Deadlock

What is a Dead Lock? When two processes are waiting for each other directly or indirectly, it is called dead lock.

As you can see in second diagram, process 1 is waiting for process 2 and process 2 is waiting for process 3 to finish and process 3 is waiting for process 1 to finish. All these three processes would keep waiting and will never end. This is called dead lock.

Race Condition

What is a Race Condition?

When two processes are competing with each other causing data corruption.

Zookeeper - coordination

Say, there is an inbox from which we need to index emails. Indexing is a heavy process and might take a lot of time. So, you have multiple machines which are indexing the emails. Every email has an id. You can not delete any email. You can only read an email and mark it read or unread. Now how would you handle the coordination between multiple indexer processes so that every email is indexed?

If indexers were running as multiple threads of a single process, it was easier by the way of using synchronization constructs of programming language.

But since there are multiple processes running on multiple machines which need to coordinate, we need a central storage. This central storage should be safe from all concurrency related problems. This central storage is exactly the role of Zookeeper.

Intro

== Intro == So what is Zookeeper? In very simple words, it is a central store of key-value using which distributed systems can coordinate. Since it needs to be able to handle the load, Zookeeper itself runs on many machines.

Zookeeper provides a simple set of primitives and it is very easy to program to. It uses a data model like directory tree.

It is used for synchronization, locking, maintaining configuration and failover management.

It does not suffer from Race Conditions and Dead Locks.

To start Zookeeper - zookeeper-client

Lets take a look of all the nodes at the top level by typing `ls /` You can see there are some znodes. Lets see what are the children znodes under a node called brokers using the following command: `ls /brokers` Do you see three children znode ids, topics, seqid?

Now, lets see the data inside the znode brokers using `get /brokers`

You will see all the details about the znode.

== Data Model ==

The way you store data in any store is called data model. In case of zookeeper, think of data model as if it is a highly available file system with little differences.

We store data in an entity called znode. The data that we store should be in JSON format which Java script object notation.

The znode can only be updated. It does not support append operations. The read or write is atomic operation meaning either it will be full or would throw an error if failed. There is no intermediate state like half written.

znode can have children. So, znodes inside znodes make a tree like heirarchy. The top level znode is "/".

The znode "/zoo" is child of "/" which top level znode. duck is child znode of zoo. It is denoted as /zoo/duck

Though "." or ".." are invalid characters as opposed to the file system.

== Types of Znodes == There are three types of znodes or nodes: Persistent, Ephermal and Sequential.

== Types of Znodes - Persistent == Such kind of znodes remain in zookeeper untill deleted. This is the default type of znode. To create such node you can use the command: create /name_of_myznode "mydata"

== Types of Znodes - Ephermal == Ephermal node gets deleted if the session in which the node was created has disconnected. Though it is tied to client's session but it is visible to the other users.

An ephermal node can not have children not even ephermal children.

== Types of Znodes - Sequential == Quite often, we need to create sequential numbers such ids. In such situations we use sequential nodes.

Sequential znode are created with number appended to the provided name.

You can create a znode by using create -s. The following command would create a node with a zoo followed by a number: create -s /zoo v

This number keeps increasing monotonically on every node creation inside a particular node. The first sequential child node gets a suffix of 0000000000 for any node.

Create znode create /cloudxlab mydata

To see contents :- get /cloudxlab

To delete a znode cloudxlab, we can simply use rmr /cloudxlab

Create an ephermal node myeph using create -e /myeph somerandomdata

Persistent node first with a name cloudxlab using command: create /cloudxlab "mydata"

create first sequential child with name starting with x under /cloudxlab using: create -s /cloudxlab/x "somedata"

If we create another sequential node in it it would be suffixed with 1. lets take a look: create -s /cloudxlab/y "someotherdata"

It should print: Created /cloudxlab/y0000000001

Now, even if we delete previously created node using rmr command [execute rmr /cloudxlab/x0000000000]

And try to create another sequential node, the deletion would have no impact on the sequence number. Lets take a look: create -s /cloudxlab/x data

As you can see, The new number is 2.

== Architecture ==

Zookeeper can run in two modes: Standalone and Replicated.

In standalone mode, it is just running on one machine and for practical purposes we do not use stanalone mode. This is only for testing purposes as it doesn't have high availability.

In production environments and in all practcal usecases, the replicated mode is used. In replicated mode, zookeeper runs on a cluster of machine which is called ensemble. Basically, zookeeper servers are installed on all of the machines in the cluster. Each zookeeper server is informed about all of the machines in the ensemble.

As soon as the zookeeper servers on all of the machines in ensemble are turned on, the phase 1 that is leader selection phase starts. This election is based on Paxos algorithm.

The machines in ensemble vote other machine based on the ping response and freshness of data. This way a distinguished member called leader is elected. The rest of the servers are termed as followers. Once all of the followers have synchronized their state with newly elected leader, the election phase finishes.

The election does not succeed if majority is not available to vote. Majority means more than 50% machines. Out of 20 machines, majority means 11 or more machines.

If at any point the leader fails, the rest of the machine or ensemble hold an election within 200 millseconds.

If the majority of the machines aren't available at any point of time, the leader automatically steps down.

The second phase is called Atomic Broadcast. Any request from user for writing, modification or deletion of data is redirected to leader by followers. So, there is always a single machine on which modifications are being accepted. The request to read data such as ls or get is catered by all of the machines.

