Redesign LIR implementation

Note: Before reading further you should know that, this document mentioned some problems of current LIR implementation and proposed a totally different approach (which I think is safer, more robust and correct) to SOLR-5495. You may question why don't we fix the current LIR implementation because implementing a totally new approach will cost us a lot of time and effort.
First of all, I and Shalin tried to do that (mostly based on the idea of SOLR-9555) but failed, there are many unsolved questions/problems on that design (this is described in the last part of this document). Therefore I kinda feel that fixing current problems by a bandage will then see other problems soon.
Secondly, I think this approach is not hard to implement correctly, so most of the effort will be put on writing tests.

Problem

LIR stands for Leader Initiated Recovery. This means that when the leader and replicas are network partitioned but still be able to connect to ZK. Leader must have a way to tell replicas that “hey, you are out-of-sync so step back and do recovery”.

Problem in current implementation of LIR (SOLR-5495)

Failed uses case 1 (SOLR-9555)

1. A replica goes down and then restart
2. Leader start LIR thread for that replica (after failed on sending a document to that replica, ex 503 ‘not ready yet’)
3. The replica publish itself as RECOVERING
4. Leader publish the replica as DOWN
5. Leader waiting for replica to finish recovering
6. The replica wait for the leader to see it as RECOVERING which will take forever

Failed uses case 2 (Varun)

1. Leader start LIR Thread, publish lir state of replica as DOWN.
2. s
3. A new update from leader, mark lir state of replica as DOWN
4. Replica success in recovering but failed to publish as active because lir state = DOWN.

The current design has several problem

- Both leader and replica can update to state of a replica
- Both leader and replica can update to lir state of a replica
- For a replica we will have a combination of 9 different lir state + state, this is very complex

Idea

Instead of using lir node, I propose a different scheme to solve the case “when leader and replica can talk to ZK but can not talk to each other”.

Each replica will have a special positive number called *term*. The idea of *term* is only replicas (in the same shard) with highest term are considered healthy. Other replicas with lower term are considered out of sync.

The reason why *term* is safer than lir state is term can only be changed by using **two operations**

- Op1 : Set term equal to leader term (only replica can do this)
- Op2 : Increase term of leader and some other replicas equals to leader (only leader can do this)

Term rules

- Term can only be set to a larger number

Election rules

- Only replicas with highest term can win the election (or term of leader must be greater or equal to all other replica’s terms)
- Only replicas with state DOWN or ACTIVE can win the election

Leader rules

- Leader only send updates to replicas that
    - Replica’s term is equals to leader’s term
    - Replica’s state is RECOVERING or ACTIVE
    - Replica’s node is live
- When leader failed to send an update to a replica (A) ,
    - Leader will increase its term and all other replicas who success in receiving the update. To prevent this operation being stacked, if terms of all alive replicas already larger than replica A, skipping this operation.
    - Leader will make no guarantee about the result of this update. It may succeed, it may have failed. In this case client must resend this update.
- Leader will make sure that any replicas are skipped will have term less than its before return to users.

Replica rules

- Whenever a replica find its term less than leader’s term, cancel current recovery and redo it.
- Replica will have an internal/persisted flag indicate was it in the middle of an unfinished recovery process or not

Q&A - implementation detail

Q: Where terms are stored
A: All replica's terms of a same shard will be stored in a same ZK node.

Q: How to prevent racing in saving new value to term ZK node
A: Replica will do atomic updates by on ZK node version, so if the ZK node version is match with its local version. It will overwrite its local term node with ZK's value.

Q: How leader update the term node
A: To prevent pollute ZK, the leader keep a local copy of term node, it won't touch ZK if the local term node tell the leader that replica who failed to receive update already have a term less than leader term.

Q: When leader update its local term node
A: Leader only needs to update its local term node when receive a PrepRecoveryCmd from a replica, no need to watch ZK term node.

Q: When replicas update its local term node
A: Replica updates its local term node by watching ZK term node. Whenever it find that its term node is less than leader's term node, start recovery process. (Leader does not have to do this)

Q: What happen when a leader get elected but get down right after that.
A: Because the election do not touch the term, so it will behave normally as today.

