Fault Tolerant DMTCP Coordinator

Team Members: Dixit Patel, Ashutosh Verma

Abstract

Purpose of our project is to make DMTCP fault tolerant. We have integrated zookeeper with DMTCP, so that standby masters (dmtcp_coordinator) can take over the task of failed coordinator and solve problem of single point of failure of coordinator for computation. Worker initiates leader election among znodes and reads value from master znode. This value is basically server details of a coordinator to connect to corresponding coordinator. Zookeeper watchers are configured to make sure if coordinator dies then connected workers can find another coordinator and connect to it.

Contribution

We are a team of two and have done pair programming, debugging the application together as well as finding out information about zookeeper and its API.

Introduction

Currently, DMTCP has a centralized coordinator that communicates with all worker nodes. The centralized nature makes it a single point of failure for the computation. One way to address is to use distributed coordinator processes with leader-election. These coordinators can share the current state using some external tool such as Zookeeper. Leader election is required so that a leader is chosen amongst the available dmtcp_coordinator, which can communicate to all workers.

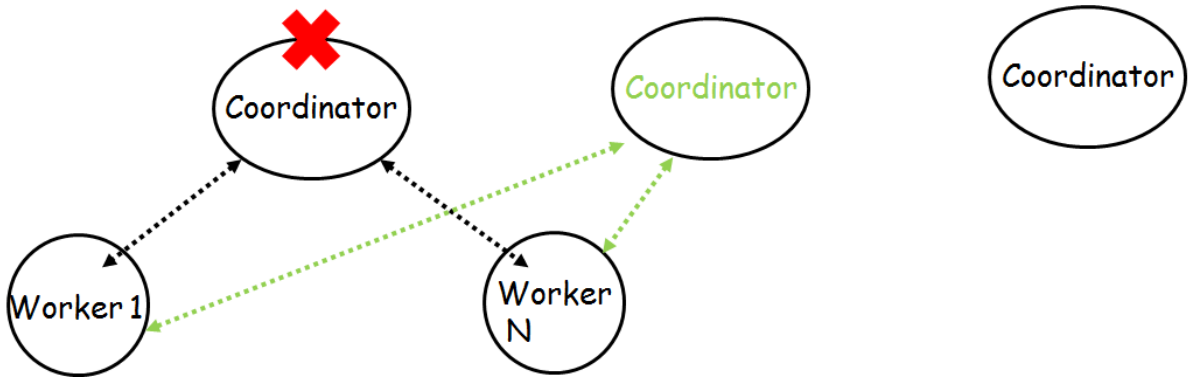
We will be working on 2 functionalities:

1. Worker is connected to coordinator. If this coordinator dies then worker should be able to connect to new coordinator
2. Worker is connected to coordinator. Coordinator initiates check-pointing and dies while check-pointing. We will be saving a value which is available across all coordinators for now to demonstrate shared state. Worker should connect to new coordinator and new coordinator should do a fresh check-pointing.

Novelty

We need to have a distributed coordinated process where we can have multiple coordinators. We need a leader election algorithm so that a master coordinator is chosen. We are building a fault-tolerant service so that if one coordinator fails then another coordinator should take up its place. We need to have a mechanism to save the state of coordinator and make it available to other coordinators. If a coordinator dies then worker

nodes should connect to new coordinator. New coordinator should read the state of old coordinator.



Design/Approach

Choice of Zookeeper

Instead of designing our own Paxos algorithm, we've chosen zookeeper as the distributed key-value pair to enable maintaining shared state among coordinators. It's the most efficient solution out there for coordination and has proved its stance in distributed systems like Elasticsearch, Hadoop, etc. There's other solutions out there like "etcd" but we choose zookeeper over all this since the API documentation and the promises looked good.

Znode

Zookeeper maintains a shared hierarchical namespace model. The namespace consists of data registers, called znodes which are similar to files and directories. Znodes maintain version numbers for data changes and timestamps to allow cache validations and coordinated updates. A client can create a znode, store up to 1MB of data and have multiple children znodes under it. We will leverage this znode to maintain dmtcp_coordinator server details and shared state which will help us build this fault tolerant system.

Watchers

Zookeeper provides us to configure watches, which are listener methods, which get notified when a znode changes its state/value. Watches are set by operations, and are triggered by ZooKeeper when anything gets changed on znode. For example, a watch can be placed on a znode. It will be triggered when the znode data changes or the znode itself gets deleted.

Watches are triggered by write operations like, create, delete and setData. We make use of watches in our application to do leader election and watch for changes on znode. If master znode is deleted then this watcher is fired and master is re-elected.

The catch with watchers is that, it's triggered only once! We need to set the watcher again once it is triggered.

Leader Election:

We have implemented leader election algorithm to select master znode which stores server details of a master dmtcp_coordinator. There's two pieces to leader election, one is sequential flag and one is ephemeral.

