libconsensus

Jorge Timón

October 13, 2016

Contents

Abstract

In Bitcoin (a distributed consensus system), there's a set of rules which define whether a block is valid or not. In a typical software system, those rules would be described in a specification document written in a natural language that would then be translated to one or more software implementations. But since software deployment coordination is critical to the security of the system when the software validation of the rules is changed, the documentation doesn't necessarily have preference over the deployed implementation when it comes to what is the specification to follow.

This produces an unnecessary network effect in favor of the implementation deployed more widely, in this case, Bitcoin Core. Not all of bitcoin is consensus critical, there's network messages, storage, local relay and mining policies, wallet-specific code, code specific to maintain indexes, GUI specific code... This puts alternative implementations in an unnecessarily complicated position when it comes to review and upgrade for changes to the consensus rules.

Even if Bitcoin Core was adverse to the existence of alternative full-node implementations, encapsulating consensus-critical code is necessary and urgent for Bitcoin Core because it will

enormously increase the number of potential contributions to the project by drastically reducing the probabilities that one particular change needs to touch any file containing consensus-critical code and therefore reduce the demand for critical review.

At the same time, the attack surface area of a bitcoin node is big, and this sometimes leads to necessary complexity in the code to protect a node from different potential attacks. These complexities must be kept separated from the specification of the consensus rules of the chain. But experience shows us that in the case of distributed consensus systems like Bitcoin, for the specification to be truly unambiguous it needs to be written directly in code.

Since Bitcoin Core is currently the more widely adopted full-node implementation, it makes sense to exact the first it from

1 TODO Motivation

This document describes a detailed plan to separate consensus critical code from Bitcoin Core.

to be able to give reasonable guarantees that the same specification will be exactly replicated by all nodes (that want to replicate it, see BIP99 for a classification of potential changes to the specification) These nodes may be implemented and integrated in diverse stacks with diverse machines. A common specification of the consensus rules is necessary (but not sufficient) for all participant to converge on the same view of the history of global states of the system. But this specification is currently coupled with Bitcoin Core's implementation of a full node, which reduces its clarity and ease of modification.

Additionally, alternative implementations of the consensus protocol are currently forced to chose between providing an incomplete full node and relying on trusted nodes using the reference implementation (Bitcoin Core)

Matt Corallo came up with the idea of exposing the newly encapsulated... TODO

2 Current situation

In 0.13, Bitcoin core is divided in the following basic packages:

```
LIBBITCOIN_SERVER=libbitcoin_server.a
LIBBITCOIN_COMMON=libbitcoin_common.a
LIBBITCOIN_CONSENSUS=libbitcoin_consensus.a
LIBBITCOIN_CLI=libbitcoin_cli.a
LIBBITCOIN_UTIL=libbitcoin_util.a
LIBBITCOIN_CRYPTO=crypto/libbitcoin_crypto.a
LIBBITCOINQT=qt/libbitcoinqt.a
LIBSECP256K1=secp256k1/libsecp256k1.la
LIBBITCOIN_ZMQ=libbitcoin_zmq.a
LIBBITCOIN_WALLET=libbitcoin_wallet.a
```

The figure 2.1 contains a simplified UML components diagram of the current structure of Bitcoin Core, showing the executable binaries as interfaces

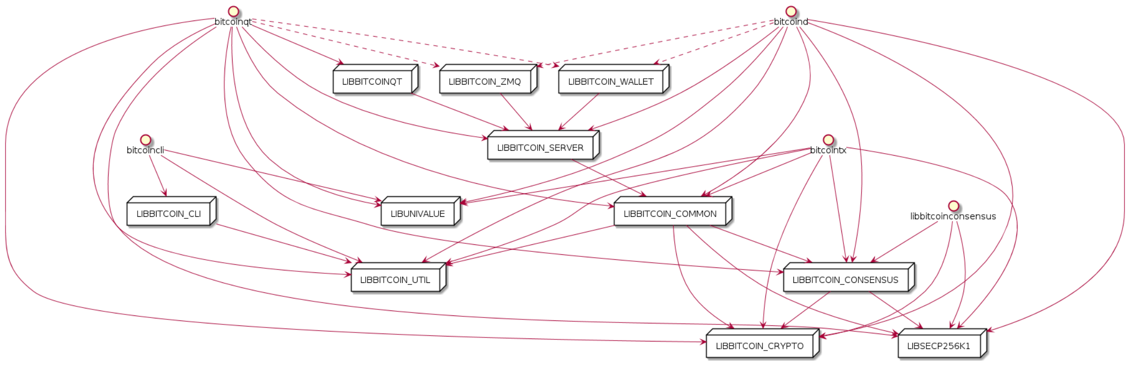

Figure 2.1: Current Bitcoin Core architecture

The arrows mean dependencies. The dotted discontinuous arrows mean that the dependency can optionally be removed at compile time (it's only used for `LIBBITCOIN_WALLET` and `LIBBITCOIN_ZMQ`, which are optional to both bitcoind and bitcoin-qt).

