Reconfiguration of the Libra Blockchain by Self-Reflecting Transactions

The LibraBFT Team

Abstract

This report describes reconfiguration of LibraBFT - an extension to LBFT to support reconfigure itself by embedding configuration-change transactions in the sequence. LBFT addresses the fundamental consensus problem of the libra blockchain with fixed configuration setup, think of validator set, consensus keys, VM features, software version etc. Those are all like configuration of one LBFT protocol instance. we want to support those configuration changes without human intervention which could be excessive work and error prone. Instead we present a protocol to handle reconfiguration just like a normal transaction.

1 Introduction

Reconfiguration of consensus protocol is not a new topic, it has been studies for decades and has more and more use cases in recent blockchain area.

We'll present our mechanism to support reconfiguration of LBFT in section 2, and discuss the approches we considered and comparison between them in section 3, finally we'll give the correctness argument for all approaches in section 4.

At the high level, reconfiguration is under the authority of the current set of validators and operates via reconfig-transactions embedded inside the normal chain of transactions. The chain continues beyond the reconfiguration transaction just like normal transactions, and the node should use the on-chain configuration to spawn LBFT instance. Genesis transaction is the first reconfiguration transaction and defines the configuration for the first LBFT instance.

Epoch is the counter for the on-chain configurations, it's bumped after reconfiguration transactions. LBFT instance is constructed with configuration including epoch, ledger state, validator set etc. and initializes its internal state such as setting round to 0, generating genesis block for block store etc. Blocks in different epochs are not chained and are internal to each LBFT instance, but the underlying ledger state(chain of transactions) are continued.

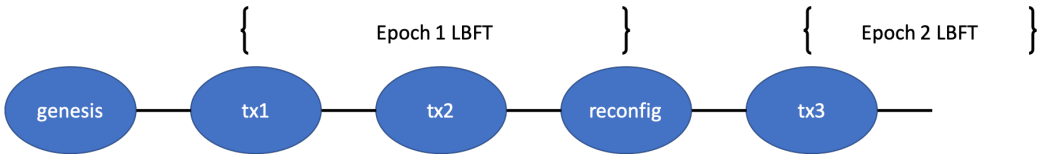

Figure 1: reconfiguraiton with multi LBFT instances

2 Reconfiguration Mechanism

In this section we describe the reconfiguration protocol and the changes required to LBFT protocol. One of the design goal is to decouple these two protocols as much as possible to provide nice abstractions.

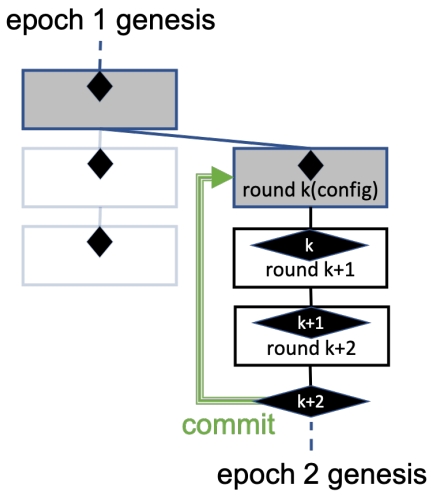

Figure 2: Commit reconfig transaction

2.1 Reconfiguration Protocol

Epoch is a monotonous increasing number used to refer to a LBFT instance running with a specific configuration. LBFT instance is agnostic to epoch numbers, so we introduce EpochManager to process epoch related information.

2.1.1 Message types

First we add *epoch* to every message types of LBFT protocol including proposals, votes, timeout msg etc. Messages with the same *epoch* as local LBFT protocol go to LBFT for processing. Messages that have different *epoch* go to `EpochManager.process_different_epoch_messages`.

Additionally we introduce two messages types `EpochChange` and `EpochRetrieval`:

```
// The commit proof signed by LBFT
LedgerInfo
    state_id ;
    signatures ;
    ...
    next_configuration: Option < Configuration > ; // Added field to support reconfiguration
// Signals an epoch change from i to j
EpochChange
    start_epoch ; // The epoch of the first ledger info
    end_epoch ; // The epoch after the last ledger info
    proof: Vec < LedgerInfo > ;
// Rquest an EpochChange from start_epoch
EpochRetrieval
    start_epoch ;
```

2.1.2 EpochManager

We have three different handlers for EpochManager to process those messages.