Q: What happen when leader and replica (R) are network partitioned but both still can connect to ZK
A: Here are the case

1. Leader send an update to a replica, the update get failed when sent to R.
2. Leader increase its term and other replicas (except R) to t+1
3. For updates after this point, Leader will skip sending updates to R (because R's term is less than leader's term)
4. *New step* : R sends a ping to the Leader and waiting for result (the ping request can have a timeout of 2 seconds and keep retry after that)
5. R sees that its term is smaller than Leader's term, start recovery process
6. Replica switch its state to RECOVERING
7. *New step* : R sets its term to leader's term
8. R send PrepRecoveryCmd to the Leader and waiting for response
9. Leader starts sending updates to A, if it get failed back to 1.
10. If the network is healed. The recovery process will be success.

Note: To prevent the looping in step 9 which causes too many updates sent to Overseer node. I introduced a new step, step #4.

Q: What happen when multiple updates get failed at a same time (on the same replica)
A: We will use a locking mechanism so only one thread can touch Zk. Other threads will see term node already get touched hence do nothing.

Q: What get changed in client side.
A: In a strict way, the client needs to watch term node (along with watching clusterstate), so it won't query out-of-sync replica. But we can skip doing this action because replica we be in RECOVERING state right after (in small delay time) term node get updated.

Q: Is replica's term enough for ensuring that a replica is out-of-sync or not. Or can a replica become leader if its term is largest?
A: No this is not enough, replica will set its term equals to leader when start recovery process. Until finish recovery, the replica still in out-of-sync.

Compare to current implementation

Current implementation	Term node implementation
Both state (lir state and replica's state) get updated by leader and replica.	Only replica touch its state. Both leader and replica update term node but in a safer - stricter way.
Leader publishes state of the replica and sends an explicit recovery request to the replica	Leader will no longer publish state of the replica. It will no longer send an explicit recovery request to the replica.
Leader puts replica into DOWN state. So client will stop querying the replica right after that.	Leader increases term node. Replica gets notified by the change then switch to RECOVERING. So the amount of time client keeps query the out-of-sync replica will be higher.
Many touch to ZK and Overseer (6 ops) ● Change LIR node to DOWN, RECOVERING and ACTIVE ● Change replica's state to DOWN, RECOVERING and ACTIVE	Less touch to ZK (4 ops) ● Leader increases term node ● Replica updates term node ● Change replica's state to RECOVERING and ACTIVE.

Proof

First of all, we assume that recovery process is good enough so any updates come before recovery process will be recovered after the recovery process finish.

Proof #1

Assume that a replica

- becomes active
- its term equals to leader's term
- missed an update u1

We will prove that this scenario can not happen.

Let's call **E1** is the event when replica published to RECOVERING and the term of the replica is **t1.**

First of all, u1 must come after **E1**. If not the recovery process of the replica won't miss update u1. Secondly, u1 must be failed before **E1**. If not term will be increased to **t1+1**. According to the rules above this will trigger another recovery process. So we won't miss u1.

This is a contradiction.

Proof #2

Assume that a replica A becomes leader, but it missed an update u1 but we tell to client that u1 is updated success.

We will prove that this scenario can not happen.

First of all, The only scenario where replica A does not have update u1, but u1 is a success update is Leader did not send update u1 to replica A.

There are 2 reason that Leader did not send update u1 to replica A

- Replica A's state is DOWN
- Replica A's term is less than leader's term

This is a contradiction because replica with above properties can not become leader.

==========

An another approach and its problems

Idea

1. We must avoid publishing state of a replica from other (leader) nodes to cut down on the race conditions
2. Leader and replica will update LIR node
3. Leader won't send updates to replica if replica's LIR state = DOWN
4. Leader will no longer publish state of the replica. It will no longer send an explicit recovery request to the replica.

5. The replica watches the LIR node and starts recovery process when the LIR node is set to recovery state.
6. To avoid too many writes to ZK from the leader, the leader batches such writes for a time interval, say 2 seconds. So, for each 2 second interval, the leader will write to ZK only once regardless of how many updates are missed by the replica.
7. The LIR node also contains the version of the update that was missed.
8. When the LIR node is updated, the replica watching the node cancels and restarts recovery if one was already in progress.

Q&A

Q: Should we let only leader update LIR node? Allow both leader and replica update LIR node will lead us to another race condition.

A: We really want only leader update LIR node but

1. In case of leader and replica are network partitioned. How can leader know that it should stop sending updates to that replica (we no longer put replica into DOWN in this approach)?
2. If the replica restarts right after the lir node get touched. How can replica know that it must do recovery on startup?

Q: Can we prove that, a replica with below properties can not exist?

- becomes active
- its term equals to leader's term
- missed an update u1

A: I don't think we can prove that

Q: Does this approach solve Varun's use case above?

A: I'm afraid that we did not solve that problem with this idea.