

Cache Component

Overview

All the consensus blocks in the Logos network, except the first genesis block, depend on some other blocks. When the blocks arrive in order, they can be validated immediately and stored locally. When the blocks arrive out of order, however, we cannot immediately tell if the blocks are valid or not. In this case, we put the blocks in the cache temporarily till all the blocks they depend on have stored locally. The initial motivation to build the cache component is to serve both the bootstrap component and the P2P component, because the blocks collected by them could be out of order. The goal of the cache is extended to be the one and only block database writer. Hence the blocks created by the core consensus component will also be written to the database through the cache. This document specifies the design of the cache.

The cache component has two main responsibilities:

- It holds all the blocks that cannot be immediately validated due to missing dependencies. Once the dependent blocks have been validated and stored locally, a block waiting in the cache will be validated and stored.
- It pipelines all the block database write transactions.

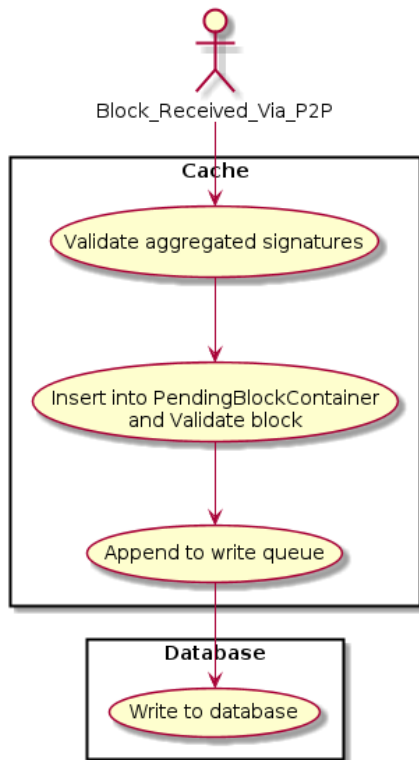
TODOs (This subsection will be deleted. It is to make the scope of the effort clear since we will have significant refactoring of some existing code.)


In this effort, we will:





- Have an API for validating (and storing if valid) blocks received from bootstrap and P2P, and also for storing blocks created by the core consensus.
- Change NonDelegatePersistentManagers to update the progress of validation. We will introduce a new struct, `ValidationProgress`, from which one can tell if the status of the block being validated, passed, failed, or pending. In case of pending, the `BlockValidationProgress` will include some information (such as the hash) of the first missing reliant block. When the block is revalidated, the fields already validated thus can be skipped.
- Merge the bootstrap cache code and the P2P cache code, and take performance optimization design from them.
 - from bootstrap cache: the blocks are naturally chained, and only the first block on a chain could be validated.
 - from p2p cache: we will use a multi-value hash table container to record the dependencies, and only revalidate blocks with dependencies cleared. Note that, depending on the detailed design, the container's size could be limited, less than the total number of chains, benefitting from the `NonDelegatePersistentManager` changes.
- Introduce a block write queue. Blocks ready to be written will be pipelined in the queue. We will also see if the blocks can help some of the cached blocks.
 - Given the current logos code, I don't see how this can happen to the blocks inserted from consensus. But it will be easy to implement and have low run time cost since delegates' caches should be empty most of the time.
 - From application's point of view, adding to the queue is equivalent to adding to the DB. As a result, when reading information from the DB, we also need to consider the blocks queued and the information derived from them, such as updated account balance. (This task is a future improvement. It will not be implemented in this release.)
- To address the concern that some blocks could stuck in the cache forever, we will detect the block that cannot make progress for a long time.



Execution Concept and Usage

When the bootstrap or the P2P component adds a block, a part of the block will be validated immediately, such as the two rounds of aggregated signatures. If the block passes this step of validation, then it is well formatted according to the LOGOS IDD and approved by a set of delegates. At this point, the only reason that the block could fail later validations will be due to a bad delegate set, which is an immediate recall situation or a case of long range attack, which will be handled by a component yet to be designed. So from bootstrap and P2P's point of view, the block should be considered as valid. The diagram below shows a typical usage scenario.



