State Synchronization and Verification of Committed Information in a System with Reconfigurations

Abstract

In this document we describe the current approach for state synchronization and verification of committed transactions in case the active set of validators and their signatures can change periodically.

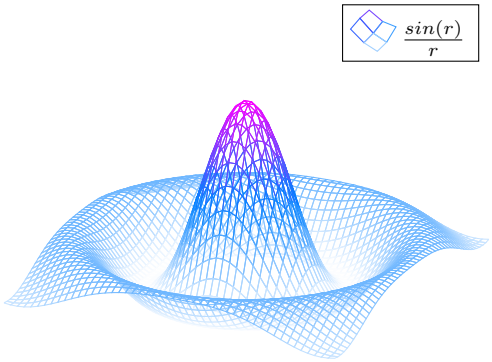
```rust
use async_stream::stream;

use futures_util::pin_mut;
use futures_util::stream::StreamExt;

#[tokio::main]
async fn main() {
    let s = stream! {
        for i in 0..3 {
            yield i;
        }
    };

    pin_mut!(s); // needed for iteration

    while let Some(value) = s.next().await {
        println!("got {}", value);
    }
}
```

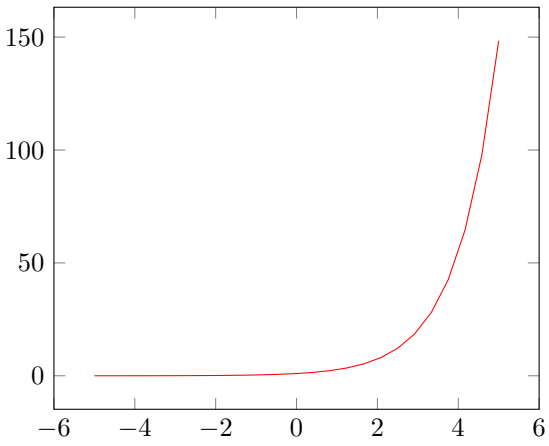
Listing 1: Rust example

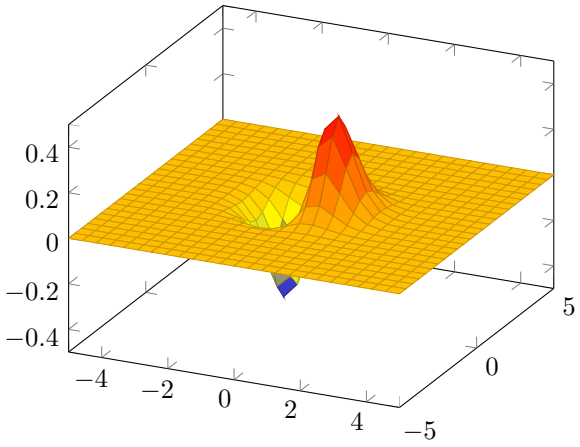
$$\begin{aligned} A &= \frac{\pi r^2}{2} \\ &= \frac{1}{2}\pi r^2 \end{aligned} \tag{1}$$

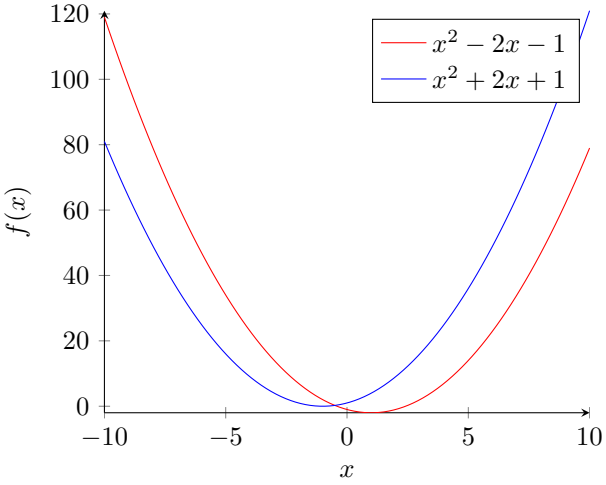
The beautiful equation 1 is known as above.

