**Approfondimento su Apache Zookeeper e Spring Cloud Config**

Apache Zookeeper è stato creato per sviluppare e mantenere un server open source che abilita la coordinazione distribuita ad alta affidabilità.

È un servizio centralizzato per mantenere l’informazione di configurazione, naming e sincronizzazione distribuita. Tutti questi tipi di servizi sono usati in qualche modo dalle applicazioni distribuite.

L’implementazione manuale di questi servizi normalmente porterebbero a tanti bug e a tanti race conditions, usando Zookeeper invece si evita tutto ciò.

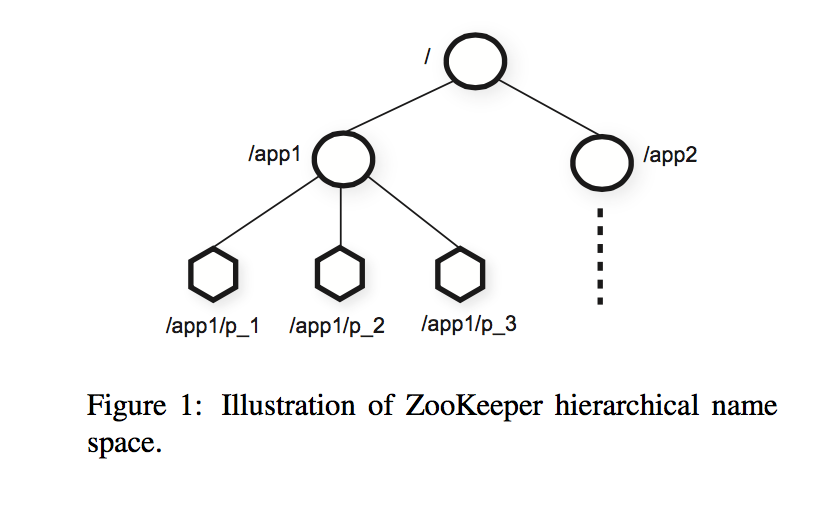
**Come funziona**

Zookeeper consente ai processi distribuiti di coordinarsi tra loro attraverso un namespace gerarchico dei registri dei dati condiviso, chiamati znodes.

Ogni znode è identificato da un percorso, in cui gli elementi del percorso sono separati da “/”, come in un file system normale.

La particolarità è che l’albero viene caricato in memoria, quindi la velocità di accesso ai nodi è molto alta.

Questa architettura consente a Zookeeper di fornire throughput e disponibilità elevati con bassa latenza, sebbene la dimensione del db gestibile sia limitata dalla memoria.



**Modalità replicato**

È possibile lanciare il server Zookeeper in modo replicato per aumentare l’affidabilità e la fault tolerance.

È robusto, poiché i dati persistenti sono distribuiti tra più nodi (l’insieme) e un client si connette a uno di essi (ovvero un "server" specifico), migrando se un nodo fallisce; fintanto che una rigida maggioranza di nodi funziona, l'insieme di nodi ZooKeeper è attivo. In particolare, un nodo master viene scelto dinamicamente per consenso all'interno dell'insieme; se il nodo principale ha esito negativo, il ruolo del master passa a un altro nodo.

Il tutto è gestito automaticamente da Zookeeper stesso.

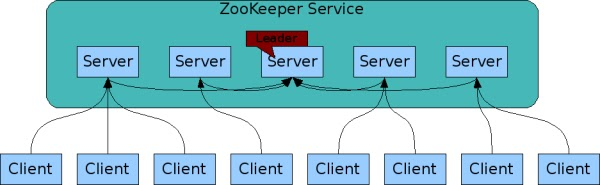
Il master è l'autorità di riferimento per le scritture: in questo modo le scritture possono essere garantite come persistenti nell'ordine, cioè le scritture sono lineari. Ogni volta che un client scrive nell'insieme, la maggior parte dei nodi mantiene le informazioni: questi nodi includono il server per il client e ovviamente il master. Ciò significa che ogni scrittura rende il server aggiornato con il master. Significa anche che non è possibile avere scritture simultanee.

La garanzia di scritture lineari è la ragione per cui ZooKeeper non funziona bene per i carichi di lavoro dominanti in scrittura. In particolare, non dovrebbe essere utilizzato per lo scambio di dati di grandi dimensioni, come i media.

