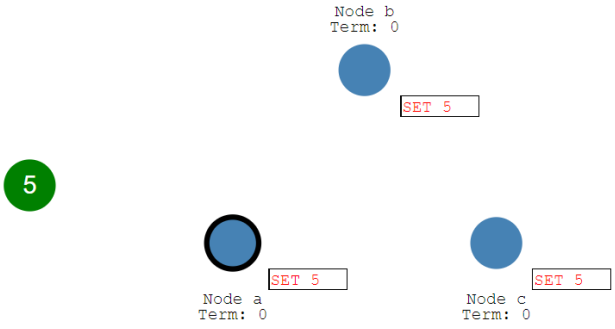


Figure 11: I nodi ricevono la entry.

Dopodichè il nodo Leader attende che la maggioranza dei nodi abbia ricevuto la entry, non ancora salvata:

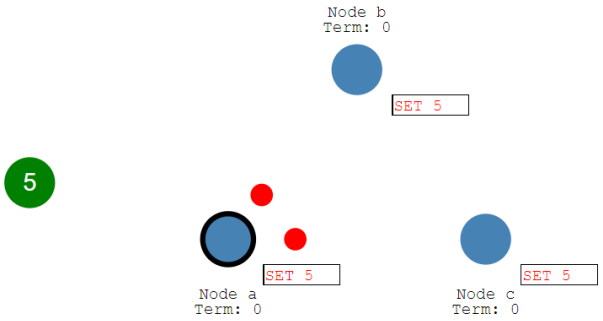


Figure 12: I nodi follower avvisano il Leader di aver ricevuto la entry.

Il nodo Leader sapendo che i nodi follower hanno ricevuto la entry salva la sua entry:

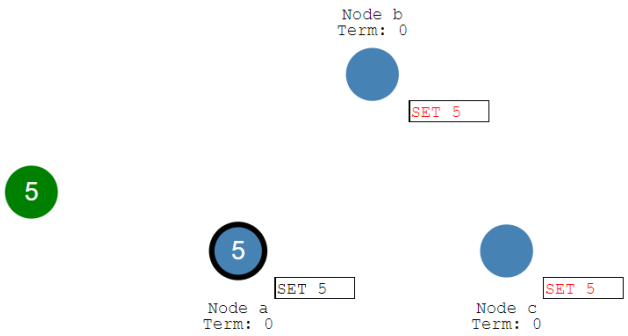


Figure 13: Il nodo leader salva la entry.

Il leader quindi notifica ai follower che la entry è stata salvata, i follower a loro volta salvano la loro entry:

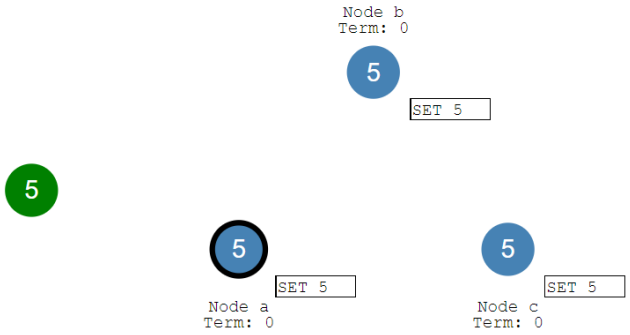


Figure 14: I follower salvano la entry dopo essere stati avvisati dal Leader.

Il cluster a questo punto ha raggiunto il consenso sullo stato del sistema!

N.B. le richieste di scrittura/modifica vengono gestite passando dal nodo Leader mentre le richieste di lettura possono essere gestite da qualsiasi nodo.

2.6 Leader Election

In questa sezione viene mostrato in modo dettagliato il processo di elezione del nodo Leader in un cluster di nodi.

In Raft ci sono due tipi di timeout che controllano le elezioni del nodo leader.

Il primo tipo è detto **election timeout**: tempo in cui un follower attende prima di diventare un candidato.

Solitamente l'election timeout ha un valore (generato casualmente) tra 150 ms e 300 ms.

Nella figura sottostante, il bordino intorno ai nodi rappresenta l'election timeout.

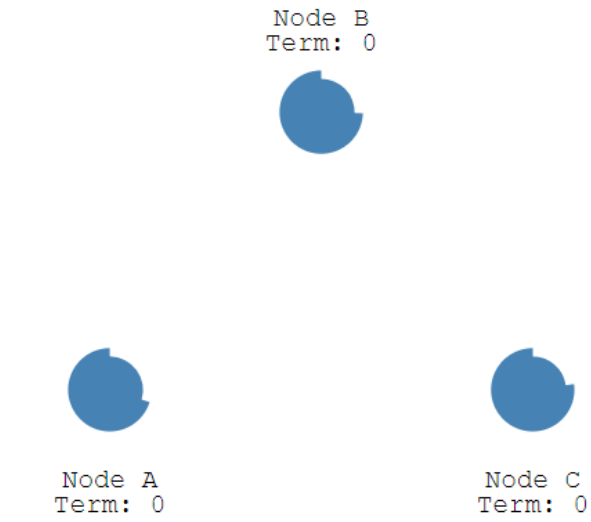


Figure 15: Election timeout per ogni nodo.

Dopo lo scadere del primo election timeout, il follower diventa un candidato e inizia un nuovo mandato elettorale votando per se stesso:

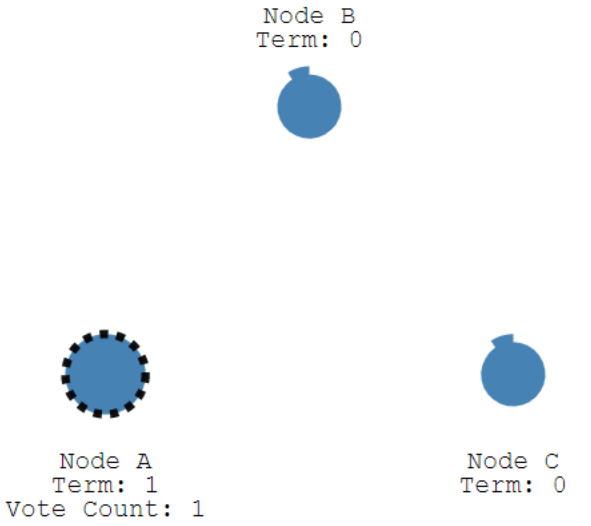


Figure 16: Dopo lo scadere dell'election timeout il nodo A diventa un candidato.

Il nodo diventato candidato manda la richiesta di essere votato agli altri nodi:

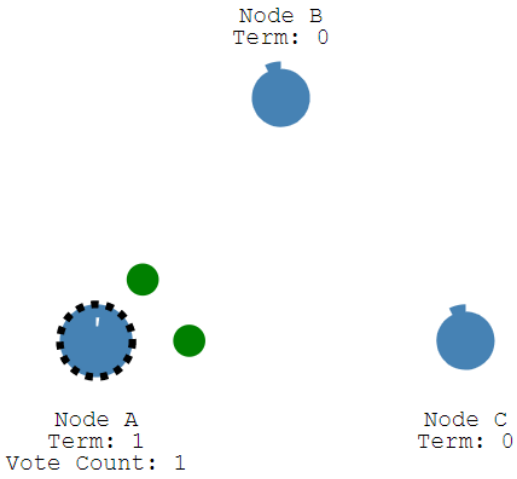


Figure 17: Il nodo candidato A manda la richiesta di essere votato.

I nodi che ricevono la richiesta di voto dal nodo candidato votano per il candidato:

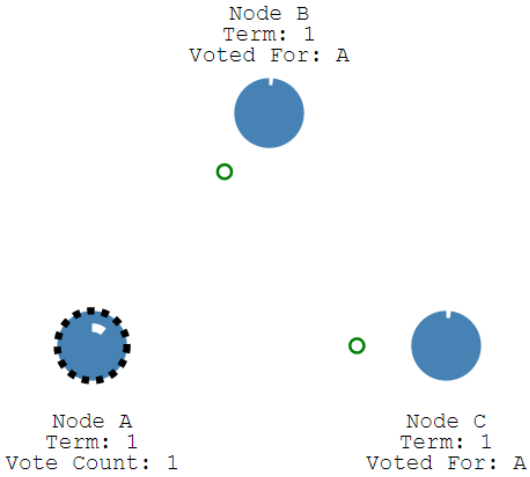


Figure 18: I nodi follower votano per il candidato.

Una volta ricevuto il voto il nodo candidato resetta il suo timer e diventa il Leader del sistema:

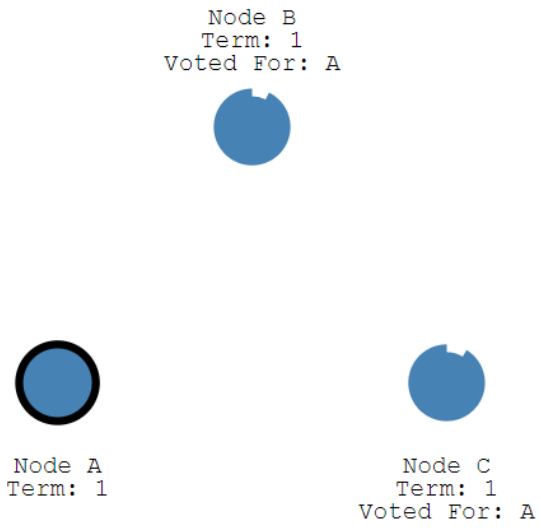


Figure 19: Il nodo candidato diventa Leader del sistema.

Una volta leader, il nodo inizia a inviare messaggi di aggiunta di entry ai suoi follower:

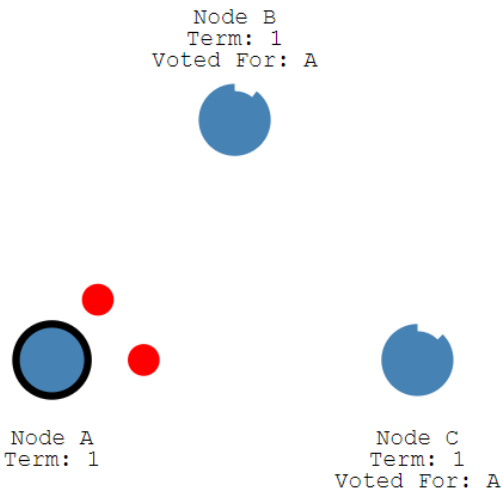


Figure 20: Il Leader invia messaggi ai follower.

I nodi followers rispondono al messaggio del Leader:

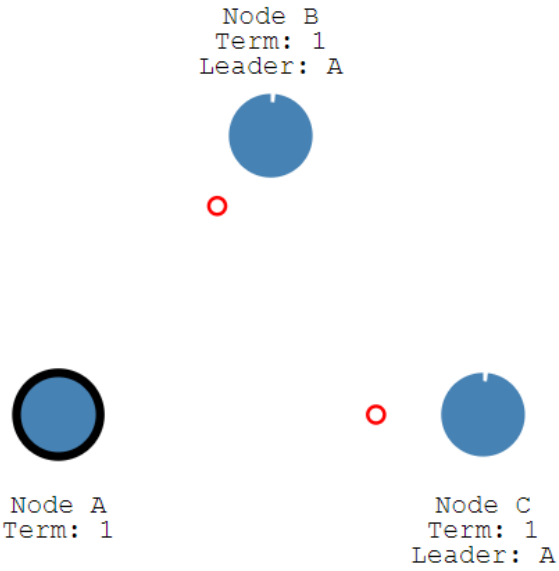


Figure 21: I followers rispondo al Leader inviando a loro volta un messaggio ciascuno.

Questi messaggi vengono inviati a intervalli specificati dal **heartbeat timeout**.

Questo scambio di messaggi continuerà fino a quando un follower non smetterà di ricevere heartbeat e diventerà un candidato.

Proviamo ad esempio a fermare l'attuale Leader:

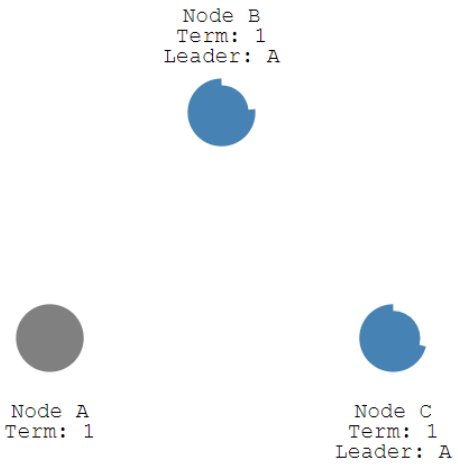


Figure 22: Il leader viene stoppato.

In seguito, con la stessa procedura di prima dell'election timeout, il nodo C si candida,

richiede i voti, viene votato e diventa nodo Leader:

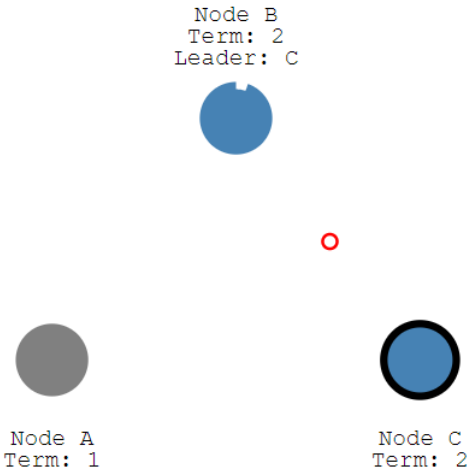


Figure 23: Il nodo C dopo essersi candidato ed essere stato votato diventa Leader.