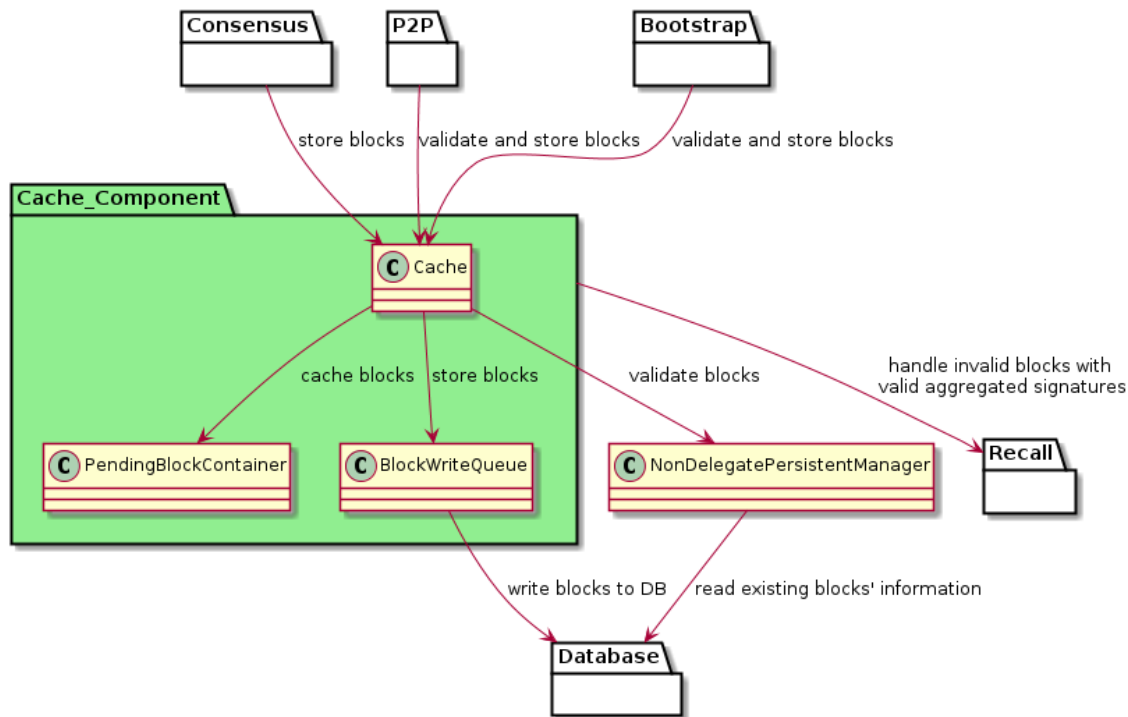
Internally, the cache component maintains  chains sorted by the blocks' sequence numbers. In the next step of processing, the block is inserted to one of the chains. Only if the block ends up being the 1st block on a chain, it will be validated, since the later blocks on the same chain depend on the 1st block (directly or indirectly).

- If the block is valid, it will be added to the BlockWriteQueue. On  block is stored, we will also see if any pending blocks can be re-validated now.
- If the block is invalid, the  have a bad delegate set (current or ). How to handle the situation is yet to be designed. For now, we log the block and discard it.
- If the block has missing ces, then we keep it on the chain, and insert its “progress mark” (see below) in a DependencyTable. Only when its entry is cleared from the DependencyTable, we will re-validate the block.

To support the efficient operation of the cache, the current NonDelegatePersistentManager will be refactored. So that, when validating a block, tion to provide a yes/no answer with an error reason, it also provides a “progress mark” for a “pending” block, where a pending block is the one with missing reliances, and the progress mark identifies the  missing reliance. With this modification, we will significantly reduce the unnecessary re-validation. In addition, when the time comes to re-validate the block again, we can skip the fields in the block that are previously validated already.