Finché la comunicazione coinvolge dati condivisi, ZooKeeper aiuta molto. Quando i dati possono essere scritti contemporaneamente, ZooKeeper si mette in mezzo, perché impone un rigoroso ordinamento delle operazioni anche se non strettamente necessario dal punto di vista degli autori. Il suo utilizzo ideale è per il coordinamento, in cui i messaggi vengono scambiati tra i client.

Invece il campo in cui eccelle Zookeeper sono le letture simultanee poiché sono servite dal server specifico a cui si connette il client. Tuttavia, questo è anche il motivo dell'eventuale coerenza: la "vista" di un client potrebbe essere obsoleta, poiché il master aggiorna il server corrispondente con un ritardo limitato ma non definito.

Però per l’uso di essa come repository di file di configurazione, si suppone che ci siano poche scritture, in particolare quelle consecutive, dando la possibilità di aggiornarsi sempre correttamente.



**Obiettivo del progetto**

Implementare un semplice sistema di gestione della configurazione di alcuni server http distribuiti con l’utilizzo di Zookeeper.

La distribuzione di configurazioni dovrebbe riguardare tanti server separati ciascuno dei quali ha una configurazione diversa dalle altre.

Le configurazioni dei vari server http dovrebbero risiedere in un repository rappresentato/gestito da un server Zookeeper.

Quando una configurazione viene cambiata nel repository il server interessato deve essere notificato del cambiamento, attivarsi per ricevere e installare la nuova configurazione.

Per ottenere questo accanto ad ogni server http da gestire deve essere installato un client Zookeeper che scarichi la nuova configurazione e la installi.

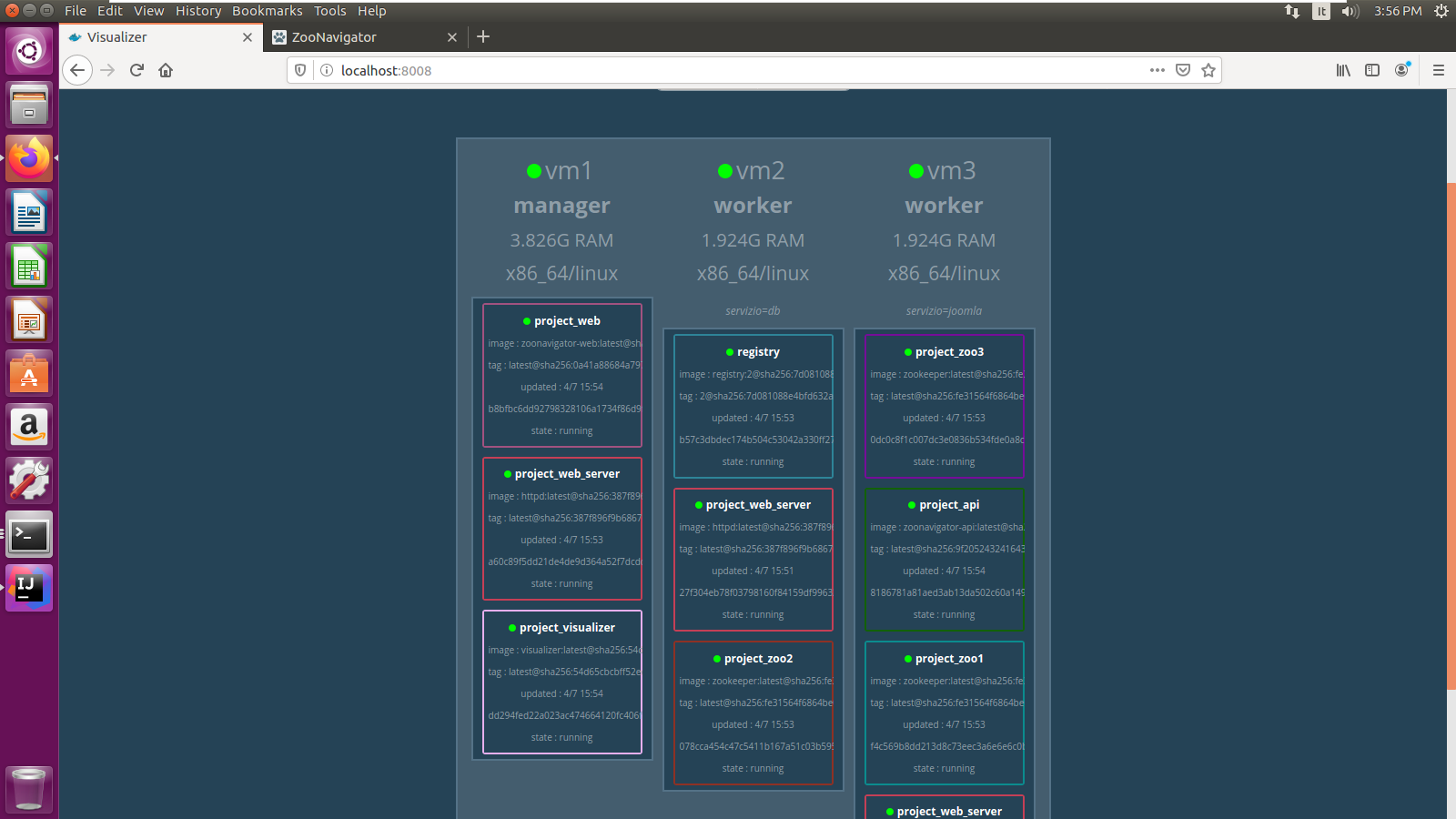
Non è importante quale insieme di server viene gestito ma il fatto che la loro configurazione venga aggiornata e gestita tramite un sistema client/server Zookeeper.