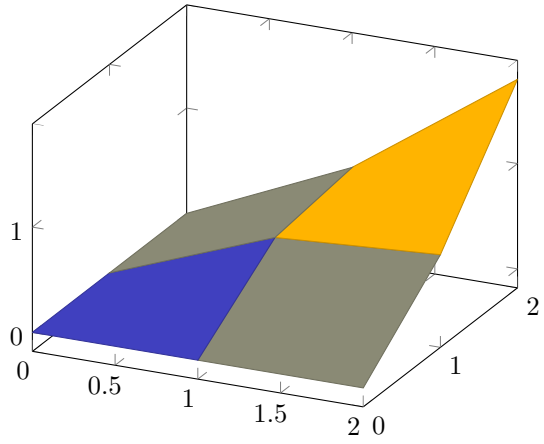
Example using the mesh parameter

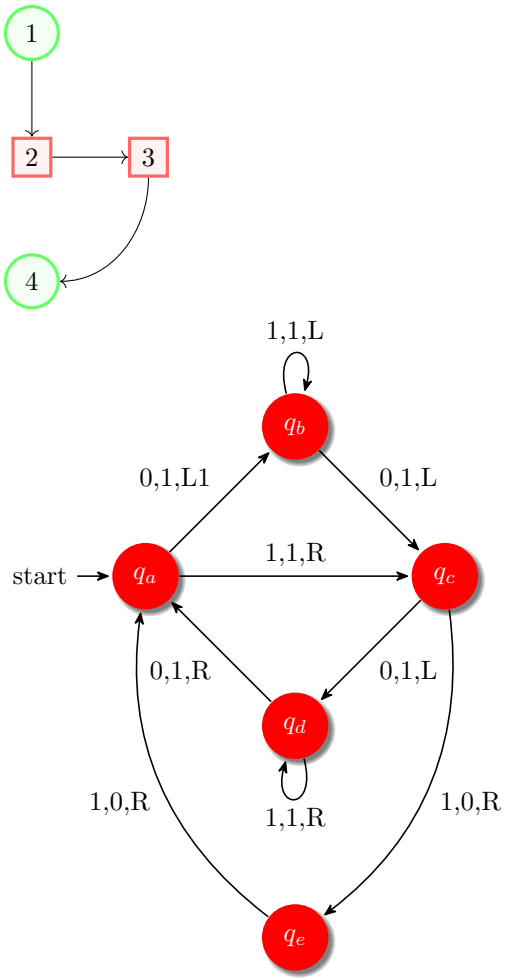



150
100
50
0
−6
−4
−2
0
2
4
6

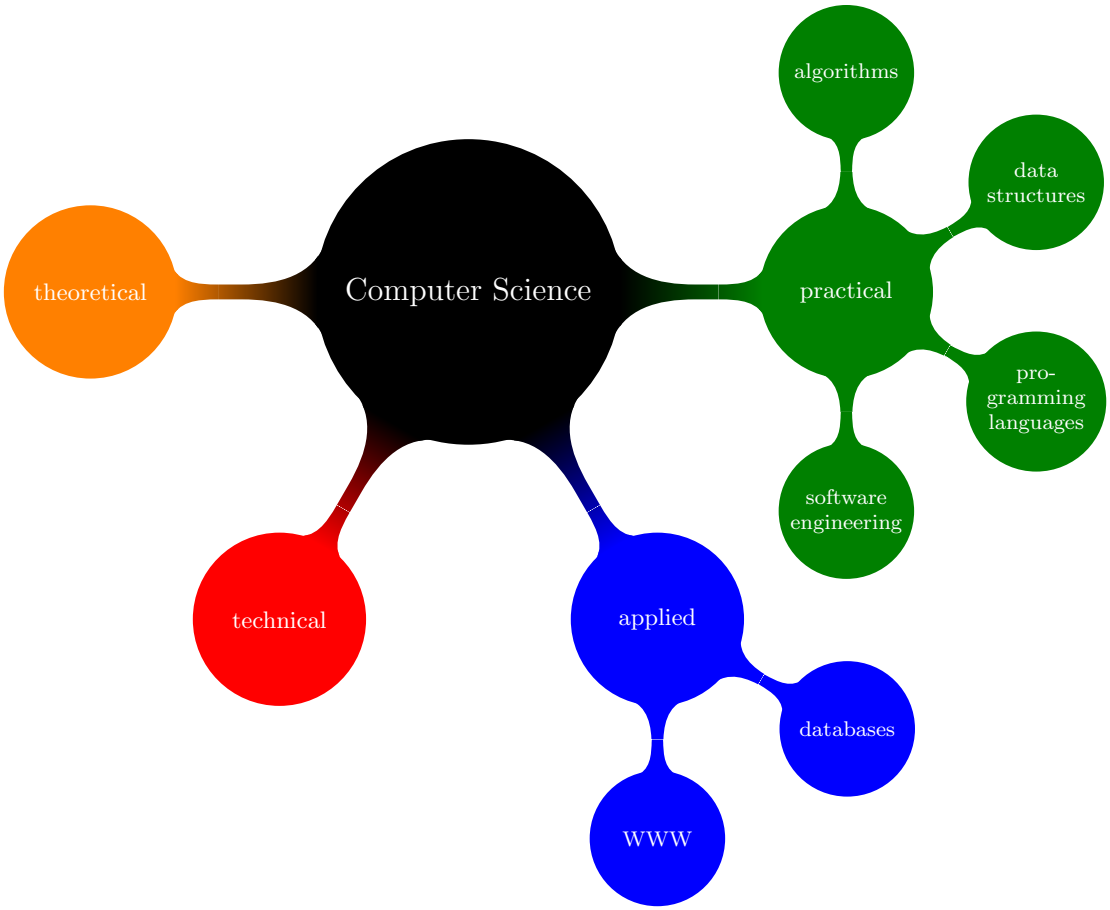

0.4
0.2
0
−0.2
−0.4
−4
−2
0
2
4
−5
0
5


$x^2 - 2x - 1$
$x^2 + 2x + 1$
$f(x)$
x
120
100
80
60
40
20
0
−10
−5
0
5
10


1
0
0
0.5
1
1.5
2
0
1
2



The current candidate for the busy beaver for five states. It is presumed that this Turing machine writes a maximum number of 1's before halting among all Turing machines with five states and the tape alphabet $\{0, 1\}$. Proving this conjecture is an open research problem.



1 Introduction

Below we give a brief overview of the actors and their verification operations that are relevant in light of reconfigurations in the system.

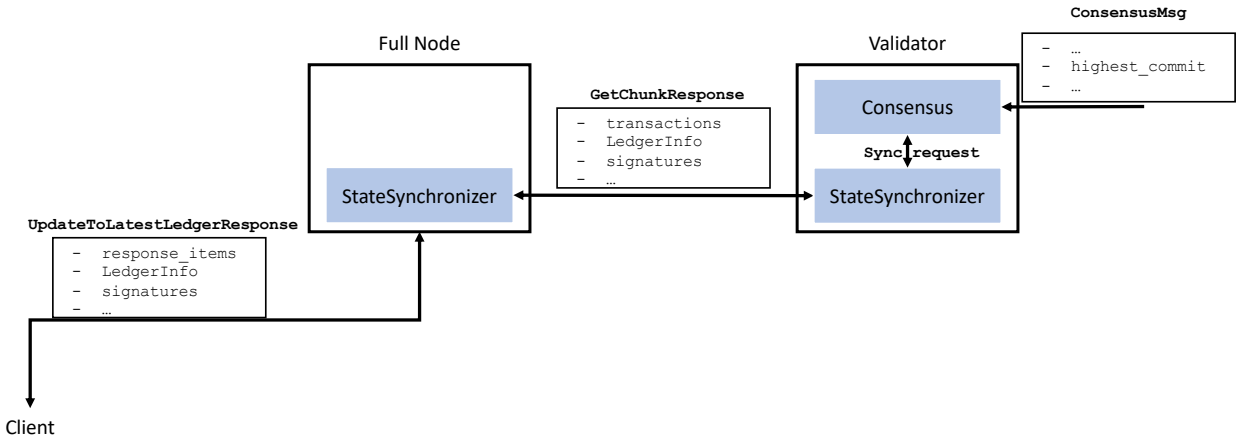


Figure 1: Main participants during verification and state sync

- **Clients** learn about the state of the blockchain via the `UpdateToLatestLedgerRequest` and `UpdateToLatestLedgerRes` messages. The response includes the `LedgerInfo` signed by the validator set of a given epoch (a client should have an ability to verify this validator set, see below). All the items in the response have merkle tree proofs relative to the version of the signed `LedgerInfo`.
- **StateSynchronizer** module is used by both the validators and full nodes: it is responsible to sync up the local storage to a given remote via the `GetChunkRequest` and `GetChunkResponse` messages. The `GetChunkResponse` includes the `LedgerInfo` signed by the validator set of a given epoch and a chunk of transactions with a merkle tree range proof relative to the version of that ledger info. Upon receiving a chunk response, `StateSynchronizer` verifies the signatures of the ledger info, verifies the range tree proof, applies the transactions locally and adds the `LedgerInfo` to the local storage in case it matches the local accumulator state. There is exactly one (latest) `LedgerInfo` per epoch kept in storage.
- **Consensus** module is run by the validators: it is using the state synchronizer to sync up the local storage in case one of the received consensus messages indicates that the peer's committed version is higher than the information stored in the local consensus speculative tree. In this case Consensus requests `StateSynchronizer` to sync up to a specific version, for which it has received a valid `LedgerInfo`.