Validated blocks are appended to the BlockWriteQueue and will be written to the DB one by one, since LMDB only allow one write transaction at a time.

Interfaces



The Cache provides an interface to bootstrap, consensus, and P2P to validate and store blocks. The Cache uses the service of the database and NonDelegatePersistentManager. It also needs a way to report cached blocks which have valid aggregated signatures and are later proven to be valid. Below is a list of the interface functions.

```
//should be called by bootstrap and P2P
bool AddEpochBlock(EBPtr block)
bool AddMicroBlock(MBPtr block)
bool AddRequestBlock(RBPtr block)

//should be called by consensus
void StoreEpochBlock(EBPtr block)
void StoreMicroBlock(MBPtr block)
void StoreRequestBlock(RBPtr block)

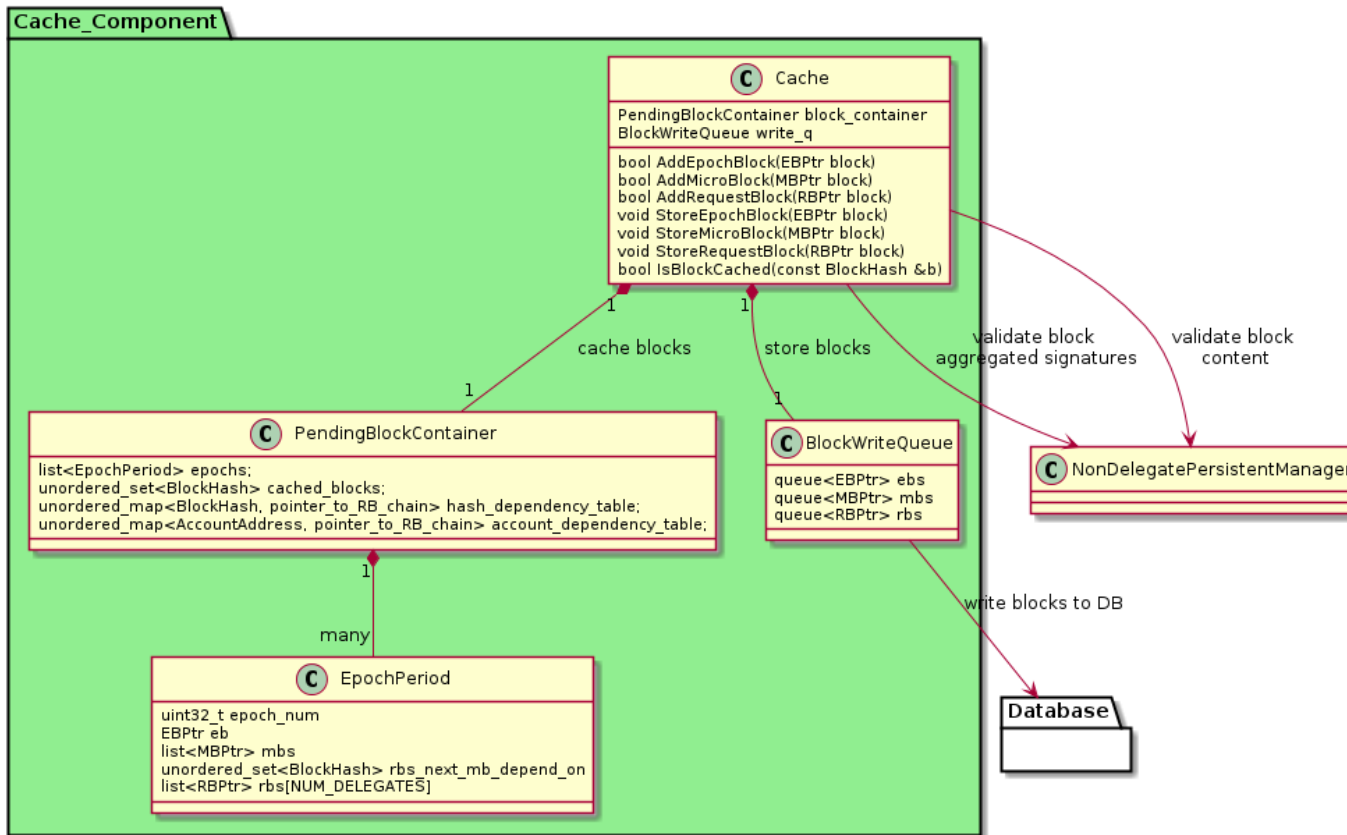
//should be called by bootstrap
bool IsBlockCached(const BlockHash &b)
```