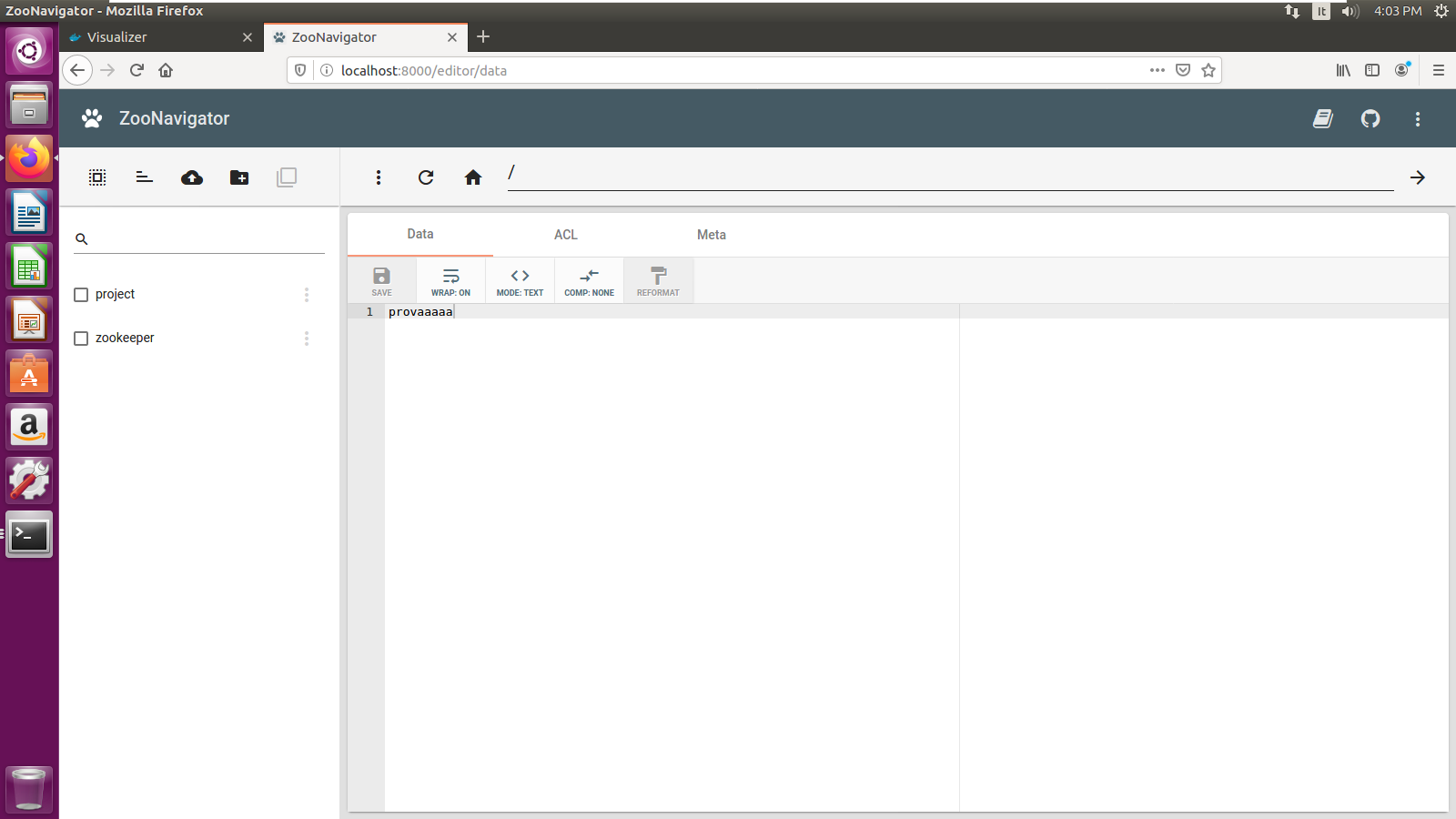
**Svolgimento del progetto**

Prima di tutto sono state create 3 macchine virtuali ubuntu per installare i vari server con configurazione da gestire in modo centralizzato.

Successivamente sono stati collegati i 3 nodi creati nel docker swarm, per usare più tardi come rete per il deployment dei vari servizi condivisi.

I servizi condivisi principali sono il **server Zookeeper**, l’**API Apache Curator** usato dall’interfaccia grafica, l’interfaccia web grafica **ZooNavigator** per gestire il server Zookeeper e infine **Visualizer** per vedere i servizi nei vari nodi in modo semplice.





Successivamente su ogni nodo è stato deployato sempre con docker, un servizio web, diversi tra loro, ma tutti basati su apache web server.

La parte client di Zookeeper per gestire la configurazione dei vari server è stato sviluppato in java ed esportato come jar.

Il jar prende come parametri il path dello znode su cui ascoltare e il path del file di configurazione da aggiornare.