2 Verification through epochs

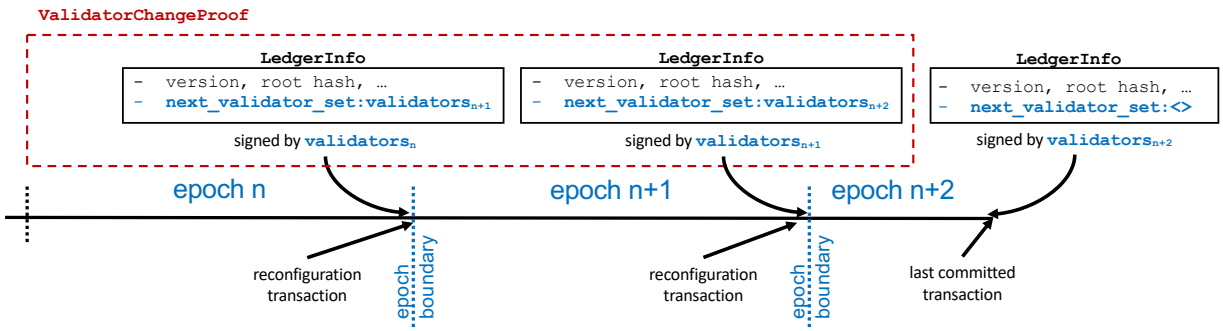


Figure 2: Latest `LedgerInfo` of an epoch keeps a validator set for the next epoch, forming a `EpochChangeProof` chain.

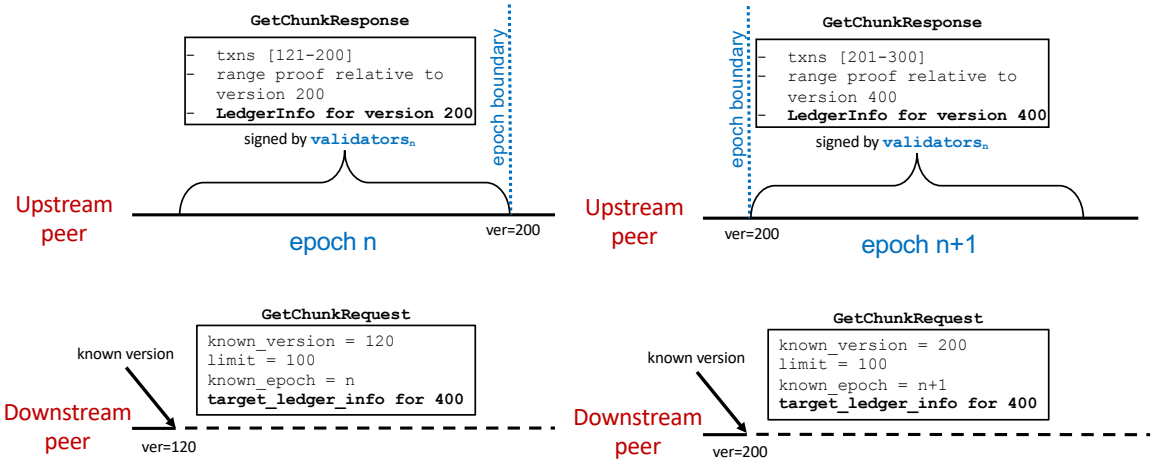
Fundamentally, each change in the validator set is part of the state managed by the ledger. In order to simplify the protocol the new validator set is also included in the last `LedgerInfo` of each epoch in a special field `next_validator_set`. As shown in Figure 2, storage keeps last `LedgerInfo` for every epoch, forming a `EpochChangeProof` chain, in which each new validator set is verified using the public keys of the previous validator set.

