Distributed Locking in Elasticsearch using Zookeeper

Table of Contents

Concept

What is Zookeeper?

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.
For more information visit https://zookeeper.apache.org/

Why do we need Zookeeper?

To implement fully distributed locks that are globally synchronous on a monitored object, meaning at any point of time no more than one clients think they hold the same lock.
A typical production level environment expects Elasticsearch cluster, so same plugin is installed on each node that has there own JVM and in such cases java specific synchronized blocks wont work. Hence to avoid this problems of getting lock on Elasticsearch object by multiple clients we are using one of the Zookeeper solutions called locks, for more information visit documentation

Types of locks:

In Contentserv, based on the nature of use case , and the way its triggered, locks can be classified in two types viz. Foreground lock and Background lock. The usage of the locks are described below.

These locks are acquired on a monitor object:

1. Foregound locks : Especially for the tasks for which browser based user or a synchronous REST API is waiting to finish
   E.g. Save Content, Type Switch etc.
2. Backgound locks : taken by activities those are running in the background.
   E.g. Update Searchable, KPI update, Data transfer via Bulk Propagation etc.

Foreground locks have a higher priority as compared to Background locks. The reasoning behind having a higher priority for FG Locks, is to ensure all Synchronous operations do not timeout prematurely because of BG requests.

The locking mechanism is implemented and deployed as a part of the Elasticsearch plugin.

API for Developers

Locking Client API:

```
  ZkBlockingWriteLock(String lockInfo, ZooKeeper zooKeeper, String monitorObject)
  ZkBlockingWriteLock(String lockInfo, ZooKeeper zooKeeper, String monitorObject,
                      LockTypes lockType)
```

lockInfo : Short information of current lock type
zooKeeper : Zookeeper service instance(available in ZooKeeperUtil)
monitorObject : Typically its ID uniquely identifying monitoring object
lockType : one of Foregound or Backgound lock(default is foreground)

Example :

1. **Without Zookeeper**

```
synchronized (monitorObject) {
     // Set of operations modifiying monitor obeject
}
```

2. **Using Zookeeper**

```
//Getting zookeeper service instance
ZooKeeper zk = ZooKeeperUtil.getZookeeper();
```

```
//Creating lock instance(Foregound lock type)
ZkBlockingWriteLock writeCall;
writeCall = new ZkBlockingWriteLock("Save Content", zk, monitorObject);
```

```
//Creating background lock instance
writeCall = new ZkBlockingWriteLock("Bulk Propagation", zk, monitorObject,
                                    LockTypes.BACKGROUND_LOCK);
```

```
//Synchronous block of code
writeCall.lockWithTimeout();
     // Set of operations modifiying monitor object
    // Code which will typically be written within the synchronized block
writeCall.unlock();
```

Exceptions :

1. InstanceLockException : Unable to get lock within stipulated time

2. SessionExpiredException : Connectionis idle more than session time out
3. ConnectionLossException : Connection to Zookeeper is lost
4. NoNodeException : unlock() is trying to delete non existing lock

Retry Mechanism / Dead Letter Queue:

Since the background operations have a lower priority, there is a possibility that the BG operations may have to wait for a longer duration to acquire a lock for the same object. In such cases, there is provision for retry of the operations.

This mechanism is implemented on Tomcat/ Web Application/ Interactor , and number of reties can be configured using property *elastic.request.retry.count* in ***elasticsearch-details.properties*** file. Once this retry limit is exceeded the operation / transaction is logged in a Dead Letter queue (Example, **MdmDeadLetterQueue** Kafka topic).