The first set of functions is for the bootstrap and the P2P to add blocks. They return if the block should be considered as valid from the bootstrap or the P2P's point of view. In more detail, the block's aggregated signatures are verified. The functions return true if the signatures are valid. Even though the blocks could have missing reliances (i.e. pending in the cache) or be proved to be invalid later. From the bootstrap or the P2P's point of view, the block can be considered valid.

The second set of functions is for the consensus to add blocks. Since the blocks are already validated in the consensus session, they do not need to be validated again, so they are write to the DB directly, through the BlockWriteQueue.

The function IsBlockCached() is for the bootstrap to find out if a block is cached or already validated.

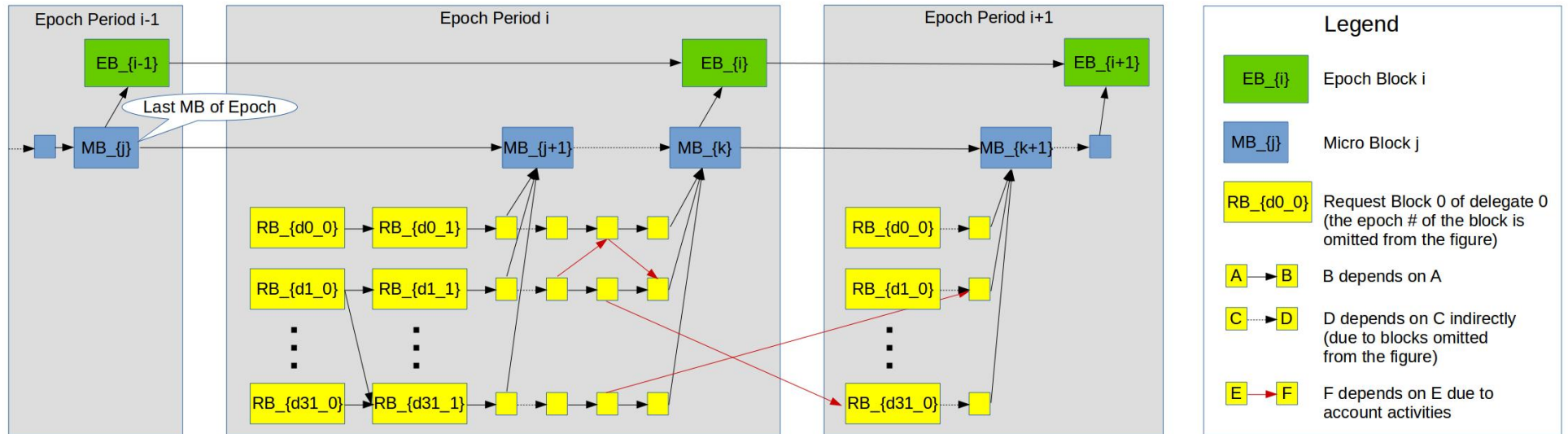
The Cache Class



The cache component has a main class, Cache, which implement the interface functions, and several structs for holding information used by the Cache class. The PendingBlockContainer holds all blocks with missing reliant blocks, and the BlockWriteQueue contains all validated blocks to be written to the database.