Dopo ciò, si continua con il solito scambio di messaggi definiti dal heartbeat timeout.
Nota: se due nodi diventano candidati nello stesso momento, il candidato che riceve più voti diventa Leader.

2.7 Funzionalità di etcd

etcd mette a disposizione dell'utente che utilizza la tecnologia le seguenti funzionalità:

- **Creazione di un cluster**: creare un cluster di nodi che comunicano tra loro per mantenere le informazioni replicate. E' possibile configurare per ogni nodo informazioni come: nome, data directory in cui salvare i dati, heartbeat timeout, election timeout, indirizzi di ascolto per il client e per le comunicazioni peer-to-peer.
- **Autenticare le comunicazioni**: autenticare le comunicazioni client-server e peer-to-peer tramite crittografia **SSL/TLS**, approfonditi in seguito.
- **Gestire i nodi del cluster**: possibilità di aggiungere, rimuovere, aggiornare o visualizzare i nodi di un cluster etcd (per esempio tramite etcdctl che verrà mostrato in un secondo momento).
- **Verificare lo stato del cluster**: controllare lo stato e la "salute" di un insieme o di tutti i nodi del cluster.
- **Gestire le entry**: aggiungere, aggiornare, leggere, cancellare le entry replicate nei nodi del cluster.

- **Gestire gli snapshot del cluster**: il ripristino di un cluster richiede uno **snapshot** dello spazio delle chiavi da un nodo etcd. Lo snapshot permette di salvare l'attuale stato del cluster, che potrà essere ripristinato in caso di guasti.
- **Integrare etcd con container**: possibilità di utilizzare etcd con container **Docker**, aumentando la portabilità dello sviluppo iniziale.

In seguito verranno mostrate alcune **API** (Application Programming Interface), sia da linea di comando che da linguaggi di programmazione attraverso cui è possibile interfacciarsi con etcd. Inoltre verranno mostrate le funzionalità definite sopra a lato pratico.

3 Introduzione a Docker

Oggi, lo sviluppo di applicazioni richiede molto più della semplice scrittura del codice. L'esistenza di linguaggi, framework, architetture e interfacce tra gli strumenti per ogni fase del ciclo di vita creano, infatti, un'enorme complessità di sviluppo. Docker semplifica e accelera il flusso di lavoro, offrendo la libertà di innovare attraverso la scelta di strumenti, applicazioni e ambienti di distribuzione.

Docker verrà utilizzato in uno degli scenari di test di etcd in seguito.

3.1 Cos'è Docker?

Docker [4] è una piattaforma per lo sviluppo e l'esecuzione di applicazioni in container. Docker consente di separare le applicazioni dall'infrastruttura in modo da poter distribuire rapidamente il software.

3.2 Cosa sono i container?

Docker offre la possibilità di creare ed eseguire applicazioni "pacchettizzate" in un ambiente isolato chiamato **container**.

L'isolamento e la sicurezza consentono di eseguire molti container contemporaneamente su un determinato host.

I container sono leggeri, perché non richiedono il carico aggiuntivo di un **hypervisor**, ma vengono eseguiti direttamente all'interno del kernel della macchina host. Ciò significa poter eseguire più container su diversi hardware utilizzando macchine virtuali.

Docker fornisce gli strumenti e una piattaforma per gestire il ciclo di vita dei container che permette di:

- Sviluppare applicazioni e i suoi componenti di supporto utilizzando i container.
- Utilizzare il container per la distribuzione e il test dell'applicazione.
- Distribuire l'applicazione nell'ambiente di produzione, come container o servizio orchestrato. Funziona sia se l'ambiente di produzione è un data center locale, un provider cloud o un ibrido dei due.

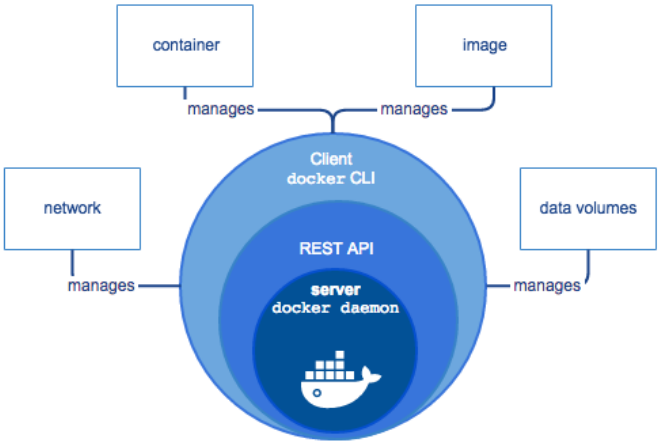
3.3 Docker Engine

Docker Engine è un'applicazione client-server con i seguenti componenti principali:

- **Un processo daemon**: server a lunga esecuzione (comando **dockerd**).
- **Un'API REST**: specifica le interfacce che i programmi possono utilizzare per parlare con il daemon e istruirlo su cosa fare.
- **CLI**: interfaccia a linea di comando (comando **docker**).

L'interfaccia a linea di comando (CLI) utilizza l'API REST Docker per interagire con il daemon Docker tramite script o comandi CLI diretti.

Figure 24: Struttura di Docker Engine



3.4 Per cosa può essere utilizzato Docker?

Docker può essere utilizzato per:

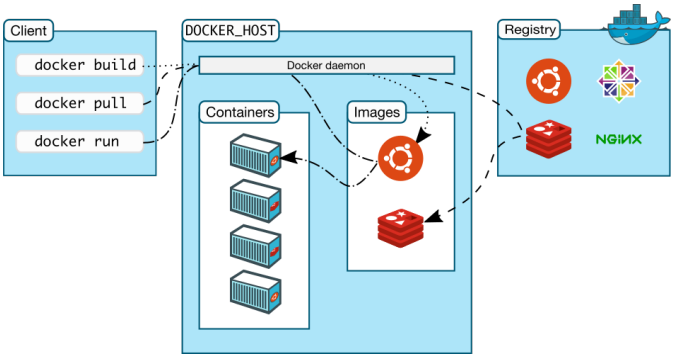
- semplificare il ciclo di vita dello sviluppo, consentendo a chi sviluppa di lavorare in ambienti standardizzati utilizzando container locali che forniscono applicazioni e servizi. I container sono ottimi per i flussi di lavoro di integrazione continua e distribuzione continua (**CI / CD**).
- consentire carichi di lavoro altamente portabili: i container Docker possono essere eseguiti su laptot, macchine fisiche, virtuali o nel cloud.
- fornire un'alternativa praticabile ed economica alle macchine virtuali basate su hypervisor, in modo da poter utilizzare una maggiore capacità di elaborazione per raggiungere gli obiettivi. Docker è perfetto per distribuzioni di piccole e medie dimensioni in cui è necessario fare di più con meno risorse.

3.5 Architettura di Docker

Docker utilizza un'architettura client-server. Il client Docker parla con il daemon Docker, che fa il lavoro di creazione, esecuzione e distribuzione dei container Docker. Il client e il daemon Docker possono essere eseguiti sullo stesso sistema oppure è possibile connettere un client Docker a un daemon Docker remoto. Il client e il daemon Docker comunicano utilizzando un'API REST, su socket UNIX o un'interfaccia di rete.

Il daemon Docker rimane in ascolto e gestisce oggetti Docker come: immagini, container, reti e volumi dati. Un daemon può anche comunicare con altri daemon per gestire i servizi Docker.

Figure 25: Architettura di Docker



Il client Docker è il modo in cui l'utente interagisce con Docker. Il client Docker può comunicare con più di un daemon.

Un registro Docker memorizza le immagini Docker. **Docker Hub** è un registro pubblico che chiunque può usare e Docker è configurato per cercare immagini su Docker Hub.

3.6 Gli oggetti Docker

Quando si utilizza Docker, si creano e utilizzano immagini, container, reti, volumi e altri oggetti:

- **Immagini**: un'immagine è un **template** di sola lettura con istruzioni per la creazione di un container Docker. Si possono creare le proprie immagini o usare solo quelle create da altri e pubblicate in un registro. Per costruire la propria immagine, si crea un **Dockerfile** con una semplice sintassi, utilizzata per definire i passaggi necessari per creare l'immagine ed eseguirla. Ogni istruzione in un Dockerfile crea un livello nell'immagine. Quando si modifica il Dockerfile e si ricostruisce l'immagine, vengono ricostruiti solo i livelli che sono stati modificati. Questo fa parte di ciò che rende le immagini così leggere, piccole e veloci rispetto ad altre tecnologie di virtualizzazione.

- **Container**: un container è un'istanza eseguibile di un'immagine. Si può creare, avviare, arrestare, spostare o eliminare un container utilizzando l'API Docker o la CLI. È possibile connettere un container a una o più reti, assegnare spazio di archiviazione o persino creare una nuova immagine in base al suo stato corrente. Per impostazione predefinita, un container è isolato dagli altri container e dalla sua macchina host. E' possibile controllare quanto sono isolati dalla rete, dallo spazio di archiviazione di altri container o dalla macchina host. Quando un container viene rimosso, tutte le modifiche del suo stato che non sono salvate nel disco, scompaiono.

- **Servizi**: i servizi consentono di scalare i container su più daemon Docker, che

lavorano tutti insieme come uno **swarm** con più **manager** e **workers**. Ogni membro di uno swarm è un daemon Docker e tutti i daemon comunicano utilizzando l'API Docker. Un servizio consente di definire lo stato desiderato, come il numero di repliche del servizio che devono essere disponibili in ogni momento. Di base, il servizio è con bilanciamento del carico su tutti i nodi di lavoro (workers).

4 Configurazione pre-test

Dopo aver introdotto e approfondito le tecnologie principali, in questa sezione verranno mostrati gli strumenti utilizzati e spiegati passo passo le installazioni di questi. Finita la parte di configurazione, si passerà alla parte di testing della tecnologia.

4.1 Ambiente e strumenti utilizzati

In questo caso il sistema operativo utilizzato è **Linux Lubuntu**, ma è possibile replicare le varie configurazioni e test utilizzando anche **MacOS** o **Windows** (altamente sconsigliato).

Prima di tutto è necessario scaricare i seguenti strumenti (le installazioni verranno spiegate passo dopo passo in seguito):

- **Docker**: installando Docker è possibile richiamare comandi Docker dalla bash.
- **Docker Desktop (opzionale)**: ambiente grafico alternativo e a supporto della linea di comando, permette di visualizzare graficamente i container che stanno eseguendo, fermarli, cancellarli, ecc... Il link per scaricare Docker Desktop è il seguente: `https://docs.docker.com/desktop/`.
- **etcd**: verranno mostrati i passaggi d'installazione. Il link di GitHub è il seguente: `https://github.com/etcd-io/etcd`
- **etcdctl**: client utile per interfacciarsi con il cluster etcd. Il link relativo al repository di GitHub è il seguente: `https://github.com/etcd-io/etcd/blob/master/etcdctl/README.md`.

Sarà necessario utilizzare una **Bash** per poter eseguire tutti i comandi utilizzati durante la configurazione e il test.

E' consigliato creare degli script **.sh** in cui inserire tutti i comandi necessari per eseguirli una sola volta.

4.2 Installare Go e etcd

Prima di passare all'installazione vera e propria di etcd, è necessario scaricare i file relativi al linguaggio di programmazione **Go** con cui è stato sviluppato etcd.

Lo script mostrato sotto permette di installare il linguaggio di programmazione Go, creare la variabile di ambiente **GOPATH** (utile per specificare il workspace) nel caso non sia già presente nel sistema e verificare la corretta installazione.

Lo script, in questo caso chiamato **downloadGo.sh**, è il seguente:

```
GO_VERSION=1.10.3 #Versione di Go da scaricare

#Elimina file e cartelle go recenti (se presenti)
sudo rm -f /usr/local/go/bin/go &&
sudo rm -rf /usr/local/go

#URL per il download di Go
DOWNLOAD_URL=https://storage.googleapis.com/golang

#Scarica e estrae i file relativi a go
sudo curl -s ${DOWNLOAD_URL}/go$GO_VERSION.linux-amd64.tar.gz |
sudo tar -v -C /usr/local/ -xz

# Controlla che la variabile di ambiente GOPATH sia
# gia' presente, se non lo e' la aggiunge
if grep -q GOPATH "$(echo ${HOME})/.bashrc"; then
    echo "bashrc already has GOPATH";
else
    echo "adding GOPATH to bashrc";
    echo "export GOPATH=$(echo ${HOME})/go" >> ${HOME}/.bashrc;
    PATH_VAR=$PATH":/usr/local/go/bin:$(echo ${HOME})/go/bin";
    echo "export PATH=$(echo $PATH_VAR)" >> ${HOME}/.bashrc;
    source ${HOME}/.bashrc;
fi

mkdir -p $GOPATH/bin/
go version #Versione di go
#Se da errore che non riconosce il comando go togliere il commento alla riga sotto
#export PATH=£PATH:/usr/local/go/bin
```

Per eseguire lo script è sufficiente dare i permessi al file attraverso **chmod +x nomefile** e, infine, eseguire lo script tramite **./nomefile.sh**.

