

On the way to...

Apache ZooKeeper

di Ilaria Ciavatti

Cos'è ZooKeeper?

ZooKeeper è un servizio che permette di coordinare i processi. Nel momento in cui ci si trova in un sistema distribuito, per i processi sarà necessario concordare su valori di determinati dati in presenza di processi che possono fallire (*consensus problem*).

ZooKeeper mette a disposizione gli strumenti per effettuare tale **coordinamento**.

Come ci si coordina usando i servizi di ZooKeeper?

”ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.”

ZooKeeper mette a disposizione un *namespace*, molto simile ad un *file system* Unix.

Gli elementi del namespace sono dei nodi, *znode*, e la loro organizzazione è ad albero. Tutti i processi del sistema possono comunicare, creando scrivendo e leggendo informazioni dagli *znode*.

Tipologie di znode

Gli znode sono quindi i “tasselli” dello spazio di nomi, costituiscono i percorsi o path, contengono anche dati di modesta dimensione. Ci sono due tipi di znode: persistent znode ed ephemeral znode.

- Persistent znode, esiste nello spazio dei nomi finché non viene esplicitamente cancellato.

- Ephemeral znode, viene cancellato quando termina la sessione del client che l'ha creato.

Un attributo per i nodi

Gli znode, sia quelli “*effimeri*” che quelli “*permanenti*”, possono essere sequenziali o meno. Gli znode sequenziali hanno la caratteristica di essere etichettati con numeri **non randomici**, ma appunto, sequenziali.

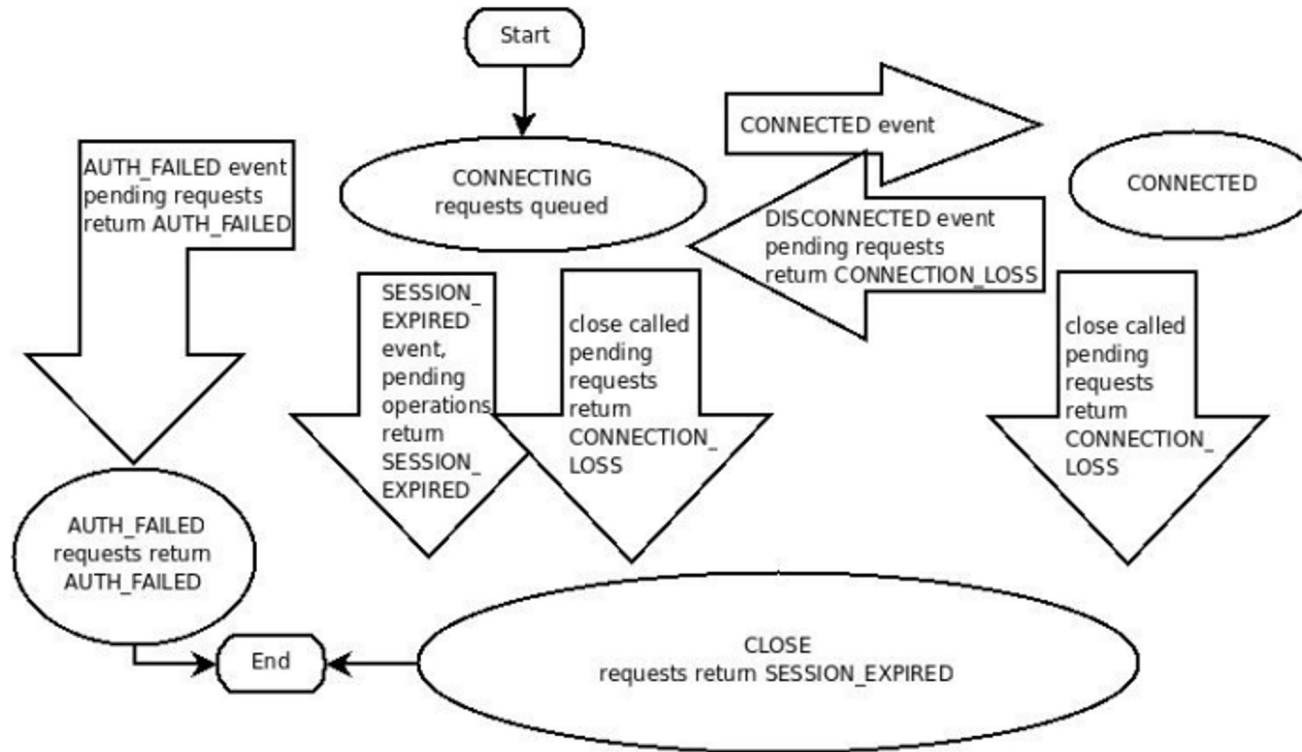
Nel momento in cui il nodo viene creato gli viene attribuito un numero che è **sequenziale** rispetto al numero del nodo **genitore**.