Configuration Properties For Zookeeper (Standalone /Cluster):

This file is present at elasticsearch.

```
# comma-separated list of host:port pairs as localhost:2181,localhost:2182,localhost:2183
zookeeper.servers=localhost:2181

# session timeout in milliseconds, It is used to do heartbeats and the minimum session timeout will be twice the tickTime.
zookeeper.session.timeout=5000

# number of retries if there is disconnection
zookeeper.connection.retry=5

# delay in milliseconds between connection try after disconnection
zookeeper.connection.delay=1000

# lock prefix for foreground tasks
# lowest the alphabet index, highest is the priority for the task
zookeeper.foregound.lock.prefix=A

# lock prefix for background tasks
# lowest the alphabet index, highest is the priority for the task
zookeeper.backgound.lock.prefix=B

# lock wait timeout for foreground tasks in milliseconds
zookeeper.foregound.task.timeout=2000

# lock wait timeout for background tasks in milliseconds
zookeeper.backgound.task.timeout=2000
```

Architecture

Lock Acquisition Algorithm:

1. Create Persistent node under root with monitor object name as path

"/monitorObject" if not exists.
Example,
/{articleInstanceID}

2. Create a Sequential Ephemeral node under monitor object node with a pathname "monitorObject/LockType-"
   In response Zookeeper gives a Number/Name to the Node. This Number/Name is always unique and sequential.

   Example,
   Request: /{articleInstanceID}/LockType-
   Response: /{articleInstanceID}/LockType-0000000001

3. Get all the children of monitor object and sort them.

4. If the pathname created in step **2** is the first element i.e.which has the lowest sequence number suffix and private lock is available, the client has acquired lock and can proceed further.

5. All other client watch parent node(created by step **1)** for the step **4** to finish.

6. Repeat step **3**

Lock Releasing Algorithm:

1. clients wishing to release a lock simply delete the node they created in step **2** of lock acquisition algorithm.

Understanding Zookeeper with the help of examples

Terminology :

1. **Client** - Its a Zookeeper client, and always instantiated within Elastic search plugin

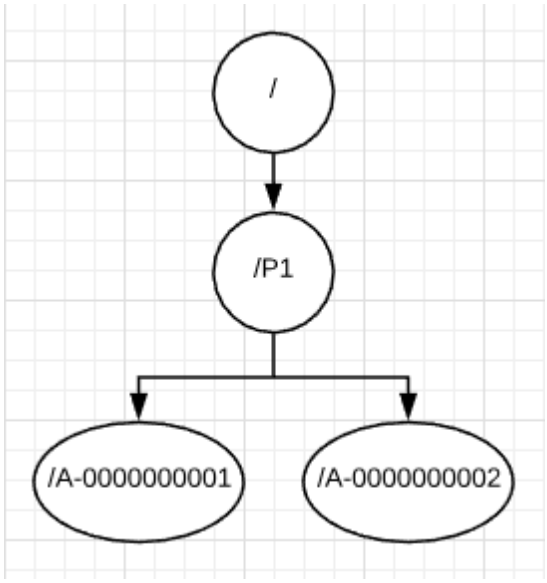
2. **Persistence Znode** - this node will be alive even after the client, which created that particular node is disconnected.
3. **Ephemeral Znode** - this type of nodes are active until the client is alive
4. **Sequential Znode -** Sequential nodes can be either persistent or ephemeral. When a new znode is created as a sequential znode, then ZooKeeper sets the path of the znode by attaching a 10 digit sequence number to the original name.

For more information, please follow this link.

Scenario #1: Save Article as Foreground operation

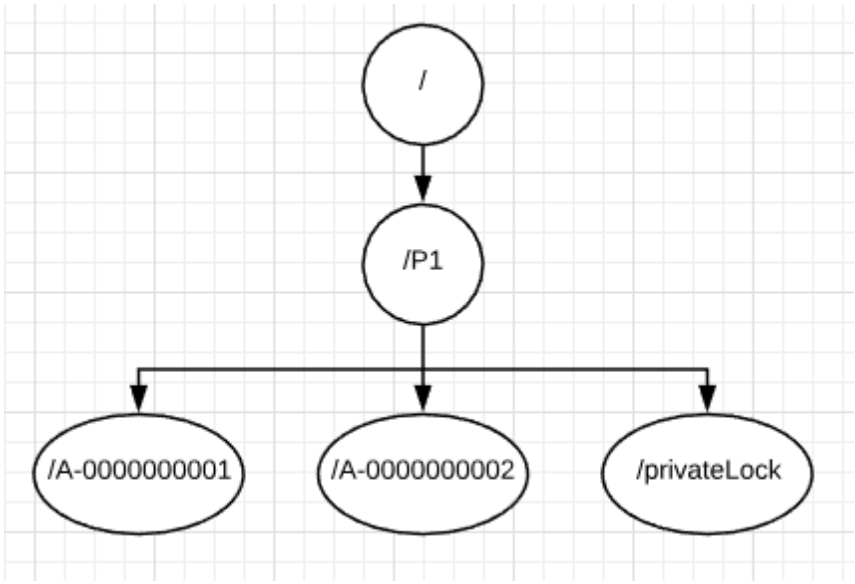
All locks/nodes created by foreground operations are prefixed by 'A-'