In questo modo è possibile creare znode separati per ogni server e di conseguenza creare notifiche solo per i client interessati, tutto in modo asincrono.

Quindi sul server Zookeeper saranno presenti tanti nodi quanti sono i diversi server http da gestire.

L’aggiornamento è automatico: una volta che si riceve dal server zookeeper una notifica di evento, il client creerà un thread che avrà il compito di modificare il file di configurazione e aggiornare il servizio web server associato, sempre tramite la tecnologia docker.

La tecnologia docker permette di aggiornare tutti i servizi con il comando

docker stack deploy --compose-file docker-compose.yml <name>

il quale prima di aggiornare, verrà fatto un check per vedere se ci sono state modifiche rispetto al lancio precedente e in caso positivo aggiornarlo effettivamente.

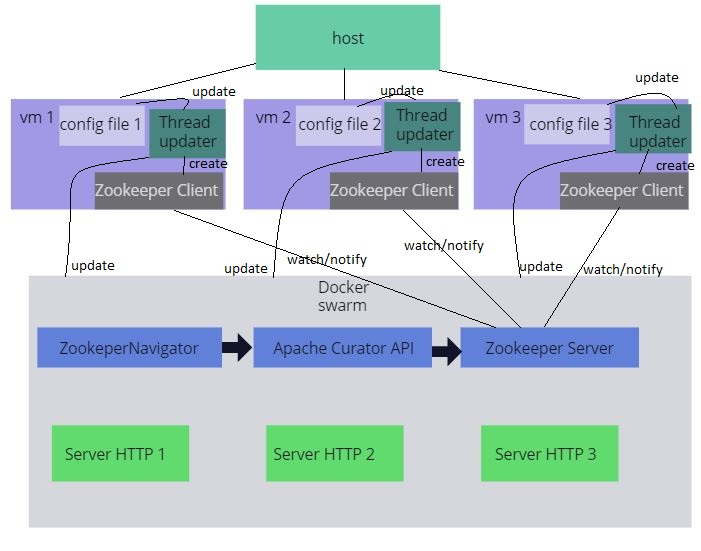
Una volta ricevuta la notifica, l’evento verrà loggato sul terminale in esecuzione.