The `EpochChangeProof` chain is included in the `UpdateToLatestLedgerResponse`, hence every participant in the system can maintain the uptodate validator set (given a trusted validator set VS_i one can verify a `LedgerInfo` with a non-empty `next_validator_set` and thus update its trusted validator set to VS_{i+1}). Note that this approach assumes **some initial** trusted validator set, a challenge thoroughly described in Section 3.

2.1 State Synchronization through epochs

`StateSynchronizer` could verify the latest `LedgerInfo` similarly to the way it is done by the clients described above. However, this verification might not be enough: in order for a node to be able to serve state-synchronization to its downstream peers it should store all the end-of-epoch `LedgerInfos` as well.

Thus, the sync up protocol of `StateSynchronizer` makes sure that both the transactions and the `EpochChangeProof` are properly synchronized. The downstream peer is managing the known epoch and version; the upstream peer makes sure to deliver all the end-of-epoch `LedgerInfos` (see Figure 3):



(a) The `LedgerInfo` in response is determined by the last version of the request epoch.

(b) The `LedgerInfo` in response is determined by the target LedgerInfo of the request.

Figure 3: Response chunks do not cross epoch boundaries and carry proofs relative to the `LedgerInfo` of the chunk's epoch.

- `GetChunkRequest` carries the information about the known version and epoch of the requester.
- `GetChunkResponse` carries chunks that never cross epoch boundaries. In case a chunk terminates the epoch it belongs to, the `LedgerInfo` in the chunk also includes the `next_validator_set`, which helps the receiver to move to the next epoch (Figure 3a).
- One interesting aspect worth noting here is that in case a state synchronization request has a predefined target, the chunk response carries either the `LedgerInfo` of the given target, or the last `LedgerInfo` of the request epoch (the smaller between the two), to make sure that the receiver can still verify the response as depicted in Figure 3. In these cases the chunk can be smaller than the max allowed chunk size, just to make sure that the chunks do not cross epoch boundaries.
- The receiver of the `GetChunkResponse` executes the following verification steps:
  - Verify the signatures of the response `LedgerInfo` using the current validator set (chunk response must belong to the epoch of the request).
  - Verify the merkle range proof of the chunk transactions relative to the `LedgerInfo` of the response.
  - Apply the transactions to the local accumulator.
  - Verify that the local accumulator state matches the metadata of the response `LedgerInfo` and add this `LedgerInfo` to the local storage.

3 Waypoints

3.1 Motivation

A *cold* node (the one that comes online for the first time, or after a long period of being disconnected), faces a problem of the initial syncup because it might no longer trust the old public keys of the historical validators it is aware of, which could have been compromised since then (the so called "long range attack").

Waypoints can be used as an off-chain mechanism for selecting the initial trusted set of validators and specific points in the accumulator history. Waypoints can be published by anyone. The Aptos Foundation is also expected to publish waypoints on a regular basis (e.g., on `https://aptos.dev`). Any waypoint selected by clients can be embedded within the nodes (clients / full nodes / validators) as a trusted "starting point".

The trusted points in history determined by the waypoints can be used in the following situations:

- **Long range attacks**: trusted recovery relative to a recent point in time for a cold node that does not have an up-to-date information locally.
- **Genesis configuration**: a genesis waypoint certifies the genesis `LedgerInfo` with its initial validator set.
- **Emergency reconfigurations**: consider a disastrous scenario, in which a reconfiguration must be applied without an ability to generate a `LedgerInfo` with a valid set of signatures (e.g., private keys are lost, or the system needs to be forcefully upgraded to a previous point in time). The `LedgerInfo` of this reconfiguration cannot be verified via the standard signature checks, and it's going to be verified using waypoints (see more in Section **??**).

Obviously, being an off-chain mechanism there is no way to "prove" a waypoint, the nodes choose to trust waypoints for their initialization.

3.2 Waypoint structure

In order to provide a client with the initial trusted root hash and validators, it is enough for the waypoint to include the information about a LedgerInfo on the epoch boundary (the one that must include the validator set of the next epoch).

The waypoint representation remains a short string that can be easily included in a config or copied from an association website.

```
Waypoint
    // The version of the reconfiguration transaction that is being approved by this waypoint.
    version : u64
    // The hash of the chosen fields of LedgerInfo (including the next epoch info).
    value : HashValue
```

The fields of a LedgerInfo that are included in the waypoint value are `epoch`, `root_hash`, `version`, `timestamp`, and `next_validator_set`. Note that we cannot include all the fields because they might not be deterministic: different clients might observe different values of e.g., `consensus_block_id` of the latest `LedgerInfo` in an epoch.