(ci sarà un esempio fra qualche slide)

Le sessioni in ZooKeeper

Un client ZooKeeper stabilisce una sessione con il servizio fornito da ZooKeeper usando una API . Tale API è composta da “glue code”, che serve ad adattare diverse parti di codice. Ciò che viene creato è un “**handle**” del servizio che inizialmente si troverà nello stato **CONNECTING**. Nel momento in cui il gestore (o handle) è connesso al server ZooKeeper, lo stato passa a **CONNECTED**. Nel caso in cui avvenga un qualche tipo di errore, l'*handle* passerà nello state **CLOSED**.

Le sessioni in ZooKeeper



Caratteristiche degli znode

Ogni nodo può memorizzare fino ad 1 MB di dati, ogni nodo ha una sua ACL (*access control list*). The ACL permissions are the following:

.WRITE

.CREATE

.DELETE

I watch in ZooKeeper

In ZooKeeper i watch si utilizzano per implementare un pattern di comunicazione asincrona fra i processi. Si tratta del pattern “**publish/subscribe**” (pubblicare/abbonarsi), brevemente, in tale design ci sono due ruoli che i processi assumono:

- pubblicano un contenuto (che può interessare terze parti)
- si abbonano al contenuto (ricevono una notifica quando appare il contenuto di loro interesse)

I watch in ZooKeeper

Come detto precedentemente, il *namespace* di ZooKeeper è una raccolta di informazioni sottoforma di collezione di znode. I dati memorizzati dentro questi nodi possono cambiare.

Nel momento in cui il cambiamento si verifica, tutti i client che hanno posto un **watch** su quello specifico znode.

NOTA: sebbene, come detto prima, il meccanismo del watch è un'implementazione del pattern publish/subscribe, ha delle particolarità. E' una **one – time** operation, l'abbonamento è relativo necessariamente a **qualsiasi** tipo di cambiamento.

Caso di studio, creazione di una queue

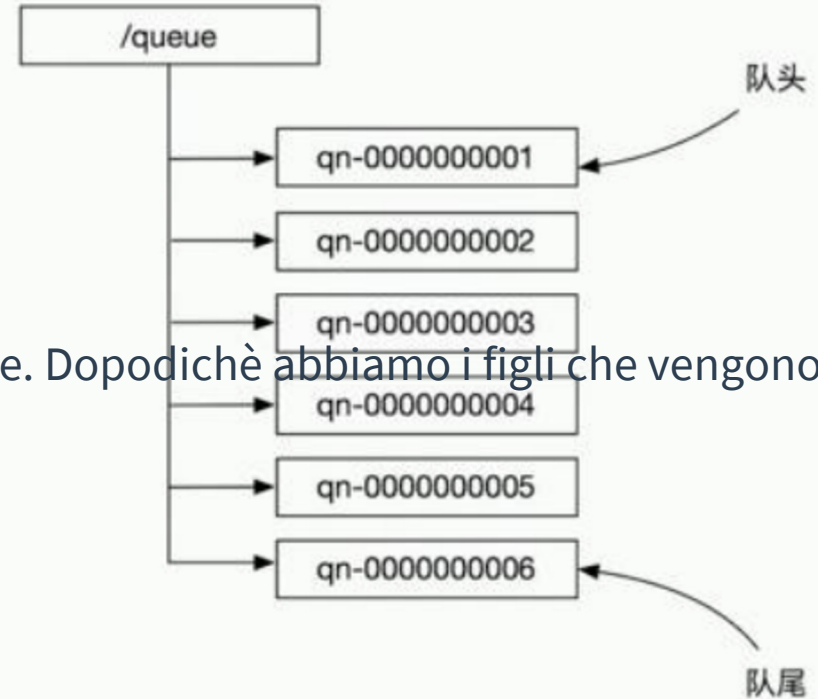
In questo caso di studio useremo i nodi sequenziali (o *sequential znode*) e quelli effimeri (o *ephemeral znode*). Questo tipo di nodi sono infatti utili per creare delle strutture dati, come ad esempio la coda (o *queue*).