1. Lets say Client1 and Client2 wants to save same article P1.
2. One of the clients will create Persistent node P1.
3. In a race condition both clients creates Sequential Ephemeral node under monitor object node P1
   i.e. Client1 and Client2 creates node A-0000000001 and A-0000000002 respectively

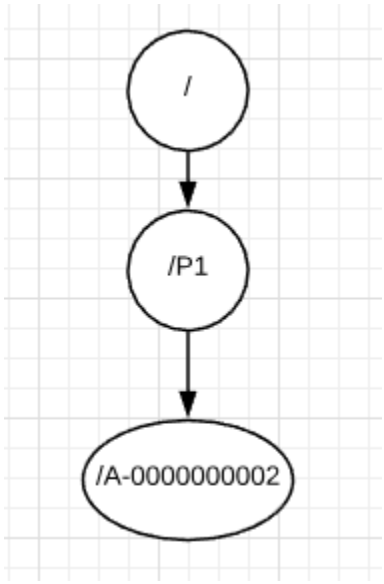


4. Client1 and Client2 gets all the children under P1 and sorts them. Since node /A-0000000001 has lowest sequence number compared to other child nodes, which is created by Client1 is eligible to acquire a lock.
5. Client1 checks for the /privateLock under P1, if there is no such lock it will create

one and proceed with task execution



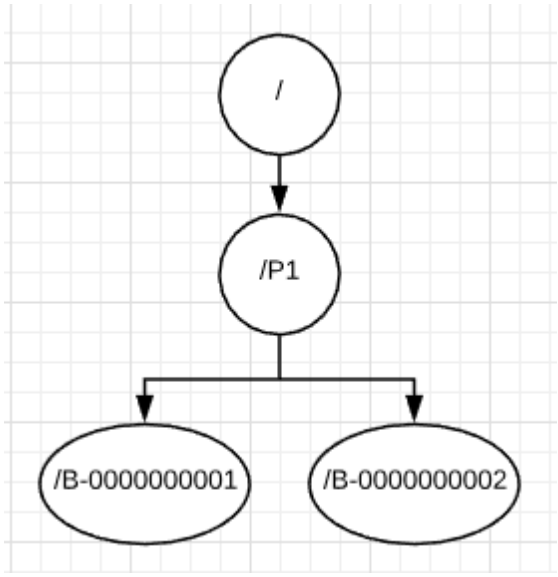
6. By this time Client2 will be awaiting for Client1 to finish
7. As soon as Client1 finishes its task it deletes its node and privateLock. In this case /A-0000000001 and /privteLock will be deleted