Una volta scaricato e installato Go attraverso lo script riportato sopra, è il momento di installare etcd.

Anche qui è consigliato creare uno script .sh con, ad esempio, nome **downloadEtcd.sh**. Lo script permette di installare etcd di versione specificata nella variabile **ETCD_VER**, utilizzando come sorgente di installazione **Google** o **Github**. Scaricato ed estratto, etcd viene messo all'interno di **usr/local/bin** per permettere all'utente di richiamarlo come comando dalla bash.

Lo script è il seguente:

```
ETCD_VER=v3.3.8 #Versione di etcd da scaricare
```

```
# Utilizzare uno dei due link per scaricare etcd
GOOGLE_URL=https://storage.googleapis.com/etcd
GITHUB_URL=https://github.com/coreos/etcd/releases/download
DOWNLOAD_URL=${GOOGLE_URL} #In questo caso Google

#Elimina vecchi file o cartelle se presenti
rm -f /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz #
rm -rf /tmp/test-etcd && mkdir -p /tmp/test-etcd

#Scarica etcd e lo estrae
curl -L ${DOWNLOAD_URL}/${ETCD_VER}/
etcd-${ETCD_VER}-linux-amd64.tar.gz -o /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz
tar xzvf /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz -C
/tmp/test-etcd --strip-components=1

# Per utilizzare etcd esclusivamente dalla
propria working directory
#sudo cp /tmp/test-etcd/etcd* [WORKING_DIRECTORY]
sudo mkdir -p /usr/local/bin/ &&
#Per richiamare etcd dalla bash
sudo cp /tmp/test-etcd/etcd* /usr/local/bin/

etcd --version #Versione di etcd
#/tmp/test-etcd/etcd --version
```

Anche qui è sufficiente cambiare i permessi del file ed eseguirlo dalla bash.
Questi due file vanno eseguiti solo una volta per installare Go ed etcd.

4.3 SSL/TLS cosa sono e come configurarli in etcd

SSL (Secure Socket Layer) è una tecnologia, che garantisce la sicurezza di una connessione a Internet e protegge i dati sensibili scambiati fra due sistemi, impedendo ad un potenziale hacker di leggere e modificare le informazioni trasferite, che potrebbero comprendere anche dati sensibili. La comunicazione fra sistemi può riguardare un server e un client (o due server).

TLS (Transport Layer Security) è una versione aggiornata e più sicura di SSL.

Poichè etcd potrebbe memorizzare informazioni che non dovrebbero essere pubbliche, la crittografia è altamente raccomandata. etcd infatti supporta **SSL / TLS** e l'autenticazione per le comunicazioni client-server e peer-to-peer.

In questa sezione viene mostrato come generare certificati TLS autofirmati utilizzando **cfssl** e **cfssljson**. cfssl è sia uno strumento da riga di comando che un server API HTTP per la firma, la verifica e il raggruppamento di certificati TLS. La maggior parte dell'output di cfssl è in JSON. L'utilità cfssljson può prendere questo output e suddividerlo in file chiave, certificato, CSR e bundle separati, a seconda dei casi.

Innanzitutto, come già fatto in precedenza, va creato uno script .sh per installare cfssl e cfssljson:

```
#Rimuove cartelle e file gia' presenti di cfssl (se presenti)
rm -f /tmp/cfssl* && rm -rf /tmp/certs && mkdir -p /tmp/certs

#Scarica cfssl
curl -L https://pkg.cfssl.org/R1.2/cfssl_linux-amd64 -o /tmp/cfssl

#Scarica cfssl e lo mette in /tmp/cfssl
chmod +x /tmp/cfssl #Cambia i permessi a cfssl

#Sposta cfssl in /usr/local/bin/cfssl
sudo mv /tmp/cfssl /usr/local/bin/cfssl

#Ripete il procedimento con cfssljson
spostandolo infine in /usr/local/bin/cfssljson
curl -L https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64 -o /tmp/cfssljson
chmod +x /tmp/cfssljson
sudo mv /tmp/cfssljson /usr/local/bin/cfssljson

#Versione di cfssl
/usr/local/bin/cfssl version
#/usr/local/bin/cfssljson -h

#Crea infine la cartella /tmp/certs
mkdir -p /tmp/certs
```

Come effettuato in precedenza, è sufficiente eseguire lo script per scaricare e installare cfssl e cfssljson.

Ora è possibile generare i **CA (Certificate authority)** per la convalida delle comunicazioni tra client-server e peer-to-peer. In crittografia, un Certificate Authority è un soggetto terzo di fiducia, pubblico o privato, abilitato ad emettere un certificato digitale tramite una procedura di certificazione.

Viene utilizzata la crittografia a doppia chiave, o asimmetrica, in cui una delle due chiavi viene resa pubblica all'interno del certificato (chiave pubblica), mentre la seconda, univocamente correlata con la prima, rimane segreta e associata al titolare (chiave privata).

L'autorità dispone di un certificato con il quale sono firmati tutti i certificati emessi agli utenti, e ovviamente deve essere installata su una macchina sicura.

In questo caso il soggetto terzo di fiducia viene creato come nello script sotto.

Per creare un certificato CA realizzare uno script .sh con i seguenti comandi:

```
mkdir -p /tmp/certs
```

```
#All'interno di etcd-root-ca-csr.json mette tutto
il contenuto sino a EOF
#Dettagli della chiave, RSA di dimensione 2048 bit
#Nomi di Organization, Organization unit, City, State, Country
cat > /tmp/certs/etcd-root-ca-csr.json <<EOF
{
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "O": "etcd",
      "OU": "etcd Security",
      "L": "Cesena",
      "ST": "Italy",
      "C": "IT"
    }
  ],
  "CN": "etcd-root-ca"
}
EOF

#Genera il certificato a partire dal CSR creato sopra
cfssl gencert --initca=true
/tmp/certs/etcd-root-ca-csr.json
| cfssljson --bare /tmp/certs/etcd-root-ca

# Verifica che le informazioni del certificato
corrispondano con la chiave privata
openssl x509 -in /tmp/certs/etcd-root-ca.pem -text -noout


# Configurazione del certificato
cat > /tmp/certs/etcd-gencert.json <<EOF
{
  "signing": {
    "default": {
        "usages": [
          "signing",
          "key encipherment",
          "server auth",
          "client auth"
        ],
```

```
        "expiry": "87600h"
    }
  }
}
EOF
```

Eseguito lo script, è possibile generare, a partire dal root CA, i certificati e le chiave dei singoli nodi del cluster, permettendo loro di comunicare autenticandosi. Per ogni nodo **s1, s2...** deve essere richiamato lo script come: **./generateLocalCertificate.sh sN** (con N numero intero corrispondente al nodo del cluster); e allo stesso modo per ogni nodo che si vuole aggiungere lo script da utilizzare è il seguente:

```
mkdir -p /tmp/certs

#Se non viene inserito il numero del nodo
if [ "$#" == "0" ]; then
        echo "Inserire il nodo sN (con N numero intero)"
        exit 1
fi

#Come prima per il root CA
#Gli host vengono indicati come locali (localhost)
cat > /tmp/certs/$1-ca-csr.json <<EOF
{
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "O": "etcd",
      "OU": "etcd Security",
      "L": "Cesena",
      "ST": "Italy",
      "C": "IT"
    }
  ],
  "CN": "$1",
  "hosts": [
    "127.0.0.1",
    "localhost"
  ]
}
EOF
```

```
# Genera il certificato a partire dal:
# - CA (Certificate authority generato sopra): etcd-root-ca.pem
# - Chiave privata del CA: etcd-root-ca-key.pem
# - Configurazione del certificato: etcd-gencert.json
cfssl gencert \
  --ca /tmp/certs/etcd-root-ca.pem \
  --ca-key /tmp/certs/etcd-root-ca-key.pem \
  --config /tmp/certs/etcd-gencert.json \
  /tmp/certs/$1-ca-csr.json | cfssljson --bare /tmp/certs/$1

# Verifica il certificato come gia' visto prima
openssl x509 -in /tmp/certs/$1.pem -text -noout
```

N.B. lo script definito sopra va eseguito per ogni nodo che si vuole utilizzare all'interno del cluster di macchine etcd. Nel caso in cui si vogliano utilizzare, ad esempio, tre macchine, è necessario richiamarlo per **s1, s2 ed s3**.

4.4 Installare Docker

La via più semplice è quella di installare Docker dal sito ufficiale. E' possibile scaricarlo per i tre principali sistemi operativi, Linux, MacOS e Windows, ed il link è il seguente: `https://docs.docker.com/get-docker/`.

Per l'ambiente Linux è consigliato installare Docker direttamente dalla bash, aggiungendo come prima cosa il repository Docker attraverso i seguenti comandi:

- Aggiornare la lista dei packages con il comando

  ```
  sudo apt update
  ```

- Installare alcuni packages necessari per permettere ad apt di utilizzare i pacchetti su HTTPS

  ```
  sudo apt install apt-transport-https
  ca-certificates curl
  software-properties-common
  ```

- Aggiungere al sistema la chiave GPG dal Docker Repository:

  ```
  curl -fsSL https://download.docker.com/linux/ubuntu/gpg
  | sudo apt-key add -
  ```

- Aggiungere alle sorgenti APT il Docker Repository:

  ```
  sudo add-apt-repository
  "deb [arch=amd64]
  https://download.docker.com/linux/ubuntu bionic stable"
  ```

- Aggiornare il database dei packages con quelli di Docker appena aggiunti:

  ```
  sudo apt update
  ```

Una volta aggiunto il repository si può installare Docker utilizzando i comandi:

- Assicurarsi che l'installazione avvenga dal Repository di Docker e non da quello di Ubuntu con il comando:

  ```
  sudo apt-cache policy docker-ce
  ```

- Infine installare Docker:

  ```
  sudo apt install docker-ce
  ```

- Ora Docker è installato. Il daemon server dovrebbe essere in esecuzione e il processo abilitato ad avviarsi all'avvio della macchina. Puoi verificare che sia in esecuzione mediante il comando:

  ```
  sudo systemctl status docker
  ```

  L'output dovrebbe essere simile:

  ```
   docker.service - Docker Application Container Engine
  Loaded: loaded (/lib/systemd/system/docker.service; enabled;
  vendor preset: enabled)
  Active: active (running) since Wed 2021-02-10 11:04:26 CET; 2h 34min ago
    Docs: https://docs.docker.com
  Main PID: 830 (dockerd)
   Tasks: 10
  Memory: 135.1M
  CGroup: /system.slice/docker.service
          830 /usr/bin/dockerd -H
          fd:// --containerd=/run/containerd/containerd.sock
  ```

Come ultima cosa va aggiunto l'utente al gruppo Docker:

- Onde evitare di dover digitare il comando "sudo" ogni volta, è possibile aggiungere l'utente al docker-group, ossia ad un gruppo di utenti abilitati ad eseguire comandi come se fossero amministratori.

  ```
  sudo usermod -aG docker nomeutente
  ```

  N.B. Possono essere aggiunti altri utenti al docker-group semplicemente eseguendo questo comando per ogni utente che si vuole aggiungere.

- Per rieffettuare l'accesso ed applicare le modifiche:

  ```
  su - nomeutente
  ```

  Dopo aver eseguito questo comando va inserita la password dell'utente per continuare.

  In seguito verranno descritti i comandi Docker durante il loro utilizzo con etcd.

5 Testing

In questa sezione verranno fatti diversi esempi sull'utilizzo della tecnologia cercando di spiegare le funzionalità principali utilizzando in un primo momento l'API etcdctl. In seguito verrà mostrata anche un API in Java.

Infine, verranno creati degli scenari di test: in un primo momento etcd sarà utilizzato senza Docker, che sarà reintegrato in un secondo momento.

5.1 Cluster etcd Bare metal

etcd scrive i dati su disco. **Bare metal** o macchina virtuale è il modo più semplice per eseguire etcd.

Con bare metal si intende che il sistema dipende interamente dalla parte fisica della macchina, ovvero l'hardware. Il bare metal, quindi, non è nient'altro che un **Server dedicato**, cioè un computer le cui risorse sono messe a disposizione esclusivamente per un servizio. Si parla anche di un **Single-tenant server**. Il bare metal si differenzia dalle macchine virtuali, che sono separate sullo stesso hardware.

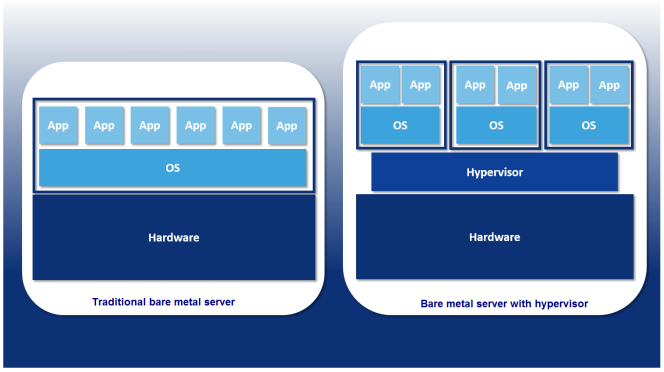


Figure 26: Sistema bare metal.