Le operazioni possibili sulla coda sono, come sappiamo, l'aggiunta e la rimozione di un elemento, seguendo l'ordinamento FIFO (*first in first out*).

Per creare tale struttura il primo passo è la creazione di un root node (nodo radice).

Caso di studio, creazione di una queue

Il nodo radice in questa immagine di esempio è /queue. Dopodichè abbiamo i figli che vengono aggiunt



Possibile scenario : elezione di un leader (Leader election)

In questo *case scenario* proveremo a utilizzare il servizio offerto da ZooKeeper per coordinare dei processi che vogliano accordarsi sull'elezione di un leader.

L'idea è quella di usare la tipologia di znode “ephimeral”, per segnalare la presenza di uno specifico processo (ad ogni processo presente e attivo corrisponde uno znode effimero) e l'attributo “sequential” (ogni processo ha un numero in base al quale si può ordinare il suo arrivo rispetto al momento di arrivo degli altri processi).

Possibile scenario : elezione di un leader (Leader election)

Il sistema di elezione del leader prevede che il processo a cui corrisponde il numero sequenziale più piccolo diventa il leader.

Ogni processo controlla la presenza dello znode con il numero sequenziale inferiore al suo tramite un watch. In caso scompaia dal sistema dovrà verificare, tramite `getChildren()`, di non essere diventato il processo con lo znode con numero più piccolo. In caso, diventa il nuovo leader. Invece se tale condizione non è verificata, si limiterà a porre un watch sull'attuale znode con numero di sequenza inferiore al suo e meno distante (la cui sottrazione fra i due numeri di sequenza sia la minima possibile).

Possibile scenario : elezione di un leader (Leader election)

Per quanto lievemente macchinoso, questo posizionamento dei watch ci permetterebbe di evitare l'effetto *orda* (herd effect) che avverrebbe se i watch dei processi fossero tutti posti sul leader attuale. Infatti nel momento in cui lo znode del leader sparisse, tutti i processi del sistema effettuerebbero una `getChildren()` , diventando un'*orda* (mancata scalabilità).

Il tempo fisico (physical time)

Il tempo fisico di un computer è il tempo stimato da un orologio fisico (physical clock). Il tempo fisico in un sistema distribuito è in genere creato grazie alla sincronizzazione collettiva su una specifica istanza di tempo, considerata come *universale*.

Tale *universalità* è ottenuta tramite la continua *sincronizzazione*.

Per ovvie ragioni non è un modo conveniente per tenere traccia del tempo.

Il tempo logico (logical time)

La nozione del tempo logico è invece basata sugli orologi *logici*, o “*logical clock*”.

Per definizione:

- Un sistema di orologi logici consiste in un tempo del dominio T e un orologio logico C . Gli elementi di T formano un insieme parzialmente ordinato sopra la relazione $<$.
- La relazione $<$ è chiamata “*happen before*”=”successo prima”
- L’orologio logico C è una funzione. Prende un evento e e lo mappa sul dominio T . $C(e)$ è chiamato il *timestamp* di e .

Il tempo logico (logical time)

$$C: H \rightarrow T$$

dati due eventi e e j , $e \rightarrow j \Rightarrow C(e) < C(j)$

In tal modo l'orologio logico C viene usato per determinare un ordinamento fra gli eventi. Tale ordine è **parziale** quando alcuni eventi non possono essere comparati.