Once leader has accepted a change from user, leader broadcasts the update to the followers - the other machines. [Check: This broadcasts and synchronization might take time and hence for some time some of the followers might be providing a little older data. That is why zookeeper provides eventual consistency no strict consistency.]

When majority have saved or persisted the change to disk, the leader commits the update and the client or users is sent a confirmation.

The protocol for achieving consensus is atomic similar to two phase commits. Also, to ensure the durability of change, the machines write to the disk before memory.

If you have three nodes A, B, C with A as Leader. And A dies. Will someone become leader?

Either B or C will become the leader.

If you have three nodes A, B, C with C being the leader. And A and B die. Will C remain Leader?

C will step down. No one will be the Leader because majority is not available.

As we discussed that if 50% or less machines are available, there will be no leader and hence the zookeeper will be readonly. Don't you think zookeeper is wasting so many resources?

The question is why does zookeeper need majority for election?

Say, we have an ensemble spread over two data sources. Three machines A B C in one data center 1 and other three D E F in another data center 2. Say, A is the leader of the ensemble.

And say,The network between data centres got disconnected while the internal network of each of the centers is still intact.

If we did not need majority for electing Leader, what will happen?

Each data center will have their own leader and there will be two independent nodes accepting modifications from the users. This would lead to irreconcilable changes and hence inconsitency. This is why we need majority for election in paxos algorithm.

== Sessions ==

Lets try to understand how do the zookeeper decides to delete ephermals nodes and takes care of session management.

A client has list of servers in the ensemble. The client enumerates over the list and tries to connect to each until it is successful. Server creates a new session for the client. A session has a timeout period - decided by the client. If the server hasn't received a request within the timeout period, it may expire the session. On session expire, ephermal nodes are deleted. To keep sessions alive client sends pings also known as heartbeats. The client library takes care of heartbeats and session management.

The session remains valid even on switching to another server.

Though the failover is handled automatically by the client library, application can not remain agnostic of server reconnections because the operation might fail during switching to another server.

== Use case - 1 ==

Let us say there are many servers which can respond to your request and there are many clients which might want the service. From time to time some of the servers will keep going down. How can all of the clients can keep track of the available servers?

It is very easy using ZooKeeper as a central agency. Each server will create their own ephermal znode under a particular znode say "/servers". The clients would simply query zookeeper for the most recent list of servers.

Lets take a case of two servers and a client. The two server duck and cow created their ephermal nodes under "/servers" znode. The client would simply discover the alive servers cow and duck using command ls /servers.

Say, a server called "duck" is down, the ephermal node will disappear from /servers znode and hence next time the client comes and queries it would only get "cow".

So, the coordinations has been made heavily simplified and made efficient because of ZooKeeper.

== Guarantees ==

What kind of guarantees does ZooKeeper provide?

Sequential consistency: Updates from any particular client are applied in the order Atomicity: Updates either succeed or fail. Single system image: A client will see the same view of the system, The new server will not accept the connection until it has caught up. Durability: Once an update has succeeded, it will persist and will not be undone. Timeliness: Rather than allowing a client to see very stale data, a server would prefer shut down.

== Ops ==

ZooKeeper provides the following operations. We have already gone through some of these.

create Creates a znode (parent znode must exist) delete Deletes a znode (mustn't have children) exists/ls Tests whether a znode exists & gets metadata getACL, setACL Gets/sets the ACL for a znode getChildren/ls Gets a list of the children of a znode getData/get, setData Gets/sets the data associated with a znode sync Synchronizes a client's view of a znode with ZooKeeper

== Multiupdate == ZooKeeper provides functionality of multiupdate. It batches multiple operations together. Either all fail or succeed in entirety.Others never observe any inconsistent state. It is possible to implement transactions using multiupdate.

== API ==

You can use the ZooKeeper from within your application via APIs - application programming interface.

Though ZooKeeper provides the core APIs in Java and C, there are contributed libraries in Perl, Python, REST. == sync == For each function of APIs, synchronous and asynchronous both variants are available.

While using synchronous APIs the caller or client will wait till ZooKeeper finishes an operation. But if you are using asynchronous API, the client provides a handle to the function that would be called once zooKeeper finishes the operation.

== Watches ==

Similar to triggers in databases, ZooKeeper provides watches. The objective of watches is to get notified when znode changes in some way. Watchers are triggered only once. If you want recurring notifications, you will have re-register the watcher.

The read operations such as exists, getChildren, getData may create watches. Watches are triggered by write operations: create, delete, setData. Access control operations do not participate in watches.

WATCH OF exists is TRIGGERED WHEN ZNODE IS created, deleted, or its data updated. WATCH OF getData is TRIGGERED WHEN ZNODE IS deleted or has its data updated.

WATCH OF getChildren is TRIGGERED WHEN ZNODE IS deleted, or its any of the child is created or deleted

== ACL ==

ACL - Access Control Lists - determine who can perform which operations.

ACL is a combination of authentication scheme, an identity for that scheme, and a set of permissions.

ZooKeeper supports following authentication schemes: digest - The client is authenticated by a username & password. sasl - The client is authenticated using Kerberos. ip - The client is authenticated by its IP address.

== Use case ==

Though there are many usecases of ZooKeeper. The most common ones are: **Building a reliable configuration service A Distributed Lock Service Only single process may hold the lock**

== Not to Use === It is important to know when not to use zookeeper. You should not use it to store big data because the number of copies == number of nodes. All data is loaded in ram too. Also, there is a Network load for transfer all data to all nodes.

Use ZooKeeper when you require extremely strong consistency