A questo punto è possibile eseguire il primo scenario di test, che comprende tre diversi nodi etcd (è sempre consigliato un numero dispari di nodi per il raggiungimento di un quorum dato da (n/2)+1 nodi). In questo caso per semplicità sono stati creati tre diversi script bash di configurazione per ognuno dei nodi da avviare. La struttura di ogni script si differenzia dagli altri per nome del nodo (s1, s2, ...), porta client (su localhost), porta peer e certificati di ogni nodo (se non si vuole utilizzare TLS nelle comunicazioni non è necessario metterli).

Per costruire un cluster di nodi etcd è necessario utilizzare il comando **etcd** seguito da un insieme di flag di configurazione.

Verrà mostrato solo lo script del primo nodo:

```
# Viene creata una cartella nella HOME
# dell'utente se inesistente
```

```
dove vengono spostati tutti i certificati
mkdir -p ${HOME}/certs
#cp /tmp/certs/* £{HOME}/certs

# Assicurarsi che etcd abbia i permessi a £HOME/certs
# Togli il commento sotto se il cluster e' nuovo
#rm -rf /tmp/etcd/s1

# Parte diversa di ogni nodo
# Esegue etcd
etcd --name s1 \
  --data-dir /tmp/etcd/s1 \
  --listen-client-urls https://localhost:2379 \
  --advertise-client-urls https://localhost:2379 \
  --listen-peer-urls https://localhost:2380 \
  --initial-advertise-peer-urls https://localhost:2380 \
  --initial-cluster s1=https://localhost:2380, \
  s2=https://localhost:22380, s3=https://localhost:32380 \
  --initial-cluster-token tkn \
  --initial-cluster-state new \
  --client-cert-auth \
  --trusted-ca-file ${HOME}/certs/etcd-root-ca.pem \
  --cert-file ${HOME}/certs/s1.pem \
  --key-file ${HOME}/certs/s1-key.pem \
  --peer-client-cert-auth \
  --peer-trusted-ca-file ${HOME}/certs/etcd-root-ca.pem \
  --peer-cert-file ${HOME}/certs/s1.pem \
  --peer-key-file ${HOME}/certs/s1-key.pem
```

I flag utilizzati per ogni nodo che si vuole avviare sono i seguenti:

- –**name**: nome del nodo, deve essere univoco all'interno del cluster.
- –**data-dir**: percorso dove vengono salvati i dati, in questo caso per ogni nodo è /tmp/etcd/sN.
- –**listen-client-urls**: elenco di URL su cui ascoltare il traffico client, utilizzando https (per via di TLS): https://localhost:portaClientSN.
- –**advertise-client-urls**: elenco degli URL dei client di questo cluster da annunciare al di fuori, uguale a sopra.
- –**listen-peer-urls**: dove si mette in ascolto per i peer, in questo caso https://localhost:portaPeerSN.
- –**initial-advertise-peer-urls**: elenco degli URL dei peer di questo cluster da annunciare al di fuori, uguale a sopra.

- –**initial-cluster**: configurazione iniziale del cluster all'avvio, vengono messi in questo caso s1, s2 e s3.
- –**initial-cluster-token**: token del cluster durante l'esecuzione, il token indica l'appartenenza di un nodo ad un cluster rispetto ad un altro.
- –**initial-cluster-state**: stato iniziale del cluster (new o existing), inizialmente new, quando si vuole aggiungere un nodo al cluster va messo existing.
- –**client-cert-auth**: abilita l'autenticazione del client tramite certificati.
- –**trusted-ca-file**: percorso del root CA (generato all'inizio).
- –**cert-file**: percorso al certificato TLS del nodo (sN).
- –**key-file**: percorso alla chiave privata del nodo (sN).
- –**peer-client-cert-auth**: abilita l'autenticazione del peer tramite certificati.
- –**peer-trusted-ca-file**: percorso del root CA (generato all'inizio).
- –**peer-cert-file**: percorso al certificato TLS del nodo (sN).
- –**peer-key-file**: percorso alla chiave privata del nodo (sN).

Per i nodi da aggiungere (in questo caso tre) è sufficiente:

- cambiare nome con sN.
- cambiare porte client, 22379 per s2, 32379 per s3.
- cambiare porte peer, 22380 per s2, 32380 per s3.
- cambiare con sN il nome dei certificati in fondo allo script.

Eseguito lo script in tre bash distinte, i nodi vengono eseguiti e tramite gli indirizzi specificati di peer iniziano a comunicare tra loro.

Vengono stampate tutte le informazioni relative al nodo etcd. Tra le informazioni presenti si può notare anche il valore di **heartbeat timeout** di 100ms e quello **election timeout** di 1000ms. I valori di heartbeat e election possono essere impostati quando si crea il nodo, attraverso i flag –**heartbeat-interval** e –**election-interval**.

Infine, viene fatta la Leader Election tra i nodi e replicate le informazioni inserite dai client attraverso Log Replication, che verrà mostrata in atto a breve.

Il primo cluster etcd è stato creato ed eseguito con successo!

5.2 etcdctl

E' possibile interagire con il cluster etcd attraverso **etcdctl**, che rappresenta il punto di ingresso delle API. etcdctl si connette al cluster etcd sfruttando gli indirizzi client definiti dal cluster al momento della sua "costruzione".

etcdctl può utilizzare due diverse versioni per interagire con il cluster: la versione 2 (di default) e la 3. E' consigliato utilizzare la versione 3 in quanto più recente e ricca di funzionalità.

Per utilizzare la versione 3 è sufficiente definire la variabile d'ambiente **ETCDCTL_API=3** come mostrato qui:

```
#Definire la variabile prima di utilizzare l'api
ETCDCTL_API=3 etcdctl

#Oppure utilizzare export per utilizzare la variabile nella bash corrente
export ETCDCTL_API=3
```

Nelle successive sezioni verrà mostrato come è possibile usufruire delle funzionalità messe a disposizione da etcd attraverso etcdctl.

5.3 Monitorare lo stato del cluster

Una volta che il cluster è in esecuzione, è possibile interagirci utilizzando **etcdctl**.

Per monitorare lo stato dei nodi del cluster, è possibile utilizzare il comando: **endpoint**. Il comando si divide in altri sottocomandi, tra i più utili: **endpoint status** ed **endpoint health**.

Utilizzando endpoint status è possibile ottenere lo stato dei nodi del cluster, in particolare:

- **Endpoint**: indica il nodo, in questo caso: localhost:2379 per s1, localhost:22379 per s2 e localhost:32379 per s3.
- **ID**: id univoco del nodo, utile con altri comandi di etcdtl (ad esempio rimuovere un nodo dal cluster).
- **Versione**: versione di etcd sul nodo.
- **Leader**: se il nodo è leader o meno (true o false), solitamente è leader il primo nodo avviato dato che il suo election timeout terminerà prima degli altri (se tutti e tre hanno stesso election timeout, di default a 1000ms).
- **Raft Term**: valore del term nel quale è stata fatta l'elezione del leader.
- **Raft Index**: valore che corrisponde alla posizione alla log entry in una serie di log entry.

Prendendo come esempio il cluster configurato e avviato sopra di tre nodi, si può eseguire il comando come:

```
    #ETCDCTL_API=3: indica la versione di etcdctl da utilizzare,
    # in alternativa come visto sopra è possibile utilizzare "export ETCDCTL_API=3"
    #--endpoints: nodi di cui monitorare lo stato
    #--cacert: certificato root ca
    #--cert: certficato di uno dei 3 nodi
    #--key: chiave privata di uno dei 3 nodi

    ETCDCTL_API=3 etcdctl \
    --endpoints localhost:2379,localhost:22379 \
    ,localhost:32379 \
    --cacert ${HOME}/certs/etcd-root-ca.pem \
    --cert ${HOME}/certs/s1.pem \
    --key ${HOME}/certs/s1-key.pem \
  endpoint status
```

I certificati sono necessari in quanto il cluster etcd richiede l'autenticazione da parte del client.

L'output dovrebbe essere simile a questo:

```
    localhost:22379, a39b736457d9b634, 3.3.8, 20 kB, true, 2, 8
    localhost:32379, 8824476cc9eecda8, 3.3.8, 20 kB, false, 2, 8
    localhost:2379, 1a7febac07525fe7, 3.3.8, 20 kB, false, 2, 8
```

Come alternativa per visualizzare lo stato dei nodi, si può utilizzare **endpoint health**, che permette di verificare la "salute" di ogni nodo del cluster.

endpoint health può essere utilizzato nel seguente modo:

```
    #Come prima tranne per l'ultima riga

    ETCDCTL_API=3 etcdctl \
    --endpoints localhost:2379,localhost:22379 \
    ,localhost:32379 \
    --cacert ${HOME}/certs/etcd-root-ca.pem \
    --cert ${HOME}/certs/s1.pem \
    --key ${HOME}/certs/s1-key.pem \
    endpoint health
```

L'output del comando endpoint health mostra se i nodi specificati sono attivi:

```
  localhost:22379 is healthy: successfully committed proposal: took = 5.414332ms
    localhost:2379 is healthy: successfully committed proposal: took = 2.631114ms
    localhost:32379 is healthy: successfully committed proposal: took = 4.531087ms
```

Ora si può provare a togliere uno dei nodi dal cluster. E' sufficiente entrare in una delle bash e fare **ctrl + c** per eliminare il nodo dal cluster. L'output di **endpoint status** risulterà ora:

```
Failed to get the status of endpoint localhost:32379 (context deadline exceeded)
localhost:22379, a39b736457d9b634, 3.3.8, 20 kB, true, 2, 8
localhost:2379, 1a7febac07525fe7, 3.3.8, 20 kB, false, 2, 8
```

Dall'output sopra si può osservare come etcdctl non riesce più a contattare s3.

In seguito verrà mostrato un modo migliore per rimuovere un nodo, utilizzando la gestione dei membri in etcdctl.

5.4 Gestire le entry

Essendo etcd un archivio chiave-valore distribuito è possibile gestire le entry contenute nel cluster attaverso etcdctl.

etcdctl permette infatti di: aggiungere, aggiornare, leggere, eliminare e osservare cambiamenti sulle entry del cluster etcd.

Con il comando **etcdctl put** è possibile aggiungere delle entry all'interno del archivio distribuito. Il comando put non ha sottocomandi specifici, semplicemente aggiunge una entry di chiave-valore come nell'esempio:

```
#Come prima tranne per l'ultima riga

ETCDCTL_API=3 etcdctl \
--endpoints localhost:2379,localhost:22379 \
,localhost:32379 \
--cacert ${HOME}/certs/etcd-root-ca.pem \
--cert ${HOME}/certs/s1.pem \
--key ${HOME}/certs/s1-key.pem \
put key value
```

Ora all'interno del cluster etcd viene aggiunta la entry con chiave **key** e valore **value**, replicata su tutti i nodi grazie alla log replication.

Nel caso in cui la chiave sia già presente in un'altra entry, il valore di quella entry viene aggiornato.

N.B. Durante la leader election può verificarsi il mancato inserimento di alcune entry.

La entry viene inserita correttamente nel sistema se in output viene mostrato **OK**.

Per ottenere un valore o insieme di valori da una o più chiavi si utilizza **get**. E' possibile ottenere dalla chiave di una entry la sola chiave (per verificare che esiste) o il singolo valore.

Per ottenere semplicemente il solo valore a partire dalla chiave è possibile utilizzare: **get key –print-value-only**. Mentre per ottenere solo la chiave della entry utilizzare: **–keys-only**.

Un esempio dell'utilizzo di **get** può essere il seguente:

```
#Come prima tranne per l'ultima riga

ETCDCTL_API=3 etcdctl \
```

```
--endpoints localhost:2379,localhost:22379 \
,localhost:32379 \
--cacert ${HOME}/certs/etcd-root-ca.pem \
--cert ${HOME}/certs/s1.pem \
--key ${HOME}/certs/s1-key.pem \
get key
```

E' possibile specificare come valore del flag endpoints solo uno dei tre nodi del cluster. Utilizzando il comando **get**, si noterà che la entry è stata replicata in tutti i nodi.

Per eliminare una entry si utilizza **etcdctl del** che elimina semplicemente la entry dal sistema. Se la entry è stata eliminata con successo, viene mostrato in output 1, altrimenti 0:

```
#Come prima tranne per l'ultima riga

ETCDCTL_API=3 etcdctl \
--endpoints localhost:2379,localhost:22379 \
,localhost:32379 \
--cacert ${HOME}/certs/etcd-root-ca.pem \
--cert ${HOME}/certs/s1.pem \
--key ${HOME}/certs/s1-key.pem \
del key
```

Per visualizzare le modifiche effettuate su una entry, etcdctl ne fornisce la funzionalità attraverso **watch**. Il comando rimane in attesa di modifiche e mostra tutte le modifiche effettuate (da un'altra bash ad esempio) su quella entry, quali: aggiunta, aggiornamento e cancellazione.

Il comando non comprende particolari sottocomandi e va utilizzato in questo modo:

```
#Come prima tranne per l'ultima riga

ETCDCTL_API=3 etcdctl \
--endpoints localhost:2379,localhost:22379 \
,localhost:32379 \
--cacert ${HOME}/certs/etcd-root-ca.pem \
--cert ${HOME}/certs/s1.pem \
--key ${HOME}/certs/s1-key.pem \
watch key
```

Utilizzando, ad esempio, il comando watch sulla entry di chiave **key** inserita prima, si ottengono tutte le modifiche su quella chiave:

```
PUT
key
value
```

```
PUT
key
value1
PUT
key
value2
PUT
key
value3
DELETE
key
```

5.5 Gestire i nodi del cluster

Attraverso etcdctl è possibile gestire i nodi di un cluster. E' possibile aggiungere, cancellare, aggiornare un nodo o visualizzare i nodi presenti nel cluster attraverso **etcdctl member add/remove/update/list**.