Handling different epoch, `process_different_epoch_msg`: When the different LBFT epoch messages comes, we'll compare our local *epoch* with the messages. If we're behind, we'll issue a `EpochRetrieval` to the peer who sent the message. If we're ahead, we'll provide a `EpochChange` to the peer.

Handling epoch retrieval, `process_epoch_retrieval`: It's similar to how to process messages from lower *epoch*, we'll provide the `EpochChange` to help peer join our *epoch*.

Handling epoch change, `start_new_epoch`: When we receive an `EpochChange`, first we verify we're at *epoch* i and the `EpochChange` is correct by iterating through the ledger info and verify the signatures are corresponding to the configuration, then we ensure we're ready for the configuration of *epoch* j e.g. sync to the ledger state at the beginning of the *epoch* j. Finally we spawn a new LBFT instance with the configuration of *epoch* j.

2.2 Changes to LBFT

Although LBFT is agnostic to *epoch*, it has to support additional rules for **reconfiguration protocol** in order to maintain the safety property across *epoch*.

2.2.1 Commit Flow

Just a kind reminder that LBFT has three-phase commit rule - once we collects a QC for three contiguous rounds k, k+1, k+2, we commit the the block at round k and all its prefix.

To support `EpochChange` mentioned above, we need to add an optional field `next_configuration` to the `ExecutedState` which is only outputed when executing a reconfiguration transaction. Remember that `LedgerInfo` carries the `ExecutedState`, so that it contains such field and can serve as end-peoch proof.

Besides the normal commit flow in LBFT which persists transactions in ledger state, we have an additional step to broadcast `EpochChange` to all the current validators if the `LedgerInfo` carries next_configuration. EpochManager would then kick in and handle the `EpochChange` message and spawn new LBFT instance with next_configuration and shutdown the current one.

2.2.2 Descendants of Reconfig Block - Safety

In this section, we explain the need to keep all the descendants of a reconfig-block empty of transactions until it's committed otherwise we might have safety violation e.g. double spend.

The fundamental problem here is that LBFT spreads the phase of the protocol(for every proposal) over 3 rounds i.e. pipelining. When it comes to configuration, the proposal after reconfiguration transaction is under a different configuration with its parent and the pipelining may not work anymore, we'll discuss more about this in section 4.

We use validators configuration as example, demonstrated in Figure 3. Imagine we're at *epoch* 1 with $\texttt{validators}\{\texttt{a1}, \texttt{a2}, \texttt{a3}, \texttt{a4}\}$, and there's a reconfiguration transaction changes us to $\texttt{validators}\{\texttt{c1}, \texttt{c2}, \texttt{c3}, \texttt{c4}\}$. Reconfiguration transaction is included in B1, and a4 collects QC for B3 which has a ledger info(commit proof) of B1 but only unveils to $\texttt{c1}, \texttt{c2}, \texttt{c3}$ but not $\texttt{a1}, \texttt{a2}, \texttt{a3}$. $\texttt{c1}, \texttt{c2}, \texttt{c3}$ will start the new epoch and use B1 as its genesis while $\texttt{a1}, \texttt{a2}, \texttt{a3}$ is going through another three rounds B4, B5, B6 and finally commit B4(which has B1 as its ancestor) and unveil it to `c4`. Now `c4` thinks B4 is the genesis. If we have a double spend txn1, txn2 that txn1 committed with B4 and txn2 committed in B10 in new *epoch* 2 with $\texttt{c1}, \texttt{c2}, \texttt{c3}$, clients who query $\texttt{c1}, \texttt{c2}, \texttt{c3}$ would see the double spend compared with `c4`.

Having any descendants of reconfiguration empty solves this problem, semantically it's equivalent to run 3 phases without pipelining.

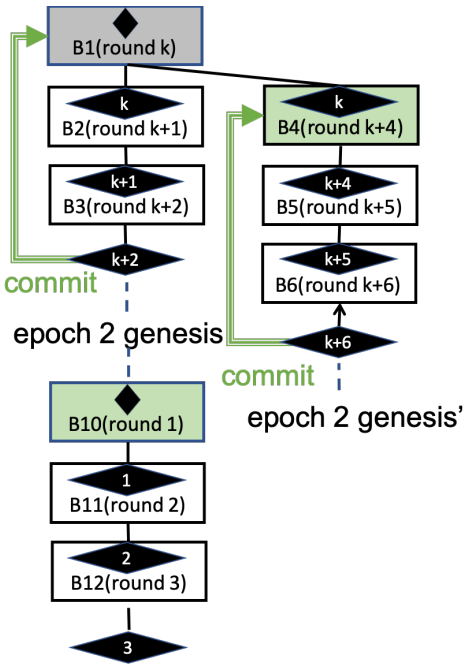

Figure 3: Double spend in B4 and B10(green box)

2.2.3 Epoch Boundary - Liveness

In this section, we explain how to specify the boundary of each *epoch* and the risk of losing liveness if we don't do it correctly.