PendingBlockContainer

Block Dependency



The figure above depicts the block dependencies. In the PendingBlockContainer, the blocks are arranged on the chains similarly. There are 32 request block chains per epoch period, 1 epoch block chain, and 1 micro block chain. When a block is received, it is added to its chain in its epoch period. If it is the first block on the chain, we validate it. If it has missing reliances, we add the first missing reliance in one of the two hash tables:

- The `hash_dependency_table` is for "previous-gap" kinds of errors. The key of the hash table is the hash of a block or the hash of a request and the value is a pointer to the chain. When the block or the request is validated, the block at the beginning of the chain will be updated.
- The `account_dependency_table` is for "insufficient-balance" kinds of errors: the key is the account address with insufficient balance and the value is a struct that consists of the current balance, the target balance, and a pointer to the chain. When the account has a new "receive", the current balance will be updated, if the new balance is enough, then the block at the beginning of the chain will be re-validated.

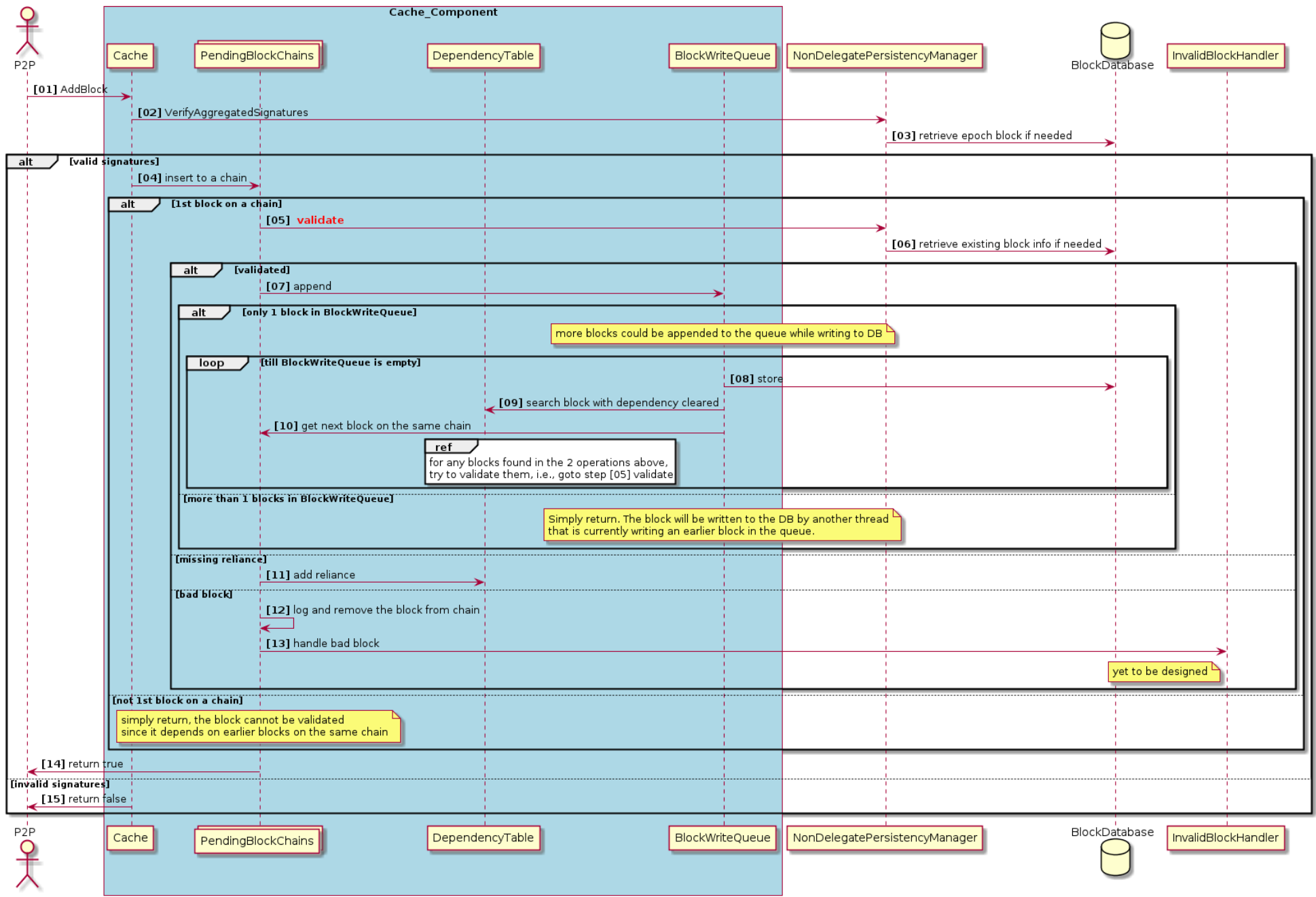
Things to take care: The hash table could have multiple entries for the same chain. For example, blocks with sequence number 1, 2, and 3, are received in the reverse order. Then both 3 and 2 will be inserted in the table. The implementation needs to make sure the extra entry in the hash table is handled.]