In etcdctl i nodi vengono chiamati **membri**.

Per visualizzare tutti i nodi del cluster si utilizza **etcdctl member list**. Per ogni nodo presente viene visualizzato:

- **ID**: id univoco del nodo.
- **Stato**: started o unstarted.
- **Nome**: nome del nodo, s1 ad esempio.
- **Indirizzi peer**: indirizzi di ascolto per le comunicazioni tra peer di ogni nodo.
- **Indirizzi client**: indirizzi di ascolto per le comunicazioni tra client e peer.

Member list può essere utilizzato specificando anche un solo endpoint del cluster:

```
#Come prima tranne per l'ultima riga

ETCDCTL_API=3 etcdctl \
--endpoints localhost:2379
--cacert ${HOME}/certs/etcd-root-ca.pem \
--cert ${HOME}/certs/s1.pem \
--key ${HOME}/certs/s1-key.pem \
member list
```

L'output mostrato è il seguente:

```
1a7febac07525fe7, started, s1, https://localhost:2380, https://localhost:2379
8824476cc9eecda8, started, s3, https://localhost:32380, https://localhost:32379
a39b736457d9b634, started, s2, https://localhost:22380, https://localhost:22379
```

Per rimuovere un nodo dal cluster si può utilizzare **etcdctl member remove**, specificando semplicemente l'id del nodo da rimuovere.

Ad esempio, si può rimuovere il nodo leader (s1) dal cluster, indicando il suo id nel comando **remove**:

```
#Come prima tranne per l'ultima riga

ETCDCTL_API=3 etcdctl \
--endpoints localhost:2379,localhost:22379 \
,localhost:32379 \
--cacert ${HOME}/certs/etcd-root-ca.pem \
--cert ${HOME}/certs/s1.pem \
--key ${HOME}/certs/s1-key.pem \
member remove 1a7febac07525fe7
```

Il nodo è stato effettivamente eliminato. Utilizzando member list è possibile vedere come il nodo s1 non sia più presente nel cluster:

```
8824476cc9eecda8, started, s3, https://localhost:32380, https://localhost:32379
a39b736457d9b634, started, s2, https://localhost:22380, https://localhost:22379
```

Eliminando il nodo leader del custer s1 viene fatta una nuova elezione, utilizzando come prima il comando **endpoint status** è possibile notare il nuovo leader del sistema.

E se si volesse riaggiungere nuovamente s1 al cluster? etcdctl permette di aggiungere un nuovo nodo al cluster attraverso il comando **etcdctl member add** specificando nome del nodo e indirizzi peer:

```
#Come prima tranne per l'ultima riga

ETCDCTL_API=3 etcdctl \
--endpoints localhost:2379,localhost:22379 \
,localhost:32379 \
--cacert ${HOME}/certs/etcd-root-ca.pem \
--cert ${HOME}/certs/s1.pem \
--key ${HOME}/certs/s1-key.pem \
member add s1 --peer-urls=https://localhost:2380
```

Il cluster viene avvisato che a breve verrà aggiunto un nuovo nodo e l'output è il seguente:

```
Member aa8ce52dfd5cb5ae added to cluster 7a06a9b4bcab2826

ETCD_NAME="s1"
ETCD_INITIAL_CLUSTER="s3=https://localhost:32380,s2=https://localhost:22380,s1=https://lo
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://localhost:2380"
ETCD_INITIAL_CLUSTER_STATE="existing"
```

Attenzione! Gli altri nodi del cluster sono stati avvisati della presenza di un nuovo nodo, nonostante non sia ancora stato aggiunto fisicamente al cluster (verificare con member list). Per aggiungere il nodo, è necessario (come consigliato dall'output sopra) aggiungerlo con il comando etcd e prestare attenzione alla configurazione del cluster con stato iniziale **existing** e non più **new**:

```
mkdir -p ${HOME}/certs
#cp /tmp/certs/* £{HOME}/certs

#Rimuove la cartella del vecchio nodo s1
rm -rf /tmp/etcd/s1

etcd --name s1 \
  --data-dir /tmp/etcd/s1 \
  --listen-client-urls https://localhost:2379 \
  --advertise-client-urls https://localhost:2379 \
  --listen-peer-urls https://localhost:2380 \
  --initial-advertise-peer-urls https://localhost:2380 \
  --initial-cluster s1=https://localhost:2380, \
  s2=https://localhost:22380, s3=https://localhost:32380 \
  --initial-cluster-token tkn \
  --initial-cluster-state existing \
  --client-cert-auth \
  --trusted-ca-file ${HOME}/certs/etcd-root-ca.pem \
  --cert-file ${HOME}/certs/s1.pem \
  --key-file ${HOME}/certs/s1-key.pem \
  --peer-client-cert-auth \
  --peer-trusted-ca-file ${HOME}/certs/etcd-root-ca.pem \
  --peer-cert-file ${HOME}/certs/s1.pem \
  --peer-key-file ${HOME}/certs/s1-key.pem
```

Eseguito lo script sopra, il nodo s1 viene aggiunto nuovamente al cluster e contattato dagli altri nodi tramite gli indirizzi peer.

Infine è possibile trasferire lo stato di leader del sistema attraverso **move-leader idNuovoLeader**.

Ad esempio se si volesse trasferire lo stato di leader del sistema ad s1:

```
#Come prima tranne per l'ultima riga

ETCDCTL_API=3 etcdctl \
--endpoints localhost:2379,localhost:22379 \
,localhost:32379 \
--cacert ${HOME}/certs/etcd-root-ca.pem \
--cert ${HOME}/certs/s1.pem \
```

```
--key ${HOME}/certs/s1-key.pem \
move-leader aa8ce52dfd5cb5ae
```

L'output è il seguente:

```
Leadership transferred from 8824476cc9eecda8 to aa8ce52dfd5cb5ae
```

5.6 Snapshot

etcd è progettato per resistere ai guasti dei nodi. Un cluster etcd si ripristina automaticamente da errori temporanei (ad esempio, riavvii dei nodi) e tollera fino a (N-1) / 2 errori permanenti per un cluster di N nodi. Se il cluster perde permanentemente più di (N-1) / 2 membri, fallisce disastrosamente, perdendo irrevocabilmente il quorum. Una volta perso il quorum, il cluster non può raggiungere il consenso e non può, quindi, continuare ad accettare aggiornamenti.

Il ripristino di un cluster richiede uno **snapshot** dello spazio delle chiavi da un nodo etcd. Lo snapshot permette di salvare l'attuale stato del cluster. Lo snapshot può essere preso da un nodo attivo con il comando di salvataggio snapshot etcdctl o copiando il file **snap / db** da una directory etcd.

etcdctl mette a disposizione delle funzionalità per gestire gli snapshot: salvataggio e ripristino di uno snapshot.

Per salvare uno snapshot si utilizza **etcdctl snapshot save**. Ad esempio, per eseguire lo snapshot e salvarlo nel Desktop dell'utente:

```
#Come prima tranne per l'ultima riga

ETCDCTL_API=3 etcdctl \
--endpoints localhost:2379,localhost:22379 \
,localhost:32379 \
--cacert ${HOME}/certs/etcd-root-ca.pem \
--cert ${HOME}/certs/s1.pem \
--key ${HOME}/certs/s1-key.pem \
snapshot save $HOME/Desktop/snapshot.db
```

Per ripristinare un cluster occorre un file "db" di snapshot. Per avviare un cluster da snapshot, quest'ultimo si serve di un nuovo cluster logico.

Attraverso **snapshot restore** è possibile ripristinare un nuovo cluster dal file .db creato precedentemente.

Il comando va eseguito per ogni nodo, specificando: percorso file .db, nome del nodo, stato iniziale del cluster, token del cluster, indirizzi peer, certificati e chiavi per l'autenticazione.

E' stato predisposto uno script per ripristinare il nodo s1 del cluster. Per ogni nodo del cluster da ripristinare va eseguito lo script, cambiando il nome del nodo e i certificati:

```
ETCDCTL_API=3 etcdctl snapshot restore $HOME/Desktop/snapshot.db \
--name s1 \
--initial-cluster s1=https://localhost:2380,
s2=https://localhost:22380,s3=https://localhost:32380  \
--initial-cluster-token tkn-1 \
--initial-advertise-peer-urls https://localhost:2380 \
--cacert ${HOME}/certs/etcd-root-ca.pem \
--cert ${HOME}/certs/s1.pem \
--key ${HOME}/certs/s1-key.pem \
```

Il comando restore eseguito per ogni nodo (s1,...,Sn) del cluster permette di creare i file **sN.etcd** contenenti i vecchi dati di ogni nodo.

Ora è possibile avviare il cluster come visto precedentemente attraverso etcd, basta specificare in **data-dir** il percorso dei file **.etcd**:

```
etcd --name s1 \
--data-dir $HOME/Desktop/etcd/s1.etcd \
--listen-client-urls https://localhost:2379 \
--advertise-client-urls https://localhost:2379 \
--listen-peer-urls https://localhost:2380 \
--initial-cluster s1=https://localhost:2380,
s2=https://localhost:22380,
s3=https://localhost:32380 \
--client-cert-auth \
--trusted-ca-file ${HOME}/certs/etcd-root-ca.pem \
--cert-file ${HOME}/certs/s1.pem \
--key-file ${HOME}/certs/s1-key.pem \
--peer-client-cert-auth \
--peer-trusted-ca-file ${HOME}/certs/etcd-root-ca.pem \
--peer-cert-file ${HOME}/certs/s1.pem \
--peer-key-file ${HOME}/certs/s1-key.pem \
```

Lo script sopra va eseguito per ogni nodo del cluster, cambiando nome, porte e certificati di autenticazione. Eseguito per tutti i nodi, il cluster tornerà allo stato precedente.

5.7 Misurare le prestazioni di etcd

Utilizzando etcdctl è possibile misurare le prestazioni del cluster etcd.

etcdctl permette di misurare le prestazioni del cluster attraverso **etcdctl check perf**, misurando: il numero di scritture al secondo (prestazioni migliori con dischi allo stato solido), la richiesta più lenta rilevata e la latenza della deviazione standard per ogni peer nel cluster.

Un esempio di utilizzo è il seguente:

```
#Come prima tranne per l'ultima riga

ETCDCTL_API=3 etcdctl \
--endpoints localhost:2379,localhost:22379 \
,localhost:32379 \
--cacert ${HOME}/certs/etcd-root-ca.pem \
--cert ${HOME}/certs/s1.pem \
--key ${HOME}/certs/s1-key.pem \
check perf
```

Un output possibile potrebbe essere il seguente:

```
    59 / 60 Boooooooooooooooooooooooooooooooooooooooooooooooooooo
    ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
    ooooooooooooooooooooooooooooooom  !  98.33% 1sPASS: Throughput is 144 writes/s
PASS: Slowest request took 0.055326s
PASS: Stddev is 0.002898s
```

5.8 Stress test

In questa sezione verrà eseguito qualche stress test sul cluster per cercare di portare etcd al di fuori della propria zona di comfort.

Per lo stress test è stato pensato di creare uno script .sh per popolare etcd di un range di valori chiavi-valore (ad esempio da 100 a 200) per poi cercare di cancellarli tutti. Infine viene mostrato quante entry sono state inserite e cancellate correttamente, lo script è il seguente:

```
#!/bin/bash
#Stress test etcd

#Verifica che ci sia il range di valori
if [ "$#" != "2" ]; then
        echo "Inserire un range di chiavi (esempio 100 200)!"
        exit 1
fi

entry_inserted=0
entry_deleted=0
total_entry=$(($2-$1+1))

#Popoliamo l'archivio etcd con il numero di entry specificato
#Verranno inserite coppie chiave valore come 1 1 e cosi' via....
for (( i=$1; i<=$2; i++ ))
do
   output_put=`ETCDCTL_API=3 etcdctl \
```

```
    --endpoints localhost:22379,localhost:32379,localhost:2379 \
    --cacert ${HOME}/certs/etcd-root-ca.pem \
    --cert ${HOME}/certs/s1.pem \
    --key ${HOME}/certs/s1-key.pem \
    put $i $i`

    if [ "$output_put" = "OK" ]; then
         entry_inserted=$((entry_inserted+1))
    fi
done

echo "Sono state inserite $entry_inserted/$total_entry entry!"

#Cerchiamo di cancellare tutte le entry!
for (( i=$1; i<=$2; i++ ))
do
    output_del=`ETCDCTL_API=3 etcdctl \
    --endpoints localhost:22379,localhost:32379,localhost:2379 \
    --cacert ${HOME}/certs/etcd-root-ca.pem \
    --cert ${HOME}/certs/s1.pem \
    --key ${HOME}/certs/s1-key.pem \
    del $i`

    if [ "$output_del" = "1" ]; then
         entry_deleted=$((entry_deleted+1))
    fi
done

echo "Sono state cancellate $entry_deleted/$total_entry entry!"
echo "Tempo impiegato: $SECONDS secondi"

exit 0
```

Lo script accetta due parametri che corrispondono a un range di chiave-valore da inserire. **./stressTestEtcd 1 1000**, ad esempio, inserirà tutte le coppie chiave valore da 1-1 a 1000-1000, per poi cercare di cancellarle tutte.