Start In LBFT we assume everyone starts from a known genesis which we can consider as part of the configuration otherwise we'll lose liveness even we preserve safety with empty blocks. Intuitively we could use the block that includes reconfiguration transaction, but we may not commit that block directly but instead one of its descendants. And even worse, we may have multiple commit of its descendants since byzantine nodes could hide commits as Figure 4 demonstrates.

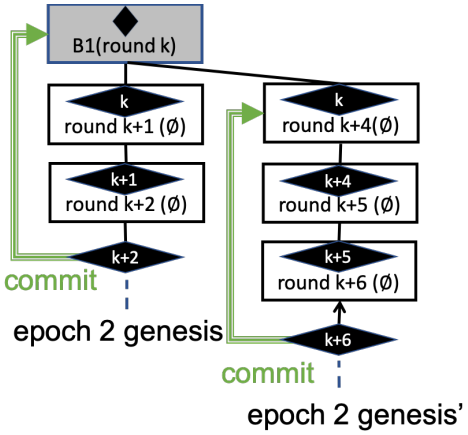

Figure 4: If genesis is not equal to genesis', we lose liveness

With the empty suffix blocks change, we ensure the same ledger state when we commit reconfiguration transaction regardless of which decendants, and we leverage that to derive a genesis block so that every ledger info that ends previous *epoch* would result in the same genesis deterministically.

End After a node sees `LedgerInfo` that has `next_reconfiguration`, it'll stop participating the current LBFT instance, and spawning next LBFT instance. If it's not part of the next reconfiguration, counter-intuitively it can not shutdown itself immediately otherwise we'll lose liveness. As an example, if we have `validators`$\{$`a1`, `a2`, `a3`, `a4`$\}$ at *epoch* i, and malicious `a4` collects a `LedgerInfo` and sends to `a1` that's not part of *epoch* $i+1$ and `a1` decides to shutdown itself. Byzantine node `a4` also stops participating then `a2`, `a3` would suffer a liveness problem and are never able to make progress. To address such problem, `a1` needs to keep the `EpochManager` running until it sees a QuorumCert from new *epoch* so that it knows at least f+1 honest nodes in the new *epoch* bootstrap.

3 Alternatives

In this section we briefly discuss some other approaches we considered for the problem and the trade offs.

3.1 Quorum Compatible

As we mentioned in 2.2.2, we give up the pipeline of LBFT to preserve safety. Here we consider an alternative that preserves both safety and pipeline.

Condition Consider a reconfiguration transaction that changes from validators `N1` to validators `N2` and $f1 = \lceil |N1|/3 \rceil - 1$, $f2 = \lceil |N2|/3 \rceil - 1$ malicious power, $|N1| - f1$, $|N2| - f2$ as quorum size respectively. We only allow reconfiguration if

$$|N1 \cap N2| > f1 + f2 + max(f1, f2) \tag{1}$$

Analysis We have an important lemma in LBFT for safety: Under BFT assumption, for every two quorums of nodes, there exists an honest node that belongs to both quorums.

With this approach, we restrict the changes between two configurations and we can prove quorums are interchangeable in both configurations, so that we implicitly maintain the safety property.

3.2 Pipelined Configuration

This is one step further on top of the quorum compatible approach. Instead of implicit compatible quorums, we require explicit two quorums to commit the reconfiguration transaction. We preserve the same safety lemma as above but doing it in a more explicit way.

One way to think about this is pipelining the reconfiguration like we pipeline blocks, reconfiguration $A \rightarrow B$ takes effect (the next blocks on this branch need B's quorum) right after the block and until a commit of another reconfiguration $B \rightarrow X$.

3.3 Comparison

We assemble a comparison table for the three approaches we describe.