Il core del client Zookeeper sviluppato è quindi:

* Il thread principale chiama un’api asincrono di Apache Curator e si mette in attesa di una notifica, ricevuta la notifica ripeterà in loop le stesse azioni.
* L’API chiamato creerà un watcher sul nodo prescelto e un thread che resterà in attesa dell’evento del nodo in questione, ricevuta la notifica creerà un thread Updater e notificherà il thread principale.
* thread Updater è il componente che aggiorna file di configurazione e server dopo aver letto la nuova configurazione dal server.

E’ possibile lanciare il server zookeeper in modo replicato, rendendo il sistema ancora più affidabile e fault tolerant con le caratteristiche descritte precedentemente, stessa cosa è possibile per i web server apache con gli stessi miglioramenti.

Ecco lo schema riassuntivo del progetto per rendere più chiaro il flusso del sistema.



Per la demo è stata mostrata la possibilità di lanciare i client Zookeeper su ogni server in cui gira apache server.

In seguito, è stata eseguita una modifica tramite interfaccia grafica, essendo un servizio centralizzato e condiviso tra i 3 nodi, si poteva fare da qualunque nodo.

Dopo la modifica, il server provvede a mandare una notifica ai client associati al znode modificato, i quali mettono in moto il meccanismo descritto precedentemente per aggiornare file di configurazione e server http.

**Spring Cloud Config**

Anche questo framework permette di creare il config Server e il config Client.

Per configurare un’applicazione Spring Boot come config server, basta mettere una annotation ***@EnableConfigServer***.

Successivamente mettere le configurazioni di questo server nel file di configurazione application.properties o application.yml, le sintassi cambiano leggermente.

Per il test su un repository Github privato sono stati inseriti le seguenti proprietà:

* spring.cloud.config.server.git.uri: per assegnare l’URL della git repository di riferimento.
* spring.cloud.config.server.git.username: username dell’account git
* spring.cloud.config.server.git.password: password dell’account git
* server.port: possiamo scegliere la porta in cui far girare il servizio config server, nel test è usato 8888.

Dopo di ciò il server è stato associato al repository, che può essere Github, Gitlab etc…

Per testare il funzionamento effettivo del server è stato mandato una richiesta GET ***http://localhost:8888/<nome applicazione>/default***, il quale ritorna tutto il contenuto del file <nome applicazione>.properties contenuto nella repository.

Successivamente è stato configurato il config Client, sempre in modo analogo tramite il file bootstrap.yml:

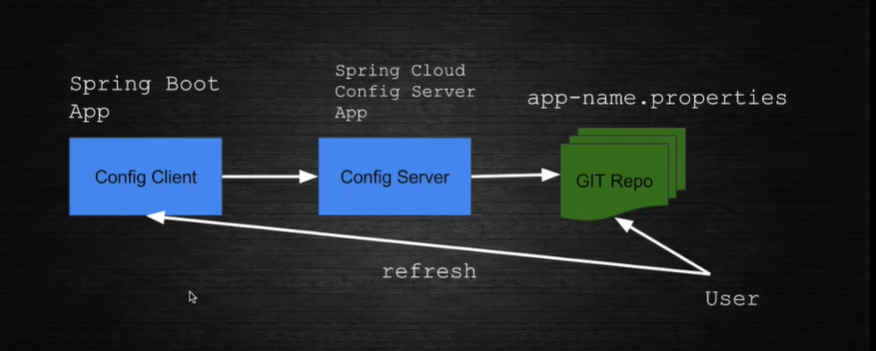
* spring.cloud.config.uri: url dove gira il config server, nel test quindi è http://localhost:8888
* spring.application.name: nome del microservizio usato per il file di configurazione, nel caso del test “application”

Il problema è che il Client a differenza del Server, non legge a comando da git, legge solo durante il boot attraverso il Server.

Per questo bisogna aggiungere l’annotation ***@RefreshScope*** per rendere possibile il refresh dinamicamente, senza riavviare il servizio.

Questo refresh viene portato con il comando ***curl localhost:8080/actuator/refresh -d {} -H "Content-Type: application/json"***.

Sfortunatamente il refresh è manuale e vale per il client scelto soltanto.