Provando ad eseguire lo script come sopra con 1 e 1000 l'output è il seguente:

```
Sono state inserite 1000/1000 entry!
Sono state cancellate 1000/1000 entry!
Tempo impiegato: 101 secondi
```

Non è particolarmente veloce, poichè è eseguito all'interno di due macchine virtuali, una relativa a Linux Lubuntu e l'altra quella di etcd bare metal, inoltre l'utilizzo dell'autenticazione tra client-server e peer-to-peer richiede più tempo.

In questo caso, le operazioni sono state eseguite in sequenza da un singolo client e risultano, quindi, facili da gestire per il cluster.

Proviamo a lanciare più operazioni in parallelo. L'output risultante lanciando tre comandi in background e parallelo è il seguente:

```
root@master-node:~/Desktop/etcd# (./stress_test_etcd_bare_metal.sh 0 500 &)
&& (./stress_test_etcd_bare_metal.sh 1000 1500 &)
&& (./stress_test_etcd_bare_metal.sh 2000 2500 &)
root@master-node:~/Desktop/etcd# Sono state inserite 501/501 entry!
Sono state inserite 501/501 entry!
Sono state inserite 501/501 entry!
Sono state cancellate 501/501 entry!
Tempo impiegato: 68 secondi
Sono state cancellate 501/501 entry!
Tempo impiegato: 69 secondi
Sono state cancellate 501/501 entry!
Tempo impiegato: 69 secondi
```

Anche eseguendo in parallelo, etcd si comporta come atteso. Ora si proverà a complicare ulteriormente le cose.

Si eseguono, come sopra, gli stessi comandi in parallelo, cercando di interrompere il nodo leader (con **ctrl + c** nella bash) durante l'esecuzione delle operazioni:

```
root@master-node:~/Desktop/etcd# (./stress_test_etcd_bare_metal.sh 0 500 &)
&& (./stress_test_etcd_bare_metal.sh 1000 1500 &)
&& (./stress_test_etcd_bare_metal.sh 2000 2500 &)
root@master-node:~/Desktop/etcd# Error: context deadline exceeded
Error: context deadline exceeded
Error: context deadline exceeded
Error: dial tcp [::1]:2379: getsockopt: connection refused
Error: dial tcp [::1]:2379: getsockopt: connection refused
Error: dial tcp [::1]:2379: getsockopt: connection refused
Sono state inserite 499/501 entry!
Sono state inserite 499/501 entry!
Sono state inserite 499/501 entry!
Sono state cancellate 499/501 entry!
Tempo impiegato: 58 secondi
Sono state cancellate 499/501 entry!
Tempo impiegato: 58 secondi
Sono state cancellate 499/501 entry!
Tempo impiegato: 58 secondi
```

Anche in questo caso etcd si è comportato bene, infatti solo qualche entry non è stata inserita correttamente (sicuramente quando è stato fermato il nodo leader).

Come ultimo test si proverà a inserire sempre in parallelo delle entry durante la Leader Election, per fare ciò i tre nodi s1, s2 ed s3 sono stati eseguiti aggiungendo il flag –**election-timeout 10000** impostando l'election timeout a 10 secondi.

Anche in questo caso etcd si è comportato bene, infatti tutte le entry sono state inserite e eliminate correttamente come mostrato nell'output:

```
root@master-node:~/Desktop/etcd# (./stress_test_etcd_bare_metal.sh 0 500 &)
&& (./stress_test_etcd_bare_metal.sh 1000 1500 &)
&& (./stress_test_etcd_bare_metal.sh 2000 2500 &)
root@master-node:~/Desktop/etcd# Sono state inserite 501/501 entry!
Sono state inserite 501/501 entry!
Sono state inserite 501/501 entry!
Sono state cancellate 501/501 entry!
Tempo impiegato: 68 secondi
Sono state cancellate 501/501 entry!
Tempo impiegato: 68 secondi
Sono state cancellate 501/501 entry!
Tempo impiegato: 68 secondi
```

5.9 etcd con container Docker

In questa ultima sezione di test verrà utilizzato etcd integrato con Docker per migliorare la portabilità di sviluppo iniziale.

Come in precedenza, è consigliato l'utilizzo di script bash .sh per non perdersi tra i vari comandi.

Innanzitutto utilizzando il comando principale **docker**, è necessario creare il volume docker da utilizzare per memorizzare i dati dell'archivio per ogni nodo del cluster in un container Docker. Per fare ciò digitare in un terminale **docker volume create etcd-data**.

Creato il volume docker, è possibile proseguire con lo script.

Lo script permette di creare tre container Docker in cui all'interno viene eseguito etcd (per semplicità non viene utilizzata l'autenticazione client-server e peer-to-peer):

```
#DATA_DIR = volume da utilizzare all'interno dei container
#IP = indirizzo ip della macchina
export DATA_DIR="etcd-data"
export IP=192.168.1.15

#Container etcd1 che corrisponde a s1
docker run -d \
  -p 2379:2379 \
  -p 2380:2380 \
  -v:${DATA_DIR}:/etcd-data \
  --name etcd1 quay.io/coreos/etcd:v3.4.14 \
```

```
 /usr/local/bin/etcd \
 -data-dir=/etcd-data -name s1 \
 -initial-advertise-peer-urls http://${IP}:2380 -listen-peer-urls
 http://0.0.0.0:2380 \
 -advertise-client-urls http://${IP}:2379 -listen-client-urls
 http://0.0.0.0:2379 \
 -initial-cluster s1=http://${IP}:2380,
 s2=http://${IP}:22380,s3=http://${IP}:32380 \
 -initial-cluster-token tkn \
 -initial-cluster-state new

#Container etcd2 che corrisponde a s2
docker run -d \
 -p 22379:22379 \
 -p 22380:22380 \
 -v:${DATA_DIR}:/etcd-data \
 --name etcd2 quay.io/coreos/etcd:v3.4.14 \
 /usr/local/bin/etcd \
 -data-dir=/etcd-data -name s2 \
 -initial-advertise-peer-urls
 http://${IP}:22380 -listen-peer-urls
 http://0.0.0.0:22380 \
 -advertise-client-urls http://${IP}:22379 -listen-client-urls
 http://0.0.0.0:22379 \
 -initial-cluster s1=http://${IP}:2380,
 s2=http://${IP}:22380,s3=http://${IP}:32380 \
 -initial-cluster-token tkn \
 -initial-cluster-state new

#Container etcd3 che corrisponde a s3
docker run -d \
 -p 32379:32379 \
 -p 32380:32380 \
 -v:${DATA_DIR}:/etcd-data \
 --name etcd3 quay.io/coreos/etcd:v3.4.14 \
 /usr/local/bin/etcd \
 -data-dir=/etcd-data -name s3 \
 -initial-advertise-peer-urls http://${IP}:32380 -listen-peer-urls
 http://0.0.0.0:32380 \
 -advertise-client-urls http://${IP}:32379 -listen-client-urls
 http://0.0.0.0:32379 \
 -initial-cluster s1=http://${IP}:2380,
 s2=http://${IP}:22380,s3=http://${IP}:32380 \
 -initial-cluster-token tkn \
```

```
-initial-cluster-state new
```

Come si può ben notare, parte dei comandi è utilizzata da etcd bare metal. Utilizzando il comando **docker run -d**, infatti, si esegue un container (in background) in cui eseguire i comandi etcd.

I flag docker utilizzati sono i seguenti:

- **-p**: si associano le porte dei container a quelle dell'host, questo permette all'utente di comunicare con i container e i container di comunicare tra di loro.
- **-v**: volume che viene utilizzato dal container in questo caso il volume è quello definito in cima al file.
- **–name**: assegna il nome al container docker, deve essere univoco, di seguito viene presa l'immagine da cui far partire il container, se non è presente in locale viene scaricata da remoto.

Quando si esegue lo script, vengono eseguiti tre container docker contenenti etcd e comunicanti tra di loro.

Docker ritorna in output l'id dei tre container appena creati:

```
root@master-node:~/Desktop/etcd# ./docker_etcd.sh
22308f935bc50a3de051503427c8d4346a1c42db7aec718311f8ea120c9dbeed
f972de409031fa58667bb70eead0019be7549173f0e9ccc34a7a27353d357ab9
edf90e8d6f91c369e91bfd90aed26dec7649bbde16eb50ac3cc6f3631a38e8af
```

E' possibile visualizzare i container in esecuzione attraverso **docker ps**:

```
root@master-node:~/Desktop/etcd# docker ps

CONTAINER ID   IMAGE                          COMMAND                  CREATED

edf90e8d6f91   quay.io/coreos/etcd:v3.4.14   "/usr/local/bin/etcd..."   45 seconds ago
f972de409031   quay.io/coreos/etcd:v3.4.14   "/usr/local/bin/etcd..."   46 seconds ago
22308f935bc5   quay.io/coreos/etcd:v3.4.14   "/usr/local/bin/etcd..."   46 seconds ago

STATUS              PORTS

Up 43 seconds       2379-2380/tcp, 0.0.0.0:32379-32380->32379-32380/tcp
Up 44 seconds       2379-2380/tcp, 0.0.0.0:22379-22380->22379-22380/tcp
Up 45 seconds       0.0.0.0:2379-2380->2379-2380/tcp

NAMES

etcd3
etcd2
etcd1
```

Per ogni container vengono visualizzate le seguenti informazioni:

- **ID**: id del container in esecuzione, utile quando si vuole ad esempio fermare il container attraverso **docker stop id** o quando lo si vuole cancellare attraverso **docker rm -f id**.
- **Image**: immagine da cui è costruito il container.
- **Command**: comando che viene eseguito all'interno del container (etcd in questo caso).
- **Created**: quando è stato creato il container.
- **Status**: stato del container (se sta eseguendo, se fermato, se uscito, ...).
- **Ports**: porte che vengono mappate da host a container.
- **Names**: nome del container.

Per interagire con i container attraverso etcdctl come già fatto in precedenza è sufficiente utilizzare il comando come visto con bare-metal, specificando l'indirizzo ip e porte dei container in cui risiedono i nodi etcd:

```
ETCDCTL_API=3 etcdctl --endpoints http://192.168.1.15:2379,
http://192.168.1.15:22379,http://192.168.1.15:32379 endpoint status
```

La forma dell'output ottenuto tramite etcdctl è identico a prima.
Utilizzando docker è possibile effettuare tutte le operazioni eseguite in precedenza, ma gestendo ognuno dei nodi più efficacemente grazie ai container Docker.
E' possibile replicare anche lo stress test in Docker cambiando solo gli indirizzi ip e senza l'utilizzo dei certificati:

```
#!/bin/bash
 #Stress test etcd con Docker

 if [ "$#" != "2" ]; then
         echo "Inserire un range di chiavi (esempio 100 200)!"
         exit 1
 fi

 entry_inserted=0
 entry_deleted=0
 total_entry=$(($2-$1+1))

 #Popoliamo l'archivio etcd con il numero di entry specificato
 for (( i=$1; i<=$2; i++ ))
 do
```

```
        output_put=`ETCDCTL_API=3 etcdctl \
        --endpoints http://192.168.1.15:2379,http://192.168.1.15:22379,http://192.168.1.15
        put $i $i`

        if [ "$output_put" = "OK" ]; then
             entry_inserted=$((entry_inserted+1))
        fi

    done

    echo "Sono state inserite $entry_inserted/$total_entry entry!"

    #Cerchiamo di cancellare tutte le entry!
    for (( i=$1; i<=$2; i++ ))
    do
        output_del=`ETCDCTL_API=3 etcdctl \
        --endpoints http://192.168.1.15:2379,http://192.168.1.15:22379,
        http://192.168.1.15:32379 \
        del $i`

        if [ "$output_del" = "1" ]; then
             entry_deleted=$((entry_deleted+1))
        fi
    done

    echo "Sono state cancellate $entry_deleted/$total_entry entry!"
    echo "Tempo impiegato: $SECONDS secondi"

    exit 0
```

Ad esempio, eseguendo (come fatto in precedenza) lo script in parallelo (stando attenti ad inserire l'indirizzo IP corretto), si ottiene:

```
root@master-node:~/Desktop/etcd# (./stress_test_docker_etcd.sh 0 500 &)
&& (./stress_test_docker_etcd.sh 1000 1500 &)
&& (./stress_test_docker_etcd.sh 2000 2500 &)
root@master-node:~/Desktop/etcd# Sono state inserite 501/501 entry!
Sono state inserite 501/501 entry!
Sono state inserite 501/501 entry!
Sono state cancellate 501/501 entry!
Tempo impiegato: 29 secondi
Sono state cancellate 501/501 entry!
Tempo impiegato: 29 secondi
Sono state cancellate 501/501 entry!
Tempo impiegato: 29 secondi
```

Lo stress test è molto più veloce senza l'utilizzo dell'autenticazione client-server e peer-to-peer.

Se si vuole interrompere un nodo durante l'esecuzione dello stress test, è sufficiente utilizzare un altro terminale ed eseguire il comando **docker stop nome-container/id-container**.

Quindi qual'è la differenza nell'utilizzare etcd bare metal o Docker? Docker è molto più portabile e consente più facilmente di creare un cluster di nodi etcd. Nonostante ciò, i due metodi sono sostanzialmente uguali.