Il tempo logico in ZooKeeper

I valori utilizzati per implementare il tempo logico, sono gli *zxid*. Un evento in questo contesto è una transazione (un'operazione di modifica di un elemento nel namespace). L'algoritmo Zab ha tre fasi. La terza fase è quella in cui il leader processa la transazione. Quando un leader riceve una transazione sceglie uno *zxid* per quello specifico evento. Questo *zxid* dovrà necessariamente essere successore di tutti gli *zxid* di tutte le transazioni già presenti nel *transaction log*.

L'ordine totale di ZooKeeper

Poichè è il leader a scegliere gli *zxid*, esso stesso stabilisce l'ordine. Quindi se una transazione A è effettuata prima di una transazione B dal un server del sistema è perché, in un momento precedente, A è stata effettuata prima di B dal leader. Quindi in sostanza, A viene effettuata prima di B su ogni server. L'ordine implementato da ZooKeeper, è **totale**.

Un po' di storia...

Fra i predecessori dell'algoritmo usato da ZooKeeper (zab), vi è senz'altro Paxos. Si tratta di un algoritmo di consenso (Consensus Algorithm) laddove il consenso è definito come l'essere concordi su un determinato valore di un dato, da parte dei processi del sistema distribuito. Il problema è quindi definito dalle seguenti caratteristiche:

- Il valore del dato viene proposto da uno fra i processi
- I processi possono proporre possibili valori

Solo uno fra i valori proposti dovrà essere scelto..

- Un valore non potrà essere scelto se non è stato proposto (scelto $x \rightarrow$ proposto x)
- Se un valore è stato scelto allora deve essere possibile, per un processo, impararlo (imparato $x \rightarrow$ scelto x)
- Un valore non può essere imparato se prima non è stato scelto (scelto $x \rightarrow$ imparato x)

Valori di verità dell'implicazione:

scelto $x \rightarrow$ proposto x

ovvero

x è stato scelto e x è stato proposto : True

x è stato scelto e x non è stato proposto: False

x non è stato scelto e x è stato proposto: True

x non è stato scelto e x non è stato proposto: True

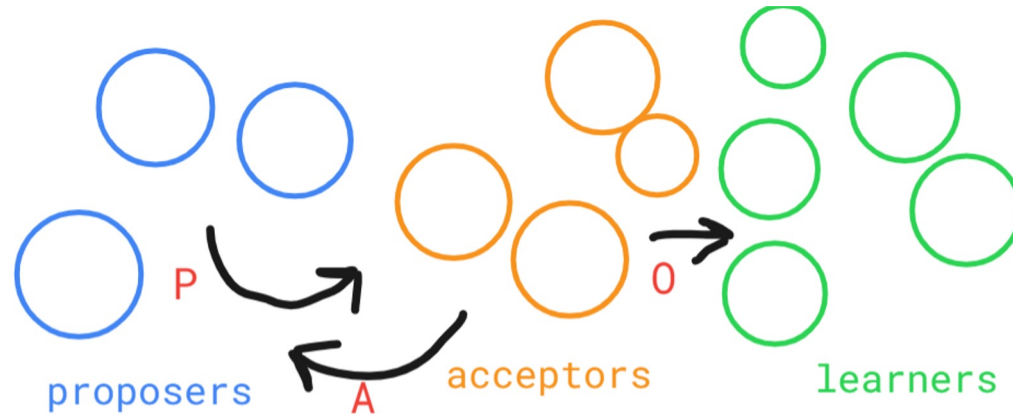
In un questo “consensus algorithm”...

ci sono essenzialmente tre azioni che si possono compiere ai fini del raggiungimento del consenso. Si può **proporre** un valore, si può **accettare** un valore (un valore viene poi **scelto** quando un insieme sufficientemente grande di **acceptor** lo ha precedentemente accettato), infine si può **imparare** un valore.

I ruoli sono *proposer*, *acceptor*, e *learner*.

Gli agenti (entità presenti nel sistema distribuito) possono comunicare fra loro inviandosi messaggi.