	Pipelining	Implementation Complexity	Flexibility
Basic	No	Small	Most
Quorum Compatible	Yes	Medium	Least
Pipelined Configuration	Yes	Big	Medium

Basic mechanim gives up pipelining/performance while provides more flexible configuration changes like protocol change, it also provides better isolation between different configurations.

Quorum Compatible enables us with pipelining and is almost seamless to LBFT by strictly limit how configuration can change.

Pipelined configuration provides pipelining the same as Quorum Compatible with more flexible reconfiguration by supporting explicit multi quorums.

Also regardless of implicit or explicit quorums, to support pipelining configuration, we need to track configuraiton per block tree branch which adds complexity compared to one configuration per whole block tree in basic mechanism.

We chose the approach considered all above and think the performance gain of pipelining can not justify the added complexity and lost flexibility for now, but could leave for future work.

4 Correctness

It is useful to think about correctness in the following manner. Each round in the chain represents a slot in a sequence of consensus decisions. Once a decision is committed, it takes effect immediately in the next slot. For normal transactions, this means that the state of the ledger is updated by each slot, starting with the state of the previous slot. For control actions like reconfiguration, it changes the algorithm itself. This again, takes effect in the immediate slot following the control transaction. Generally, note that reconfiguration transactions can change any aspect of the algorithm, not just the validator set.

Now, imagine that an entire consensus algorithm is done separately for each round/slot. Clearly, to agree on the commit output of the algorithm we must have agreement on the algorithm itself. Initially, the system is bootstrapped with an algorithm (and in particular, the validator-configuration) responsible for slot 1 pre-determined and known to all via genesis transaction. Thereafter, if a slot commits an algorithm (configuration) change, it takes effect in the succeeding slot.

It is fairly obvious to see inductively that in this manner, we maintain agreement on the algorithm for each slot.

Enters pipelining As depicted above under "Pipelined reconfiguration", the LibraBFT consensus algorithm "spreads" the phases of the protocol (for every slot) over 3 rounds. More specifically, every phase is carried in a single round and contains a new proposal. For example, the leader of round k drives only a single phase of certification of its proposal. In the next round, k+1, a leader again drives a single phase of certification. Interestingly, this phase has multiple purposes:

- The k+1 leader sends its own k+1 proposal.
- It also piggybacks the QC for the k proposal. In this way, certifying at round k+1 generates a QC for k+1, but also a QC-of-QC for k.
- In the third round, k+2, the k proposal can become committed, the k+1 proposal can obtain a QC-of-QC, and the k+2 can obtain a QC.

Importantly, it should be understood that spreading phases of the k-protocol into rounds k+1 and k+2 does not shift the responsibility away from the k-algorithm.

To complicate matters, if any phase aborts due to a timeout, the k-protocol remains undecided. It can become committed only via a transaction that extends the k branch. If the k command is a reconfiguration command, the reconfiguration takes effect only upon the next commit.

For example, say that round k+1 aborts (no QC(k+1) obtained). The leader for k+2 extends k, and if three consecutive rounds (k+2, k+3, k+4) complete, then the prefix of the branch up to and include k+2 becomes committed, including k. Importantly, in this case, the reconfiguration transaction takes effect at round k+2.

A correctness "meta argument" reduction is the follows:

If the algorithm for a slot k is A, then it is necessary and sufficient for the quorums used for three slots succeeding the first commit on a branch extending A to contain quorums of A. If there is no gap, then rounds k, k+1 and k+2 use algorithm A. If there is a gap and k' is the next commit, then rounds from k all the way to k', k'+1, k'+2 use algorithm A.

If round k* (once committed) changes the configuration to A', then similarly, it is necessary and sufficient for slots succeeding k* to contain quorums for A' but also A until k* becomes committed.

Reconfiguration condition with pipelining Consider a round-k proposal such that:

- the last committed reconfiguration transaction on the branch which the k proposal extends is A (hence, A is "in effect")
- the k proposal changes (once committed) the algorithm from A to A'
- k becomes committed by a round k* transaction which extends the k-branch

Then the following is necessary and sufficient for reconfiguration to not fork:

- A' should take effect at round k+1
- rounds k+1 thru k*+2 must use QC's that contains quorums of both A and A'.

This "meta argument" suffices to show that the above proposals all work. The proposals differ in other properties they provide, such as ease of implementation, building state-transfer transition into reconfiguration, etc.