We create znode with ephemeral and sequential flags from dmtcp_coordinator and attach watchers on znodes.

- Ephemeral: When ephemeral znodes are created, the znodes are removed as soon as the zookeeper handler dies.
- Sequential: The sequential flag gives us a 10 digit sequential number which helps uniquely identify znodes.

We'll discuss more in the implementation section about how leader election happens in zookeeper.

Implementation

Leader election algorithm:

The function: https://github.com/dixitk13/dmtcp/blob/master/src/coordinatorapi.cpp#L649

Let "/master" be a path of choice of the application. To volunteer to be a leader:

1. Create znode z with path "master/dmtcp_coord_" with both SEQUENCE and EPHEMERAL flags.
2. Let C be the children of "/master", and i be the sequence number of z.
3. Elect the master with smallest sequence of j in "/master/dmtcp_coord_j"
4. Watch for changes on "/master/dmtcp_coord_j"

Upon receiving a notification of znode deletion:

1. Let C be the new set of children of "dmtcp_coord_"
2. Execute leader procedure in C and elect "/master/dmtcp_coord_j", where j is the smallest sequence number of all the znode in C.

When znodes are created by dmtcp_coordinator the key-value store looks like the below where the znode with the smallest sequence number is considered to be a master. So here dmtcp_coord_0000000000 is the master and the get API gets the data stored under that specific znode. We store the IP:PORT inside this znode data as shown.

```
[zk: localhost:2181(CONNECTED) 0] ls /master
[dmtcp_coord_0000000000,dmtcp_coord_0000000001,dmtcp_coord_0000000002]
```

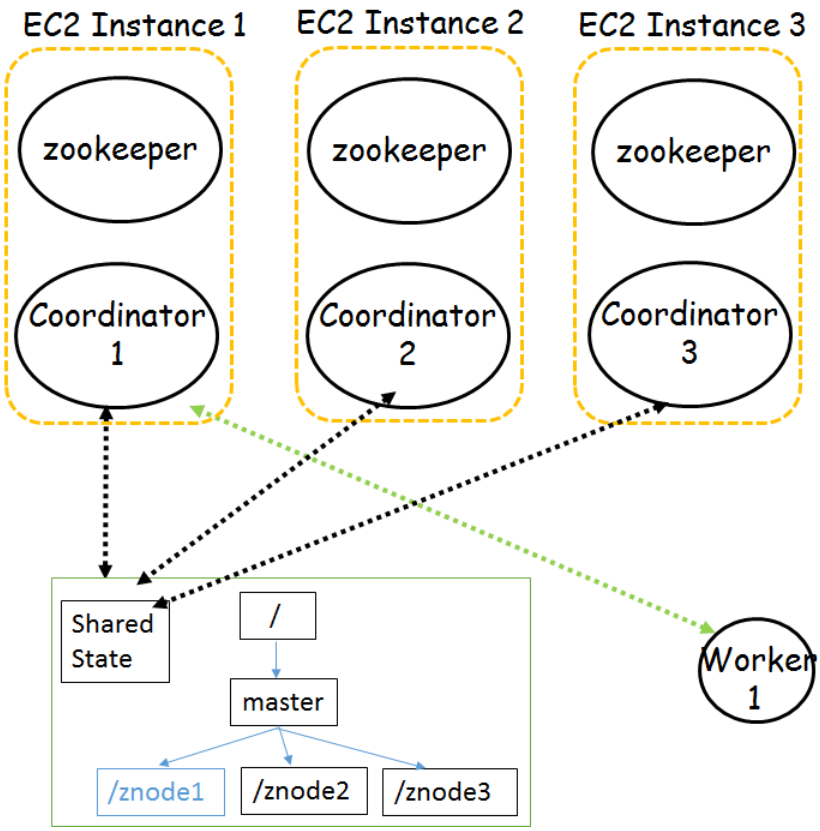
```
[zk: localhost:2181(CONNECTED) 2] get /master/dmtcp_coord_0000000001
ip-172-31-55-185:7779
cZxid = 0x1500000014
ctime = Mon Apr 25 20:59:21 UTC 2016
mZxid = 0x1500000014
mtime = Mon Apr 25 20:59:21 UTC 2016
pZxid = 0x1500000014
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x1544f3432b10003
dataLength = 21
numChildren = 0
```

The Setup

We maintain an ensemble of zookeeper servers. Each dmtcp coordinator creates respective ephemeral/sequential znodes talking to local zookeeper servers. The value under each znode is the IP address and port of the dmtcp coordinator runs.

It sets this value dynamically using the set API of zookeeper. The dmtcp launch or the worker process uses the get/getChildren API and extracts values and follows the leader election process. If the current coordinator dies, watcher process is fired on the worker process and leader election happens again, which allows to connect to another dmtcp_coordinator.