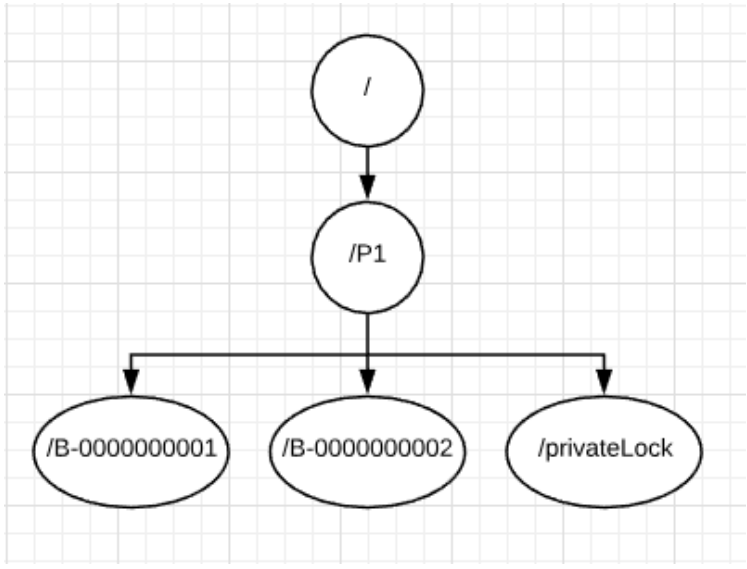


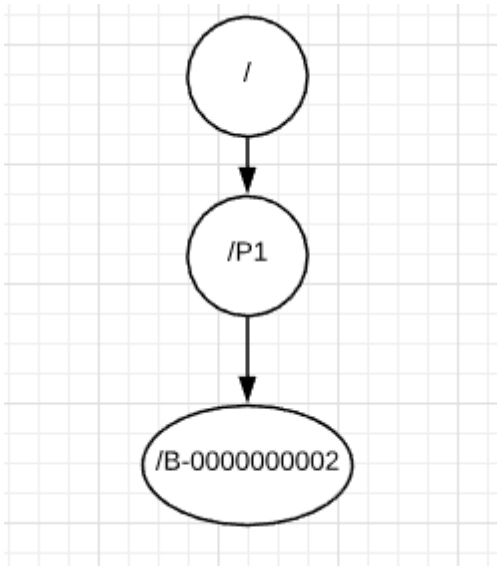
8. After this Client2 gets notified by Zookeeper and it performs step 4, 5 and 7

Scenario #2: Bulk propagation as Background operation

All locks/nodes created by background operations are prefixed by 'B-'. Locking mechanism by background operations works similar to foreground operations.



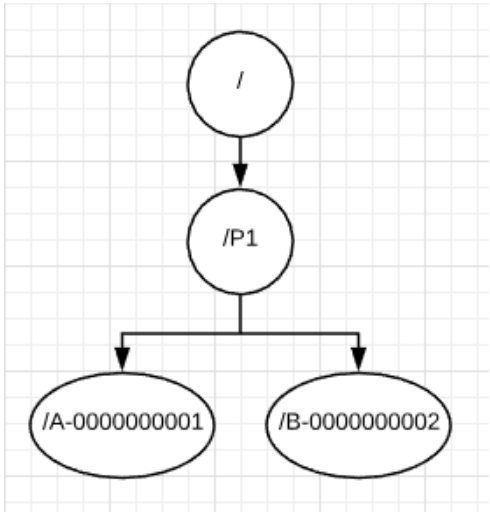




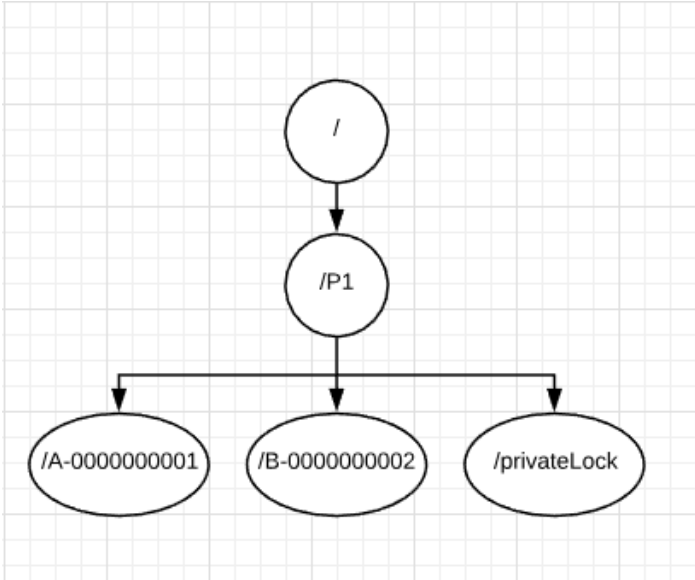
Scenario #3: Case of conflict between Foreground and Background operations

locks/nodes created by Foreground operations are prefixed by 'A-' and Foreground operations are prefixed by 'B-', this ensures that foreground task are given a higher priority during lock acquisition and all the locks/nodes are sorted based on its name in ascending order.