The PendingBlockContainer struct has a list of EpochPeriods. An EpochPeriod has one EpochBlock (part of the global EpochBlock chain), one MicroBlock chain (part of the global MicroBlock chain), and 32 RequestBlock chains, for holding all the pending consensus blocks. It also maintains a set of hashes which is populated with the hashes of the RequestBlockTips field of the next MicroBlock. Once a tip (a RequestBlock) is stored to the database, its hash is removed from the set. When the set is empty, we can try to validate the next MicroBlock.

The PendingBlockContainer also maintains a set of hashes of all cached blocks for the bootstrap to query if a block is pending or not.

Sequence Diagram of Adding a block

Add Consensus Blocks to the Cache



One thing to note from the sequence diagram is the "ref" box at the center of the diagram. Once a block is stored in the database, there is a chance that more blocks could be validated. We will try to find the blocks in the DependencyTable, and on the same chain as the block just stored. If found any such blocks, we will validate them.

Implementation Considerations

Refactoring NonDelegatePersistentManager

The NonDelegatePersistentManagers need to be significantly refactored. When validating a block, it updates a BlockValidationProgress struct, from which one can tell if the status of the block being validated, passed, failed, or in-progress. In case of in-progress, the BlockValidationProgress will include the first missing reliance. When the block is re-validated, the BlockValidationProgress updated previously will be passed in, so that fields validated previously do not need to be validated again.

Stale Block Cleanup

One of the concerns of the cache is that some blocks could stuck in the cache forever. It will be good to be able to diagnose the situation, and optionally remove the block that stuck. One way to do it is to have a periodical task to check the progress of the 1st blocks on the chains in the first EpochPeriod in the cache. If some blocks stuck for, say one hour, then we log the block and delete it.


Future Improvement

NonDelegatePersistentManager

We can add double spend detection to the NonDelegatePersistentManager. We define double spend as the same account, the same previous hash (and same sqn), but two different transactions (different target and/or amount)

Block Write Queue



From application's point of view, adding to the queue is equivalent to adding to the DB. As a result, when reading information from the DB, we also need to consider the blocks queued and the information derived from them, such as updated account balance.

When the validation of a block returns an error, such as “gap_previous” or “insufficient_balance”, which indicates a missing reliant block, the BlockWriteQueue is checked to see if it contains the missing  block. If the missing block is contained in the queue, then the caller is blocked until the missing reliant block is written to the database. Otherwise, the caller proceeds and returns the error.

The cache component will expose two more interface functions:

```
void HashInQueue(BlockHash & hash);  
void ReceiveInQueue(AccountAddress & address);
```

The first function HashInQueue() should be call for the “gap_previous” kind of errors. The second function ReceiveInQueue() should be call for the “insufficient_balance” kind of errors. When the functions return, caller should try to re-validate its block again. Note that the re-validate could fail again.

When one of the interface fun is called, internally, the hash will be used to match any of the block or request hashes, or the address will be used to match any of transaction targets in the request blocks. If there is a match at block in the queue,  **caller thread should wait for the completion** of the write transaction of the block.