Interazione fra proposer, acceptor e learner in Paxos (flusso)



P = P è la proposta inviata da un proposer ad una maggioranza di acceptor.

A = A è la risposta che un acceptor invia allo specifico proposer, per comunicare che la proposta è stata accettata.

O = O è l'outcome, ovvero la proposta accettata da uno specifico acceptor, inviata ad uno specifico learner. (Data una proposta accettata essa non viene inoltrata a tutti i learner)

Caratteristiche di Paxos

L'algoritmo determina la scelta della transazione in base al numero ad essa associato. La regola da rispettare (da parte degli accettatori) è quella di non accettare transazioni con numero inferiore all'ultimo accettato.

Nello schema precedente abbiamo visto come ogni accettatore comunicherà con un solo learner (dopo aver accettato la transazione n). Tale learner è un *distinguished learner* (ovvero un learner che fa parte di un gruppo che procederanno ciascuno ad avvisare un sottogruppo di learner). Infatti fare comunicare l'esito da parte degli accettatori a tutti i learner avrebbe dato luogo a un notevole numero di

Il cuore di Zab (protocollo implementato dai server ZooKeeper)

Quando un client vuole leggere uno specifico dato, può chiedere ad un qualsiasi nodo del gruppo di host (su cui i dati sono replicati).

Per l'operazione di scrittura è invece più complicato. Una operazione di questo tipo implica la modifica dei dati. Lo svolgimento della richiesta è più complesso.

Solo il leader può svolgere l'operazione di scrittura. Inoltre per svolgere correttamente tale operazione, è necessario seguire il protocollo.

Fase 0, elezione del “prospective” leader

In questa fase i “peer”, i server che fanno parte dell’insieme che adotta ZooKeeper, sono inizializzati con lo stato “*election*”.

Non è necessario implementare uno specifico protocollo di elezione, l’unico vincolo è che il protocollo adottato termini (con buona probabilità) scegliendo un server funzionante con il consenso della maggioranza dei *peer* (quorum).

Quindi un prospective leader è un leader votato da un quorum di server.

Fase 1, la scoperta

La fase di scoperta, o “*discovery*”, è la fase in cui il *prospective leader* acquisisce informazioni tramite i suoi *follower* (il quorum di *peer* che l’ha scelto). Tramite questa fase verrà selezionata e memorizzata come “*history*”, la sequenza più aggiornata di transizioni. Dove per *history* si intende la serie condivisa di eventi/transizioni. Ovvero il fulcro e l’obiettivo di un protocollo di coordinamento in un sistema distribuito.

Fase 2, la sincronizzazione

La fase di sincronizzazione prevede la diffusione della history. Quindi il *prospective leader* comunica ai *follower*, proponendo le transizioni dalla sua *history* (costruita nella fase precedente di *discovery*, o scoperta). In sostanza quindi si tratta della sequenza di eventi più aggiornati proveniente da uno dei follower del leader, la quale viene ora diffusa da chi ha “autorità”, il *prospective leader*.

Tutti i follower che hanno una history che deve essere aggiornata (e che quindi verrà modificata in questa fase) inviano gli *acknowledgment* al leader, che quindi diventa *established leader* (e non più *prospective*).

Fase 3, il broadcast

Questa è la fase in cui si permane indefinitamente fino al verificarsi di un *crush* del sistema. Ciò che avviene è che un client fa una request per una “write” (quindi una transazione), il server con il ruolo di leader (*established*) si occupa di fare il broadcast di questa nuova transazione a tutti i suoi follower.

In questa fase il leader accetta nuovi eventuali follower.

La fase di broadcast è la fase in cui vengono gestite le modifiche di stato (ovvero inserimento di nuove transazioni nella history).

Riflessioni

Dato il teorema **CAP**, in cui un database distribuito non può esporre tutte e tre le seguenti proprietà:

- Consistenza (consistency)
 - **Disponibilità** (availability)
 - Tolleranza alle partizioni (partition tolerance)
- E posto che la tolleranza alle partizioni è irrinunciabile (è necessario tollerare la eventuale perdita di messaggi e gli eventuali malfunzionamenti). Possiamo osservare come la consistenza sia fondamentale (dopotutto ZooKeeper è uno strumento di coordinazione

Riflessioni

ZooKeeper è un middleware?