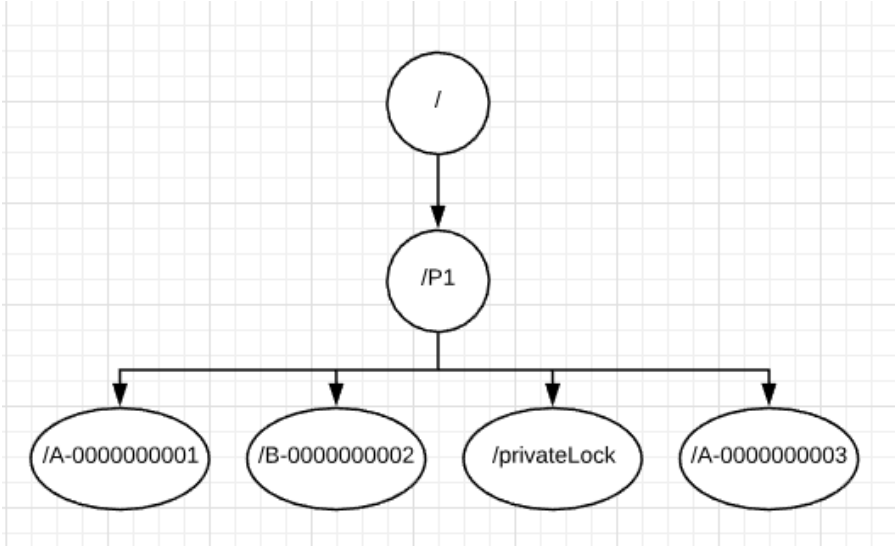
1. Lets say Client1 and Client2 wants to save same article P1 but Client1 is performing foreground operation where as Client2 is performing background operation.



2. Client acquires a lock since it has created /A-0000000001 and start processing by creating /privateLock

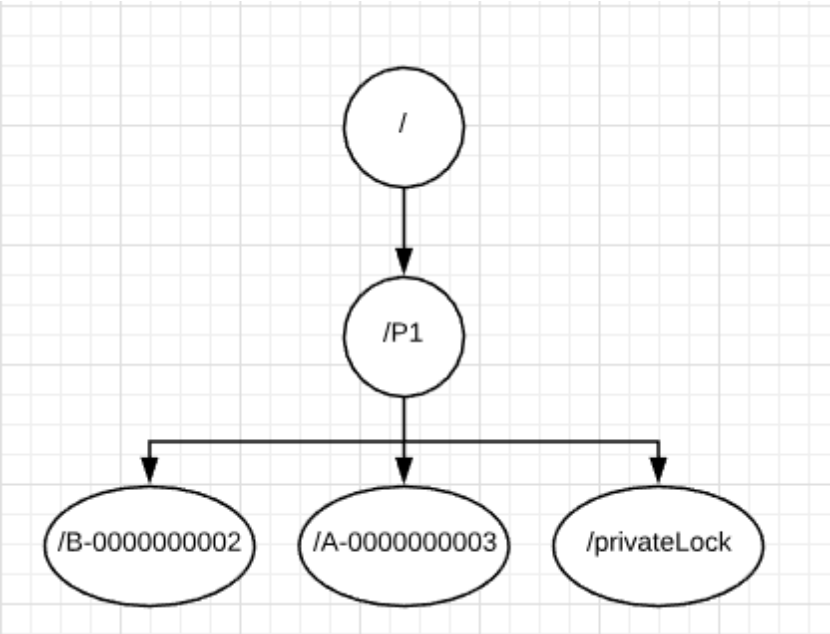


3. And before Client1 finishes its work, Client3 requests for foreground lock on P1



4. When Client1 finishes its work, Client2 and Client3 comes in race condition and evaluates locking algorithm. But this time instead of getting lock by Client2, Client3 will acquire as it has higher priority.

5. Client3 start execution by creating private lock.



6. Once Client3 finishes, Client2 proceeds to execute its background operation provided there are no foreground operations are queued.