The textual representation of a Waypoint is "version:hex encoding of the root hash". For example, the current testnet genesis waypoint is
`"0:a17035961b97359bcc22b628c1e269f543b265220d22a17e4a5b4e3aed86855b"`.

3.3 Verifying the latest LedgerInfo with waypoints

As described in Section 2, an `UpdateToLatestLedgerResponse` message carries a data structure for verifying validator changes, `EpochChangeProof`, which contains a list of all the epoch change `LedgerInfos`. A client verifies this list in an iterative fashion starting with its initial trusted `ValidatorSet`.

Waypoint verification is working slightly differently as shown in Figure 4. A waypoint version corresponds to the last version of an epoch (version 0 being somewhat special and corresponding to the last and only version of epoch 0). Hence, the `EpochChangeProof` in the response starts with the end-of-epoch `LedgerInfo` corresponding to the waypoint: this very first `LedgerInfo` is verified using the waypoint. The rest of the epoch changes is verified in a regular manner.

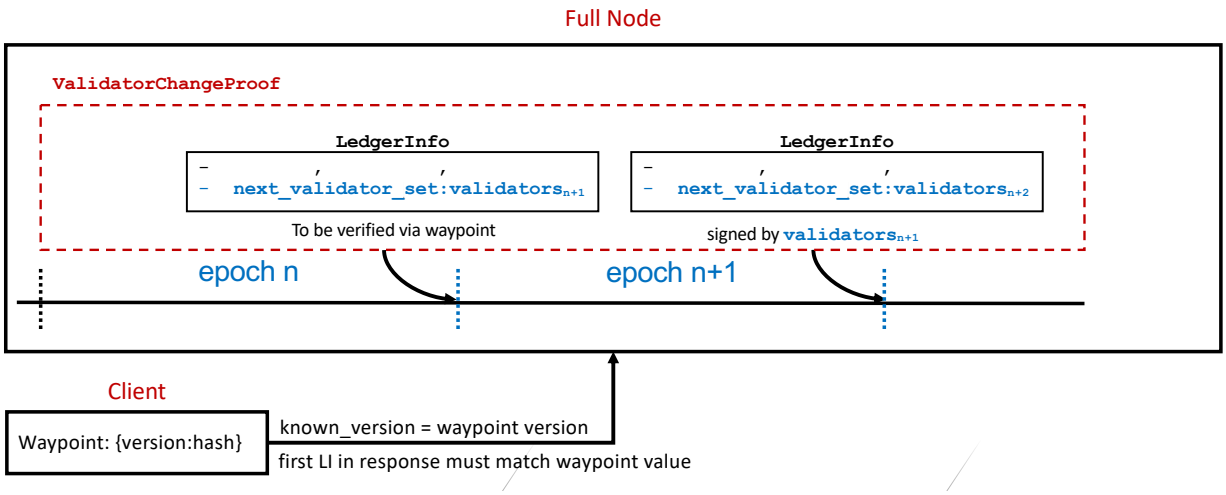


Figure 4: The first LedgerInfo in the `EpochChangeProof` is verified via a local waypoint.

3.4 State synchronization with waypoints

As described in Section 2.1, a node should possess the intermediate end-of-epoch `LedgerInfos` in order to be able to serve its downstream peers. Consider a following scenario:

- Full node A has reached epoch 100.
- Full node B is a downstream peer of A that is restarted with waypoint W_{80} corresponding to the last version of epoch 80.
- Full node C is a downstream peer of B that is restarted with waypoint W_{60} corresponding to the last version of epoch 60.

Waypoints are not transferable: the fact that B has decided to trust a waypoint W_{80} does not mean it can send this waypoint to C, which might still trust W_{60} only. Hence, B needs to be able to prove validator changes to C in the epoch range $60-80$, therefore it must bring the intermediate end-of-epoch `LedgerInfos` in a verifiable manner.