ZooKeeper, a mio avviso, è più uno strumento di coordinazione che non un vero e proprio middleware. Nonostante si trovi posizionato fra l'applicazione e il sistema operativo (tramite le librerie con cui scrivere “glue code”) non è ricco di funzionalità e non offre particolari astrazioni.

Lo strano caso di ZooKeeper e Storm Cluster

Apache Storm è un sistema di calcolo distribuito in tempo reale per l'elaborazione di grandi volumi di dati ad alta velocità. L'architettura proposta da Apache Storm è quella di una topologia con nodi con compiti differenti. Per topologia si intende essenzialmente un grafo. I possibili tipi di nodi sono i seguenti: *Nimbus*, *supervisor*, *worker*. Uno *Storm cluster* tipicamente consiste in un nodo *master* su cui viene eseguito il demone *Nimbus* che gestisce le *topology submission*, schedula le topologie, e monitora i *worker*.

Apache Storm

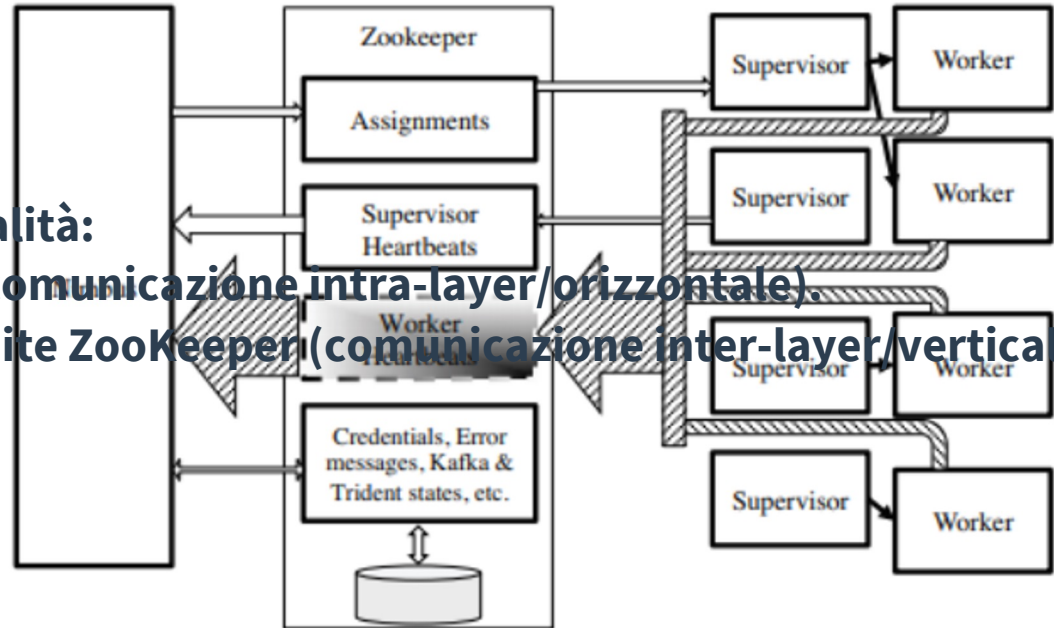
Apache Storm è un sistema di calcolo distribuito in tempo reale per l'elaborazione di grandi volumi di dati ad alta velocità. L'architettura proposta da Apache Storm è quella di una topologia con nodi con compiti differenti. Per topologia si intende essenzialmente un grafo. I possibili tipi di nodi sono dei seguenti tipi: *Nimbus*, *supervisor*, *worker*.

“Uno Storm cluster consiste in un nodo master, su cui viene eseguito il demone Nimbus che gestisce le topology submission, schedula le topologie, e monitora i worker. I supervisor sono responsabili per launching e killing dei worker in base agli assegnamenti calcolati dallo scheduler , quest'ultimo interno a Nimbus.”