The shared state is maintained inside the "/master" znode and serves as a POC for shared state.



Watcher

For the first time when the dmtcp_launch starts it gets the children of the “/master” znode and selects a leader and attaches a watcher on this leader. Upon notification of delete from the znode, it initiates leader election and sets the watcher again to look for changes on the znode.
The watcher: https://github.com/dixitk13/dmtcp/blob/master/src/coordinatorapi.cpp#L1213

Technical Challenges

Zookeeper C binding

One of the biggest challenges we faced was the lack of documentation w.r.t zookeeper and its API to be consumed for C. Blogs were the only source of information available. In contrast to the C binding, the Java API documentation was rich, which lead you to the false assumption of zookeeper being documented in “gory detail”.

Zookeeper Session and inclusion of watches

Need to figure out different threads running in coordinator. We need to create a zookeeper session and add watchers on different znodes. As check-pointing of our worker process happens in milliseconds so we need to apply sleep() command in both coordinator and watcher. Once coordinator and worker sleeps, we will kill coordinator and watcher event will be fired. Master election is initiated and worker connects to new coordinator.

https://github.com/dixitk13/dmtcp/blob/master/src/threadlist.cpp#L391

Evaluation

The demo includes a zookeeper ensemble configured on 3 EC2 instances with dmtcp_coordinator working on each. The dmtcp_launch figures out the leader via a leader election algorithm and connects to dmtcp_coordinator corresponding to master znode's data. Worker applies watch on this particular znode. Worker connects to another dmtcp_coordinator if currently connected coordinator fails. Also zookeeper maintains shared state which is shared by all coordinators. This design can be extended to maintain the state of a coordinator. If particular coordinator fails then another coordinator can read its state and start the computation from that point where last state was saved.

We are looking at a very specific use-case which is check-pointing for DMTCP. The demo also shows that, the dmtcp_coordinator can perform check-pointing even when the leading dmtcp_coordinator dies and the worker connects to another dmtcp_coordinator. The functionality is extended to being fault tolerant in the array of dmtcp_coordinator.

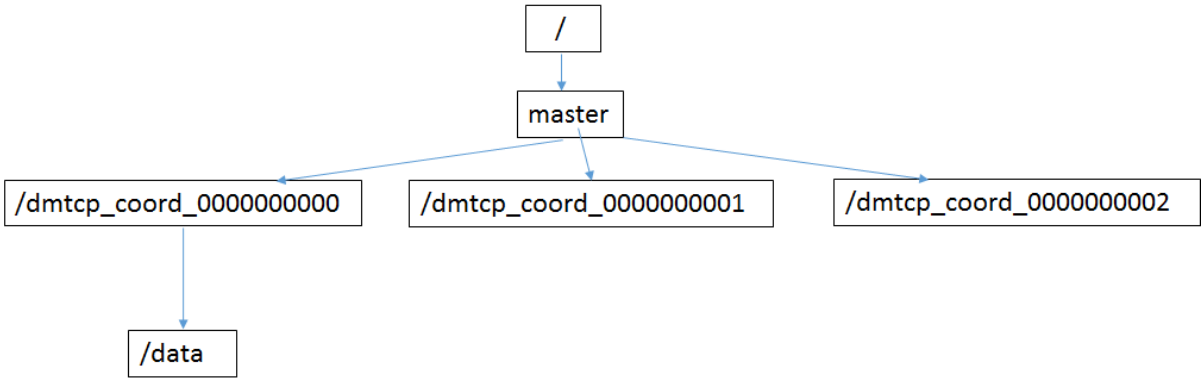
Future Scope

For future work, we need to extend this design to support following use case where if coordinator dies while check-pointing, in middle of a checkpoint then another coordinator should connect to worker and resume check-pointing.

We need to save the state of coordinator at regular intervals of time so that new coordinator can connect to worker and resume check-pointing from last saved state of old coordinator.
Additional znode data would be required under current znode for coordinator and save the state of coordinator at regular time intervals. If current coordinator dies then watcher event will be fired. Worker will connect to new coordinator and this coordinator will resume computation.

The approach is to create another znode under the coordinator as "/master/dmtcp_coord_UID/data" which contains the state of the checkpoint. This data znode will be attached with a watcher which gets fired and is returned with the watcher function.

Data to be saved includes some of the following key-value pairs like ckpt-signal, checkpoint interval, the port file and mapping of sockets for worker processes. These parameters will play a vital role for new coordinator to resume check-pointing from current saved state.



Conclusion

We have successfully integrated zookeeper in DMTCP. We are able to run multiple coordinators and create a consistent state among coordinators using checkpoint. We are able to successfully demo 2 functionalities mentioned above.