6 etcd in Java

Come detto in precedenza, etcd, oltre alla linea di comando può essere utilizzato attraverso linguaggi di programmazione, in questo caso **jetcd** per **Java**.

La libreria Java per etcd è la seguente: `https://github.com/etcd-io/jetcd`.

La libreria richiede una versiona da Java 8 in su.

Per utilizzare jetcd è consigliato per semplicità, creare un cluster, di tre nodi bare metal (come sopra) senza l'utilizzo dell'autenticazione.

Per ogni nodo infatti è sufficiente eliminare i comandi relativi all'autenticazione, sotto il codice d'esempio del nodo s1:

```
mkdir -p ${HOME}/certs
#cp /tmp/certs/* £{HOME}/certs

#rm -rf /tmp/etcd/s1

#Uguale a sopra senza autenticazione
etcd --name s1 \
  --data-dir /tmp/etcd/s1 \
  --listen-client-urls http://192.168.1.15:2379 \
  --advertise-client-urls http://192.168.1.15:2379 \
  --listen-peer-urls http://192.168.1.15:2380 \
  --initial-advertise-peer-urls http://192.168.1.15:2380 \
  --initial-cluster s1=http://192.168.1.15:2380,s2=http://192.168.1.15:22380,
  s3=http://192.168.1.15:32380 \
  --initial-cluster-token tkn \
  --initial-cluster-state new
```

Prima di tutto, è necessario importare **jetcd** nel progetto attraverso le **dependencies** di Gradle, infatti Gradle verrà utilizzato come sistema di build automation. Per farlo è sufficiente aggiungere jetcd nelle dependencies di Gradle:

```
dependencies {
    //jetcd-core:0.5.4 è l'ultima versione di jetcd
    compile "io.etcd:jetcd-core:0.5.4"
}
```

Una volta che Gradle ha scaricato e aggiunto jetcd al progetto è possibile utilizzare la libreria.

La libreria jetcd permette di eseguire le stesse operazioni effettuabili da etcdctl a linea di comando.

6.1 Connettersi a etcd

Per connettersi al cluster etcd viene fornita un interfaccia **Client**, questa permette di costruire il Client attraverso un **ClientBuilder**.

Il ClientBuilder costruisce un Client a seconda di alcuni parametri come gli endpoints, infatti come in etcdctl gli endpoints sono i nodi del cluster etcd a cui connettersi. Il ClientBuilder fornisce altrettanti metodi per la gestione della connessione al cluster (connectionTimeout, keepAliveTimeout ecc...), non essenziali in questi esempi. Il client viene costruito utilizzando il metodo **build()** del ClientBuilder.

Quindi per connettersi al cluster etcd senza autenticazione è sufficiente definire il Client con uno o più endpoints:

```
public class Main {
  public static void main(String[] args) {

    //Come endpoint viene definito solo s1 in questo esempio
    final Client client = Client.builder().endpoints("http://192.168.1.15:2379").build()


    /*

    In alternativa è possibile definire tutti i nodi del cluster

    final Client client = Client.builder().endpoints("http://192.168.1.15:2379,
    http://192.168.1.15:22379,http://192.168.1.15:32379").build();

    */

    }
}
```

Il Client potrebbe lanciare una **ConnectException** se la connessione con il cluster non va a buon fine o una **AuthFailedException** per un'errata autenticazione.

Una volta creato, il client permette di gestire: autenticazione (non utilizzata ora), stato e manutenzione del cluster, leader election, spazio delle chiavi, funzione watch sulle entry ecc...

Alcune funzionalità mostrate con etcdctl verranno mostrate man mano nelle prossime sezioni con jetcd.

Molte delle funzionalità di interazione tra client e cluster etcd vengono eseguite in maniera asincrona attraverso **CompletableFuture**.

6.2 Gestione dei membri

Come etcdctl anche jetcd permette di aggiungere, rimuovere, aggiornare e visualizzare i nodi del cluster etcd.

Infatti, attraverso il metodo **getCluster()** del client è possibile effettuare le operazioni di gestione dei nodi:

```
public class Main {
  public static void main(String[] args) {
        final Client client = Client.builder().endpoints("http://192.168.1.15:2379")
        .build();
        final Cluster cluster = client.getClusterClient();
    }
}
```

Quando si utilizzano le funzionalità dell'interfaccia Cluster possono essere lanciate due eccezioni:

- **ExecutionException**: generata durante il tentativo di recuperare il risultato di un task che è stato interrotto generando un'eccezione.
- **InterruptedException**: generata quando un thread è in attesa, inattivo o altrimenti occupato e il thread viene interrotto, prima o durante l'attività.

Per visualizzare tutti i membri dall'interfaccia Cluster si può utilizzare **listMember()** (equivalente a **etcdctl member list**), che ritorna una CompletableFuture di **MemberListResponse** su cui fare un ciclo per ogni membro presente nel cluster:

```
public class Main {
    public static void main(String[] args) throws
    ExecutionException, InterruptedException {
        final Client client = Client.builder().endpoints("http://192.168.1.15:2379")
        .build();
        final Cluster cluster = client.getClusterClient();

        final CompletableFuture<MemberListResponse> listMember = cluster.listMember();
        final MemberListResponse memberListResponse = listMember.get();
        for (final Member member : memberListResponse.getMembers()) {
            System.out.println(member.getName());
            System.out.println(member.getId());
            System.out.println(member.getClientURIs().toString());
            System.out.println(member.getPeerURIs().toString());
        }
    }
}
```

L'output mostra per ogni membro il nome, l'id, gli indirizzi client e gli indirizzi peer, proprio come etcd member list:

```
s1
-2354960994399069488
[http://192.168.1.15:2379]
[http://192.168.1.15:2380]
s3
-2275287119153532060
[http://192.168.1.15:32379]
[http://192.168.1.15:32380]
s2
-1510123999760036531
[http://192.168.1.15:22379]
[http://192.168.1.15:22380]
```

A differenza di etcdctl dove gli id dei nodi sono alfanumerici qua vengono rappresentati come **long**. Quindi per eliminare un membro, ad esempio, va considerato il suo id come long e non come mostrato da linea di comando.

Se si volesse eliminare il nodo s2 dal cluster basterebbe iterare fino a trovarlo ed eliminarlo attraverso il suo id:

```
public class Main {
    public static void main(String[] args) throws
    ExecutionException, InterruptedException {
        final Client client = Client.builder().endpoints("http://192.168.1.15:2379")
        .build();
        final Cluster cluster = client.getClusterClient();

        final CompletableFuture<MemberListResponse> listMember = cluster.listMember();
        final MemberListResponse memberListResponse = listMember.get();
        for (final Member member : memberListResponse.getMembers()) {
            if (member.getName().equals("s2")) {
                cluster.removeMember(member.getId()).get();
                System.out.println("Nodo s2 eliminato con successo dal cluster!");
                break;
            }
        }
    }
}
```

Per aggiungere nuovamente s2 nel cluster è sufficiente utilizzare **addMember()** indicando come parametro la lista di indirizzi peer del nodo:

```
public class Main {
    public static void main(String[] args) throws
```

```
    ExecutionException, InterruptedException {
        final Client client = Client.builder().endpoints("http://192.168.1.15:2379")
        .build();
        final Cluster cluster = client.getClusterClient();
        final List<URI> uriList = new ArrayList<>();
        uriList.add(URI.create("http://192.168.1.15:22380"));
        final MemberAddResponse memberAddResponse = cluster.addMember(uriList).get();
    }
}
```

addMember() è l'equivalente di **etcdctl add member** indicando solamente gli indirizzi peer.

Una volta eseguito il codice, va riaggiunto il nodo fisicamente attraverso etcd (da linea di comando) come mostrato in precedenza.

6.3 Stato e manutenzione del cluster

In questa sezione vengono mostrate alcune funzionalità di manutenzione del cluster.

Il client attraverso l'interfaccia **Maintenance** ottenibile da **getMaintenanceClient()** permette di ottenere e gestire:

- gli avvisi relativi al cluster etcd.
- l'elezione di un nuovo leader del cluster.
- gli snapshots.
- lo stato dei nodi del cluster.

Per ottenere lo stato dei nodi del cluster dall'interfaccia Maintenance si utilizza il metodo **statusMember()**. Attraverso una CompletableFuture è possibile ottenere lo **StatusResponse** che contiene le informazioni equivalenti a **endpoint status** di etcdctl:

```
public class Main {
  public static void main(String[] args) {
        final Client client = Client.builder().endpoints("http://192.168.1.15:2379")
        .build();
        final Maintenance maintenance = client.getMaintenanceClient();

        CompletableFuture<StatusResponse> statusResponse = maintenance.statusMember
        (URI.create("http://192.168.1.15:2379"));
        System.out.println(statusResponse.get().toString());
    }
}
```

A differenza di etcdctl il leader viene indicato con il suo id:

```
version: "3.3.8"
dbSize: 20480
leader: 16091783079310482128
raftIndex: 16
raftTerm: 4
```

Per eleggere un nuovo leader del sistema, in etcdctl si utilizzava **move-leader** mentre in jetcd viene fornito il metodo **moveLeader()** passando al metodo l'id del nuovo leader. Attraverso l'interfaccia Cluster è possibile ottenere l'id di un nodo desiderato per eleggerlo leader:

```
public class Main {
  public static void main(String[] args)
  throws ExecutionException, InterruptedException {
        final Client client = Client.builder().endpoints("http://192.168.1.15:2379")
        .build();
        final Cluster cluster = client.getClusterClient();
        final Maintenance maintenance = client.getMaintenanceClient();

        final CompletableFuture<MemberListResponse> listMember = cluster.listMember();
        final MemberListResponse memberListResponse = listMember.get();

        //Scorre tutti i membri del cluster
        for (final Member member : memberListResponse.getMembers()) {
            //s2 scelto come nuovo leader
            if (member.getName().equals("s2")) {
                maintenance.moveLeader(member.getId()).get();
                /* Verifica se s2 è ora leader */
                final StatusResponse statusResponse = maintenance
                .statusMember(URI.create("http://192.168.1.15:2379")).get();
                if (statusResponse.getLeader() == member.getId()) {
                    System.out.println("s2 eletto nuovo leader del sistema");
                }
            }
        }
    }
}
```

6.4 Gestire lo spazio di chiavi

In jetcd come in etcdctl è possibile gestire le entry. In particolare è possibile: aggiungere, aggiornare, rimuovere e ottenere il valore delle entry.

Il Client permette di ottenere lo spazio di chiavi utilizzando il metodo **getKVClient()**. **KV** rappresenta l'interfaccia per la gestione delle entry:

```
public class Main {
  public static void main(String[] args) {
        final Client client = Client.builder().endpoints("http://192.168.1.15:2379")
        .build();
        final KV kv = client.getKVClient();
    }
}
```

Anche qui nelle operazioni di gestione delle entry possono essere lanciate le due eccezioni: ExecutionException e InterruptedException.

Per aggiungere una entry è necessario dichiarare due **ByteSequence** corrispondenti alla coppia chiave-valore. In seguito va utilizzato il metodo **put()** per aggiungere la entry:

```
public class Main {
    public static void main(String[] args)
    throws ExecutionException, InterruptedException {
        final Client client = Client.builder().endpoints("http://192.168.1.15:2379")
        .build();
        final KV kvClient = client.getKVClient();


        //Entry di chiave=key e valore=value
        final ByteSequence key = ByteSequence.from("key".getBytes());
        final ByteSequence value = ByteSequence.from("value".getBytes());

        kvClient.put(key, value).get();
    }
}
```

Il codice sopra è l'equivalente a **etcdctl put key value**.

Ora, per verificare che la entry sia stata inserita correttamente va utilizzato il metodo **get()** del KV, equivalente a **etcdctl get key**. Il metodo permette di accedere ad una struttura **KeyValue**, contenente chiave-valore:

```
public class Main {
    final Client client = Client.builder().endpoints("http://192.168.1.15:2379")
    .build();
    final KV kvClient = client.getKVClient();

    final ByteSequence key = ByteSequence.from("key".getBytes());

    for (final KeyValue keyValue : kvClient.get(key).get().getKvs()) {
        System.out.println("Chiave: " + new String(keyValue.getKey().getBytes()
```

```
        , StandardCharsets.UTF_8));
        System.out.println("Valore: " + new String(keyValue.getValue().getBytes()
        , StandardCharsets.UTF_8));
    }
}
```

Mentre per eliminare una entry, il KV mette a disposizione il metodo **delete()** che elimina la entry dal **ByteSequence** della chiave:

```
public class Main {
    final Client client = Client.builder().endpoints("http://192.168.1.15:2379")
    .build();
    final KV kvClient = client.getKVClient();

    final ByteSequence key = ByteSequence.from("key".getBytes());

    //getDeleted() ritorna il numero di entry eliminate
    if(kvClient.delete(key).get().getDeleted() > 0) {
        System.out.println("Entry eliminata con successo");
    }
}
```

Anche in questo caso il metodo è lo specchio del comando **etcdctl del key**.