Unlike the protocol described in Section 2.1, the verification of intermediate end-of-epoch `LedgerInfos` **prior** to the waypoint (epochs below 80 in our example) cannot be based on signature verification, because B does not trust the old signatures. The verification of pre-waypoint items is relative to this waypoint:

- `GetChunkRequest` specifies the target type to be a `waypoint` with the version corresponding ot the version of the local waypoint.
- Each chunk response carries a `LedgerInfo` determined by the target waypoint version. This `LedgerInfo` is verified using a waypoint.
- The range of transactions in the response chunk carries a merkle tree proof relative to the version of the waypoint as well.
- The chunks that are sent back never cross epoch boundaries.
- In case the chunk terminates an intermediate epoch (e.g., a last chunk of epoch 62 in Figure 5), the chunk response carries an intermediate end-of-epoch `LedgerInfo`.

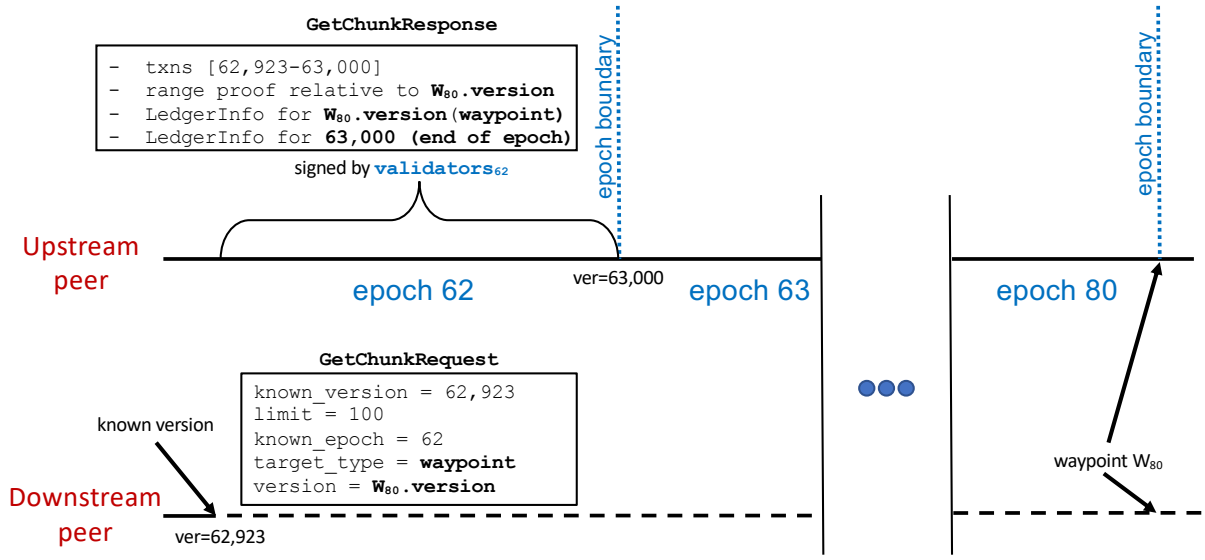


Figure 5: ChunkResponse carries the LI of a waypoint and an optional intermediate end-of-epoch LI.

- The receiver of such a chunk first executes the transactions and then verifies that the root hash and the metadata of the intermediate end-of-epoch `LedgerInfo` indeed correspond to its local accumulator state. In such case this end-of-epoch `LedgerInfo` is added to the local ledger store and can later be used for providing state synchronization to the downstream nodes.

3.5 Consensus and TCB initialization with waypoints

In this section we briefly describe the way consensus and TCB components utilize the waypoint mechanism during startup.

Consensus component trusts its local State Synchronizer. The startup order makes sure that Consensus starts only after State Synchronizer is fully initialized. In case a validator starts with a waypoint that is higher than the latest version stored in its local storage, the following chain of events occurs:

- State Synchronizer retrieves all the committed information up to the waypoint, as described in Section 3.4, and then considers itself initialized.
- Consensus starts after the initialization of State Synchronizer is complete and looks for the new root.
- Consensus detects that 1) the highest `LedgerInfo` in its local storage is higher than the previously stored consensus root, and 2) this `LedgerInfo` keeps the validator set for the next epoch (waypoint is set at the epoch change boundary). In this case, Consensus generates a new genesis block and starts a new epoch.

Unlike Consensus, TCB does not trust any other component in the system, hence, it keeps the waypoint of its own (in its local TCB config). Upon startup TCB verifies the latest `LedgerInfo` in the local store using its own waypoint (one should be careful to maintain the TCB and State Synchronizer waypoint copies identical because otherwise the startup verification might fail). TCB does not need to verify all the previous end-of-epoch `LedgerInfos` prior to its waypoint because it is not responsible for the synchronization services.