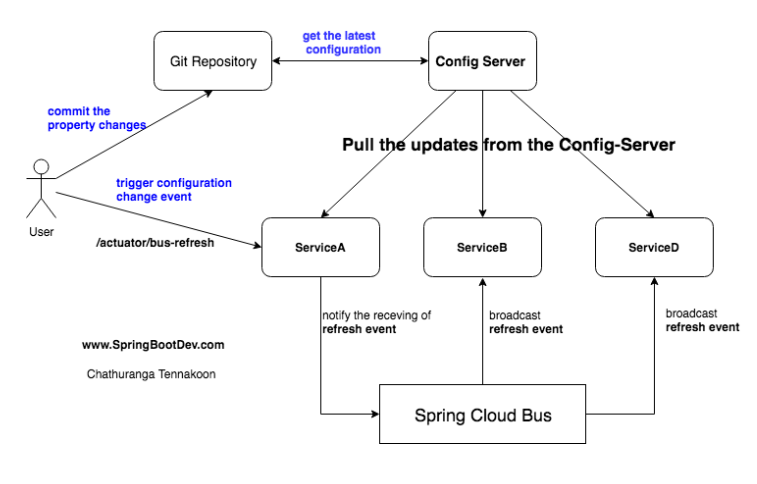


Una soluzione a questo problema è usare lo spring cloud bus, che sostanzialmente dopo il refresh del Client scelto, l’evento viene propagato a tutti gli altri servizi interessati, così da non farlo manualmente così tante volte.

Lo Spring Cloud Bus è costruito su Spring Cloud Stream e si appoggia su ad un broker (RabbitMQ, Kafka...) e viene usato come un Attuatore distribuito.

Il comando da dare per il refresh cambia quindi in ***/actuator/bus-refresh***.

L’architettura finale è la seguente:



Seguendo questo schema, basta mettere il Config Server e lo Spring Cloud Bus nel Docker Swarm per avere il sistema simile a quello mostrato con Zookeeper.

**Confronto tra Zookeeper e Spring Cloud Config**

Spring Cloud Config Server si adatta molto bene alle applicazioni Spring, ma può essere usato anche in applicazioni scritti in altri linguaggi.

Il concetto di questo framework è lo stesso di quello di Zookeeper, ma implementato in un altro modo.

Con Spring Cloud Server, abbiamo un config server (di default git repository), dove ad ogni push, il server può utilizzare un approccio event driven attraverso il cloud bus per avvertire vari client, in modo tale da aggiornare le loro configurazioni.

Questo permette ai servizi di essere sempre disponibili, senza dover riavviare il servizio al contrario di Zookeeper, che invece deve purtroppo farlo, ma per contro lo sviluppatore deve manualmente creare l’evento da propagare, mentre in Zookeeper tutto era automatico.

Ciò significa che un servizio che ha bisogno di tanto tempo per ripartire è meglio usare l’approccio con Cloud Cloud Config.

Zookeeper tiene traccia delle varie configurazioni attraverso i nomi dei Znode, mentre Spring Cloud tiene traccia con i nomi del file sul repository.

Questo approccio è molto più scalabile in caso si abbia una disponibilità di memoria non molto alta. Rispetto a Zookeeper, è meno performante in lettura, ma la lettura si fa solo 1 volta ad ogni aggiornamento del server nel nostro caso, quindi è accettabile perdere un po’ in performance per aumentare la **scalabilità** e la **disponibilità**.

Purtroppo, nell’architettura sopra descritta, la Config Server non è fault tolerant, essendo centralizzato è importante che lo sia, si può inserire un load balancer tra i client e i server replicati per ovviare a questo problema.

La documentazione di Spring è molto più vasta rendendo la vita dello sviluppatore più facile, l’uso è anche più standard, avendo solo tanti file da configurare.

Infine, usando una git repository, si ha di conseguenza la versioning di tutte le configurazioni.

Quindi Spring ha dalla sua parte anche la **facilità d’uso** e la **comodità**.

**Fonti**

<https://zookeeper.apache.org/>

<http://racyzone.blogspot.com/2016/05/dataflow-beam-vs-spark.html>

<https://qastack.it/programming/3662995/explaining-apache-zookeeper>

<https://cloud.spring.io/spring-cloud-config/reference/html/>

<https://spring.io/guides/gs/centralized-configuration/>

<https://springbootdev.com/2018/07/17/spring-cloud-config-refreshing-the-config-changes-with-spring-cloud-bus-part-2/>