Comunicazione in Storm

Figure 20: Communication between different Storm components

Storm la comunicazione avviene in due modalità:
o utilizzando delle API costruite su Netty (comunicazione intra-layer/orizzontale).
o utilizzando il ZooKeeper (comunicazione inter-layer/verticale).



Heartbeats

“Mostrano che un server è disponibile tramite l’invio periodico di un messaggio a tutti gli altri server.”

Nei sistemi distribuiti può essere essenziale rilevare la failure di uno o più componenti del sistema. In zab vengono usati gli heartbeat per chiarire lo status del sistema, ad esempio è molto importante poter verificare che il leader sia o non sia più operativo. Infatti se il leader fallisce non può più essere eseguita alcuna transazione.

Inviare periodicamente un messaggio a tutti gli altri server per garantire la propria liveness. Si deve inoltre selezionare un intervallo fra le richieste secondo i seguenti criteri:

Timeout Interval > Request Interval > Network round trip time between the

Heartbeats in ZooKeeper

La comunicazione che avviene fra *Nimbus*, i *supervisor* e i *worker* comprende la lettura degli *heartbeat*. I “battiti” sono usati per monitorare lo stato dei processi. Nimbus monitora i *battiti* di supervisor e worker. I supervisor invece li usano per monitorare i worker. Battiti e metriche vengono scritti dai worker e dai supervisor per essere quindi letti da supervisor e Nimbus. Le informazioni che interessano i supervisor sono i battiti dei worker, per conoscerne la liveness e quindi la disponibilità. Nimbus si interessa dei battiti dei supervisor e dei battiti uniti alle metriche dei worker. Queste informazioni vengono scritte e gestite dai server zookeeper (nel suo particolare “file system”)

Il problema dell'usare ZooKeeper per la memorizzazione degli heartbeat

“Once larger topologies could be executed, we began to observe ZooKeeper dropping heartbeats (due to timeouts) at around cluster size of 100 nodes (October 2013).”

Inoltre, si è verificato come il 99,2% delle scritture da parte di ZooKeeper si potevano attribuire agli attributi dei worker.

Nello specifico dei *worker* in quanto tali dati sono comprensivi di metriche (al contrario degli heartbeat dei *supervisor*).

Il problema dell'usare ZooKeeper per la memorizzazione degli heartbeat (2)

Il problema consisteva quindi nel non essere abbastanza responsive da riuscire a scrivere tutti gli heartbeat che arrivavano (dropping heartbeats) a causa dell'overload di lavoro a cui i server erano soggetti (motivo heartbeat con metriche).

Notiamo quindi che la consistenza degli heartbeat non è fondamentale e che gli heartbeat passati non hanno alcuna rilevanza al fine di monitorare il comportamento dei worker e dei supervisor. ZooKeeper non è quindi la scelta più efficiente per questo tipo di lavoro.

Pacemaker

Gli autori dell'articolo su questa problematica di Storm, descrivono il design di uno strumento più appropriato di ZooKeeper:

Pacemaker.

Nella sua più semplice implementazione i server di Pacemaker mettono a disposizione la memorizzazione key-value di cui Nimbus è il client. In tal modo Pacemaker alleggerisce il carico di lavoro dei nodi di ZooKeeper. (Ricordiamo infatti che i dati più pesanti contenenti metriche sono quelli letti da Nimbus relativi ai worker)

Pratica con ZooKeeper

ZooKeeper ensemble con tre server

Nella parte pratica si crea un *ensemble* composto da tre server.

I tre server avranno bisogno di tre ripetitive cartelle.

Per lanciare ognuno dei server ci si servirà del comando “./zkServer.sh start”.

Idea

L'idea del codice scritto è quella di lanciare n *client* che effettuano operazioni in scrittura (*write client*) e un *client* che effettua operazioni in lettura (*read client*). Mentre i *write client* richiedono ciclicamente l'esecuzione di transazioni, il *read client* testerà la reattività del servizio di lettura.

Il main()

Riceve in input il numero di client che vogliamo lanciare. Usa un executor service per lanciare i tasks. Per ogni client crea una istanza Runnable di OneZKClient (che rappresenta l'insieme dei passi di ogni client).