6.5 Watch sulle entry

In jetcd c'è la possibilità di osservare le operazioni effettuata su una entry come in etcdctl. Infatti attraverso l'interfaccia client è possibile ottenere **getWatchClient()**. L'interfaccia **Watch** mette a disposizione una serie di metodi simili tra loro per visualizzare le operazioni effettuate su una entry. Per fare ciò viene utilizzato un **listener** che mostra per ogni evento: il tipo (put, get, del, ecc...), la chiave e il suo valore.

Per rimanere in attesi di eventi, viene utilizzato un oggetto **CountDownLatch** che permette di attendere il completamento di un numero di operazioni definite:

```
public class Main {

    // 10 operazioni massime sulla entry
    private static final int MAX_EVENTS = 10;

    public static void main(String[] args) {
        final Client client = Client.builder().endpoints("http://192.168.1.15:2379")
        .build();

        final CountDownLatch latch = new CountDownLatch(MAX_EVENTS);
        final ByteSequence key = ByteSequence.from("key".getBytes());
```

```
        final Watch.Listener listener = Watch.listener(response -> {
            System.out.println("Osservando la chiave: "
            + new String(key.getBytes(), StandardCharsets.UTF_8));

            for (final WatchEvent event : response.getEvents()) {
                System.out.println("Tipo: " + event.getEventType() + " , chiave: " +
                        Optional.ofNullable(event.getKeyValue().getKey())
                            .map(bs -> bs.toString(StandardCharsets.UTF_8)).orElse("")
                        + " , valore: " +
                        Optional.ofNullable(event.getKeyValue().getValue())
                            .map(bs -> bs.toString(StandardCharsets.UTF_8))
                                .orElse(""));
            }
            latch.countDown();
        });

        try (final Watch watch = client.getWatchClient();
             final Watch.Watcher watcher = watch.watch(key, listener)) {
            latch.await();
        } catch (Exception e) {
            System.exit(1);
        }
    }
}
```

Eseguendo il codice si può notare come l'esecuzione rimanga in attesa di 10 eventi, superati i quali l'esecuzione termina.

6.6 Stress test in Java

E' stato mostrato in parte come con Java è possibile replicare le stesse funzionalità di etcdctl. Ovviamente non sono state mostrate tutte le funzionalità disponibili ma le principali.

Come ultima cosa, verranno replicati gli stress test fatti con etcdctl.

Per fare ciò viene creata una classe di Test chiamata **StressTest** in cui vengono aggiunti man mano i test.

Inizialmente prima dei test vengono configurati client e spazio delle chiavi del cluster:

```
public class StressTest {
    private Client client;
    private KV kv;

    @Before
```

```
    public void setClient() {
        this.client = Client.builder().endpoints("http://192.168.1.15:2379")
        .build();
        this.kv = client.getKVClient();
    }
}
```

Come primo stress test verrà replicato il test di aggiunta e cancellazione delle entry, lo stesso già effettuato da etcdctl. Il test semplicemente inserisce e cancella 10000 entry. Per semplicità vengono create due costanti **MIN_RANGE_ENTRY _TEST** e **MAX_RANGE_ENTRY _TEST** per indicare il range di valori chiave-valore delle entry da inserire/eliminare:

```
public class StressTest {
    private static final int MIN_RANGE_ENTRY_TEST = 1;
    private static final int MAX_RANGE_ENTRY_TEST = 10000;

    /* Utilizzate in seguito */
    private int counter_thread;
    private boolean parallel_test = false;

    private Client client;
    private KV kv;

    @Before
    public void setClient() {
        this.client = Client.builder().endpoints("http://192.168.1.15:2379").build();
        this.kv = client.getKVClient();
    }

     @Test
    public void stressTestSequentially() throws ExecutionException, InterruptedException
        final Instant start = Instant.now();
        int entry_inserted = 0;
        int entry_deleted = 0;

        /* Se non viene eseguito in parallelo il range di chiavi rimane lo stesso */
        int min = parallel_test ? MIN_RANGE_ENTRY_TEST + MAX_RANGE_ENTRY_TEST
        * counter_thread++ : MIN_RANGE_ENTRY_TEST;
        int max = parallel_test ? MAX_RANGE_ENTRY_TEST * counter_thread++ :
        MAX_RANGE_ENTRY_TEST;
```

```
        for (int i = min; i < max; i++) {
            final ByteSequence key = ByteSequence.from(String.valueOf(i).getBytes());
            final ByteSequence value = ByteSequence.from(String.valueOf(i).getBytes());

            final PutResponse putResponse = kv.put(key, value).get();
            if (putResponse != null) {
                entry_inserted++;
            }
        }

        for (int i = min; i < max; i++) {
            final ByteSequence key = ByteSequence.from(String.valueOf(i).getBytes());

            final DeleteResponse deleteResponse = kv.delete(key).get();
            if (deleteResponse.getDeleted() > 0) {
                entry_deleted++;
            }
        }

        final Instant end = Instant.now();
        final Duration timeElapsed = Duration.between(start, end);

        Assert.assertEquals(entry_inserted, entry_deleted);
        Assert.assertEquals(entry_inserted + 1, max - min + 1);

        System.out.println("Tempo di esecuzione: " + timeElapsed.toSeconds()
        + " secondi");
    }
}
```

Il test controlla attraverso **assertEquals()** che tutte le 10000 entry siano state inserite e cancellate con successo. Inoltre viene misurato e stampato il tempo di esecuzione.

Come per il test di etcdctl anche questo test inserisce e cancella correttamente tutte le entry.

Mediamente, il test ci mette poco più di un minuto ad eseguire senza autenticazione, altrimenti ci metterebbe molto di più.

Ora, per esegure il test in parallelo è sufficiente utilizzare dei Thread. Viene utilizzato come per la funzione Watch un **CountDownLatch** per attendere che i Thread finiscano di eseguire il test definito sopra. Il numero di Thread è definito nella costante **N_THREADS**:

```
    private static final int N_THREADS = 5;

    private void runThread(final CountDownLatch latch) {
```

```
        final List<Thread> threadList = new ArrayList<>(N_THREADS);
        for (int i = 0; i < N_THREADS; i++) {
            Thread thread = new Thread(() -> {
                try {
                    stressTestSequentially();
                    latch.countDown();
                } catch (ExecutionException | InterruptedException e) {
                    e.printStackTrace();
                }
            });
            threadList.add(thread);
            threadList.get(i).start();
        }


    }

    @Test
    public void stressTestParallel() throws InterruptedException {

        final CountDownLatch latch = new CountDownLatch(N_THREADS);

        //Utilizzato per gestire diversi range di chiavi per ogni thread
        counter_thread = 0;

        parallel_test = true;
        runThread(latch);

        latch.await();

        parallel_test = false;
    }
```

Lanciando 5 thread, il test termina correttamente anche nel caso in cui si interrompa uno dei nodi durante l'esecuzione. I 5 thread provano tutti i range di chiavi da 1-10000 a 40001-50000.

Quindi in conclusione jetcd permette di definire facilmente delle classi di Test, di manutenzione, di aggiunta dati e tanto altro. A differenza di etcdctl con jetcd è più semplice definire dei test per stressare la tecnologia.

7 etcd vs Zookeeper

In questa sezione vengono evidenziate le principali differenze tra la tecnologia **Zookeeper** [1] e etcd.

7.1 Analisi finale di etcd

etcd si è rivelata una buona tecnologia come supporto per il proprio sistema distribuito di nodi. Con etcd è possibile costruire un servizio di backend distribuito e consistente su tutti i nodi del cluster.

Tutto questo è possibile tramite Raft e come visto precedentemente Raft è un algoritmo di consenso, che prima di memorizzare una entry si accerta che tutti gli altri nodi l'abbiano memorizzata, garantendo consistenza e coerenza dei dati su tutto il sistema.

Non abbiamo la presenza di un single point of failure in quanto tutti i nodi condividono le stesse informazioni e sono "indipendenti" dagli altri.

Infine, abbiamo notato che utilizzare etcd insieme a docker porta una maggiore portabilità e semplicità di utilizzo. L'utilizzo di etcd bare metal è più complesso e a primo impatto meno comprensibile dall'utente.

etcd possiede anche delle criticità. I dati scritti necessitano di dischi veloci come SSD per permettere ad uno scenario reale di funzionare come dovuto e ciò influisce sul numero di nodi massimo da utilizzare (non più di 7 nodi consigliati). Non esiste un limite, ma più nodi vengono aggiunti e più scritture sui diversi nodi vanno effettuate.

L'utilizzo di pochi nodi può portare ad una bassa tolleranza dei guasti, che è possibile compensare disponendo i nodi su domini separati aumentando la failure tolerance.

Inoltre, il client potrebbe non essere informato sullo stato di un'operazione, come, ad esempio, un'interruzione di rete tra il client e il cluster etcd. etcd può anche interrompere le operazioni quando si elegge un leader. etcd non invia risposte di interruzione alle richieste in sospeso dei client.

7.2 Zookeeper

Sicuramente la tecnologia che va più in contrasto con etcd è **Zookeeper**, nato come sottoprogetto di **Hadoop** e che si è evoluto fino a diventare un progetto di **Apache**. Grazie alla sua esperienza e stabilità, Zookeeper è diventato uno dei migliori sistemi di coordinamento distribuito al mondo.

A differenza di etcd Zookeper utilizza il protocollo **ZAB**.

Al fine di ottenere un'elevata disponibilità, Zookeeper lavora in un **zookeeper ensemble** (insieme di nodi). Come in Raft, è presente una fase di elezione del leader dove viene eletto un nodo leader per maggioranza di voti. I nodi si distinguono tra Leader e Follower. Le richieste di lettura possono essere gestite da ogni nodo, mentre quelle di scrittura solo dal nodo leader, come in etcd.

Delle tre principali proprietà (availability, consistency, partition tolerance), Zookeeper garantisce le ultime due.

7.3 ZNodes

Durante la memorizzazione dei dati, Zookeeper utilizza una struttura ad albero in cui ogni nodo è chiamato **ZNode** e denomina tali ZNode in base al percorso dal nodo radice. Ogni ZNode ha un nome. Quando si accede a un determinato ZNode, viene utilizzato il percorso assoluto dal nodo radice.

L'utente può creare ZNodes, cancellare ZNodes, memorizzare dati nei ZNodes, leggere da un particolare ZNode. Ogni ZNode può contenere al massimo 1MB di dati.

7.4 Zookeeper Watches

Zookeeper fornisce una **API watcher**.

Gli utenti possono mettere un **watch** su un dato ZNode. Quando si verifica una qualsiasi modifica (creazione, eliminazione, modifica dei dati, aggiunta / rimozione di ZNode figlio) a tale ZNode, questa API watch notifica questa modifica.

L'unico inconveniente dei watch Zookeeper è che un determinato watch viene attivato solo una volta. Una volta che un watch viene notificato, per ricevere eventi futuri sullo stesso ZNode, gli utenti devono posizionare un nuovo watch su quello ZNode e questa può essere considerata come una limitazione in Zookeeper. In etcd non è necessario posizionare un nuovo watch.

7.5 Vantaggi e svantaggi di Zookeeper

I vantaggi di Zookeeper sono:

- Un'efficiente gestione della memoria attraverso gli ZNode.
- Affidabilità.
- Riconnessione automatica.
- Un'API semplificata.
- La notifica degli eventi tramite i watch Zookeeper.

Gli svantaggi sono:

- Poiché Zookeeper è scritto in Java, eredita le pause di garbage collection. etcd non presenta questo problema in quanto scritto in GO.
- Quando si creano gli snapshot, le operazioni di lettura e scrittura di Zookeeper vengono temporaneamente interrotte, mentre in etcd questo problema non è presente.
- Zookeeper apre una nuova connessione per ogni nuova richiesta di controllo. Ciò ha reso Zookeeper più complesso, poiché deve gestire molte connessioni socket aperte in tempo reale. In etcd non viene creata una connessione per richiesta, ma viene fatto un multiplexing su un range di chiavi.

Abbiamo discusso delle principali caratteristiche, dei principali vantaggi e svantaggi di Apache Zookeeper ed etcd.

Zookeeper scritto in Java è ampiamente adottato nei progetti di Apache, mentre etcd è supportato da Google (Kubernetes).

Anche se Apache Zookeeper è stabile e rinomato per essere un ottimo sistema di coordinamento distribuito, etcd è nuovo e promettente.

Quindi etcd o Zookeeper? Dipende dalle esigenze.

8 Conclusioni

In conclusione, posso ritenermi molto soddisfatto del lavoro svolto. Devo ammettere che, non avendo esperienza nel costruire un cluster di nodi o nel gestire dei container, non è stato facile inizialmente familiarizzare con etcd e Docker. Ho potuto notare, inoltre, una particolare chiarezza nella documentazione relativa a Docker, che si è rivelata più chiara ed esaustiva rispetto a quella relativa ad etcd.

Penso che queste competenze mi potranno tornare molto utili in futuro, in quanto l'utilizzo di queste tecnologie sarà sempre più fondamentale sopratutto nel Cloud.

9 Sviluppi futuri

Uno degli sviluppi futuri che mi viene subito in mente è l'utilizzo di etcd come servizio di memorizzazione dei dati in Kubernetes, la piattaforma più utilizzata per l'orchestrazione di container dove etcd trova una fortissima applicazione.

References

[1] Apache. Apache zookeeper.

[2] Golang. Go programming language.

[3] IBM Cloud Learn Hub. What is etcd?

[4] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[5] thesecretlivesofdata. Raft, understandable distributed consensus.