Classe OneZKClient

This is the client or write-client class. Each instance has its own client number (input argument of the constructor).

A OneZKClient uses a connector, instance of ZKConnect, for the connection with the ZooKeeper service.

ZooKeeper servers are three (see the section "Setting up a ZooKeeper Ensemble"), the first listen for client connections on port 2181, the second on port 2182 and the third on 2183. Each client will create a znode whose name is "/" + the respective client number, and the data is also the client number.

After that, it will delete the znode

Classe ZKConnect

Questa classe crea un oggetto ZooKeeper.

Il metodo “close” è usato per chiudere l’**handle** ZooKeeper e liberare le risorse.

Ogni Zkclient dovrà quindi creare un oggetto ZKConnect. Per effettuare le operazioni sul namespace userà quindi i metodi di questa classe: createNode, deleteNode etc

Class OneZKReadClient

Questo codice è strutturato in modo simile al codice della classe OneZKClient:

- **OneZKReadClient** è il **reader**/lettore, ovvero il client che non richiede transazioni/modifiche

- **OneZKClient** è il **writer**/scrittore, il client che richiede modifiche

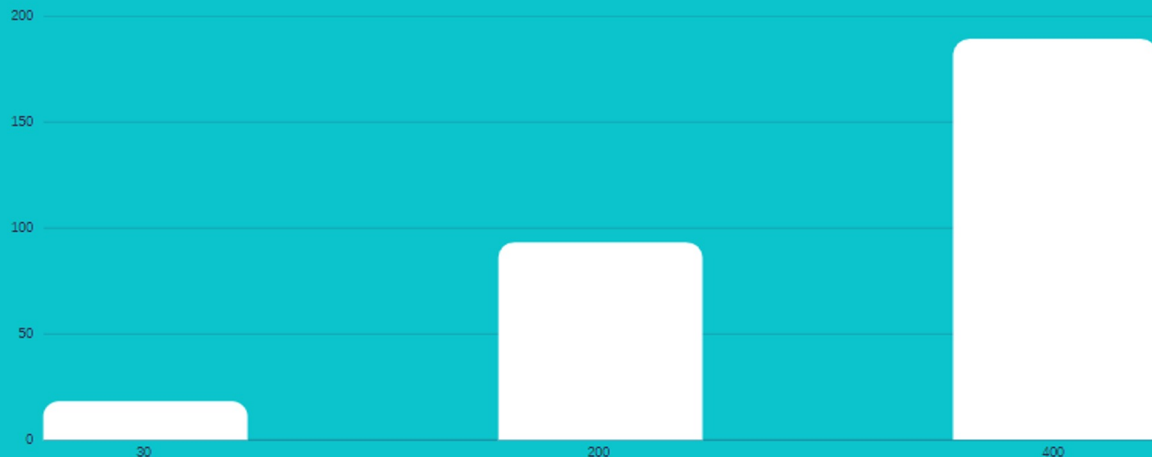
- Nel momento in cui effettua la richiesta di lettura, OneZKReadClient effettua una misurazione dell'*elapsed time* fra il momento antecedente la request e quello successivo all'arrivo della relativa risposta. Infine colleziona questi dati in un file di nome "ZKReaderStatistics.txt".

Le statistiche

PROJECT STATISTICS

Asse delle ascisse : numero di istanze di OneZKClient o scrittori.

Asse delle ordinate : il tempo medio di attesa del OneZKReaderClient (il client è uno ma effettua più iterazioni).



Conclusioni

L'ipotesi di partenza era che in una situazione in cui il servizio riceva molte richieste di transizioni, il tempo di attesa di una semplice lettura cresca anch'esso. Questo nonostante una “*read*” sia in questo contesto un'operazione molto semplice che non richiede l'intervento del leader. E che di per se è fortemente **scalabile**.

Lo stesso non si può dire, chiaramente per le scritture o “*write*”.

L'andamento (per questi numeri, relativamente bassi) non è però catastrofico come si poteva pensare. Per questi numeri cresce **linearmente** con l'aumentare del numero di scrittori.