Since some dependencies are implicit, we can remove some arrows to simplify things as shown in figure 2.2.

Also, bitcoin-cli, bitcoin-qt, `LIBBITCOIN_ZMQ` and `LIBBITCOIN_WALLET` are encapsulated enough while not being too relevant to libconsensus that we can remove them from the picture (see figure 2.3).

libbitcoinconsensus currently only exposes `VerifyScript()`. We will refer to all the previous work, including exposing `VerifyScript()` (see PR #4692 and its replacement #5235) and other preparations up to 0.13 as "phase 1". After phase 1, we actually have some code we can call libconsensus.

Before discussing phase 2, it is convenient to detail the current state further, including concrete files within the packages as shown in figure 2.4. The consensus package already contains more files than are needed for `VerifyScript()`, but which contain mostly consensus code without also including globals or undesired dependencies that would break libconsensus. Those files are marked

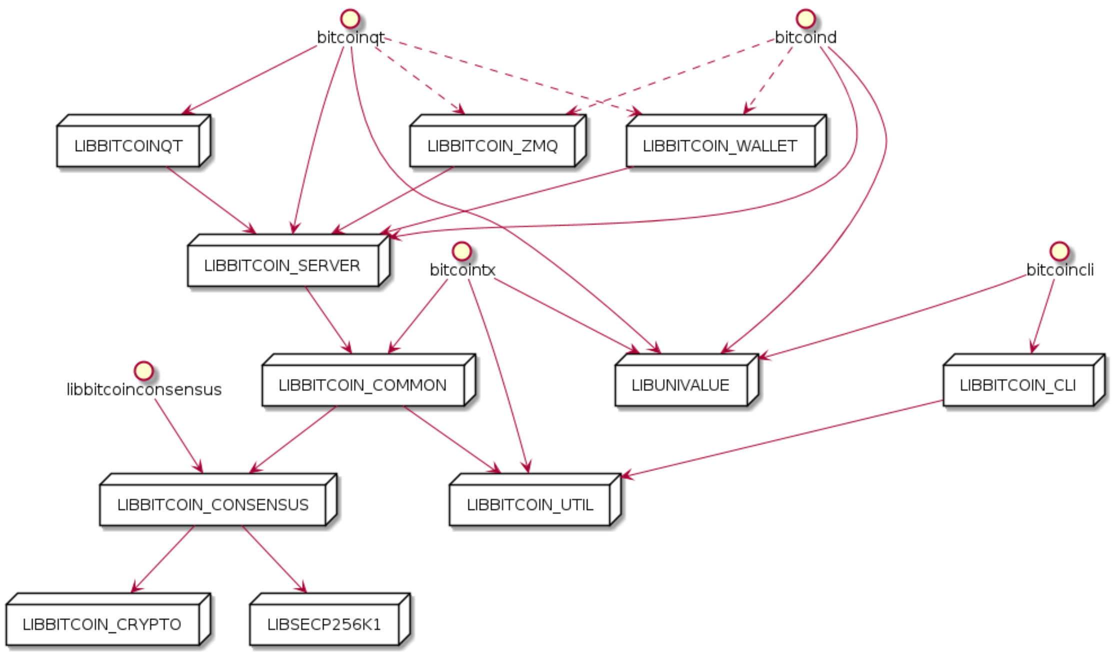

Figure 2.2: Simplified Current Bitcoin Core architecture

as green. Files marked as red contain consensus critical code but cannot be added to the consensus packages because they contain storage related dependencies or more code that should not be moved to libconsensus (main.o, chain.o, coins.o, pow.o and versionbits.o). Files marked with orange require further discussion.

For example, amount.h is necessary for libconsensus, but it contains CFeeRate, which should be moved out of the consensus packages. CFeeRate is defined in amount.cpp, which is currently **not** used by the consensus packages. This situation can be resolved by moving the class CFeeRate to its own module, for example, policy/feerate.o as in PR #7820.

The only consensus function that uses utilmoneystr.o is `Consensus::CheckTxInputs()`, currently in main.o. If the amounts in some errors in that functions are shown in satoshis instead of formatting them to full bitcoins using `FormatMoney()`, then utilmoneystr.o doesn't need to be in the consensus package.

If we don't want to have compat/endian.h in libbitcoinconsensus, crypto/common.h and seralize.h need to be decoupled from it.

If we want to include script/sigcache.o in the the consensus library, we need to consider its dependency memusage.h and decouple it from random.o and util.o.

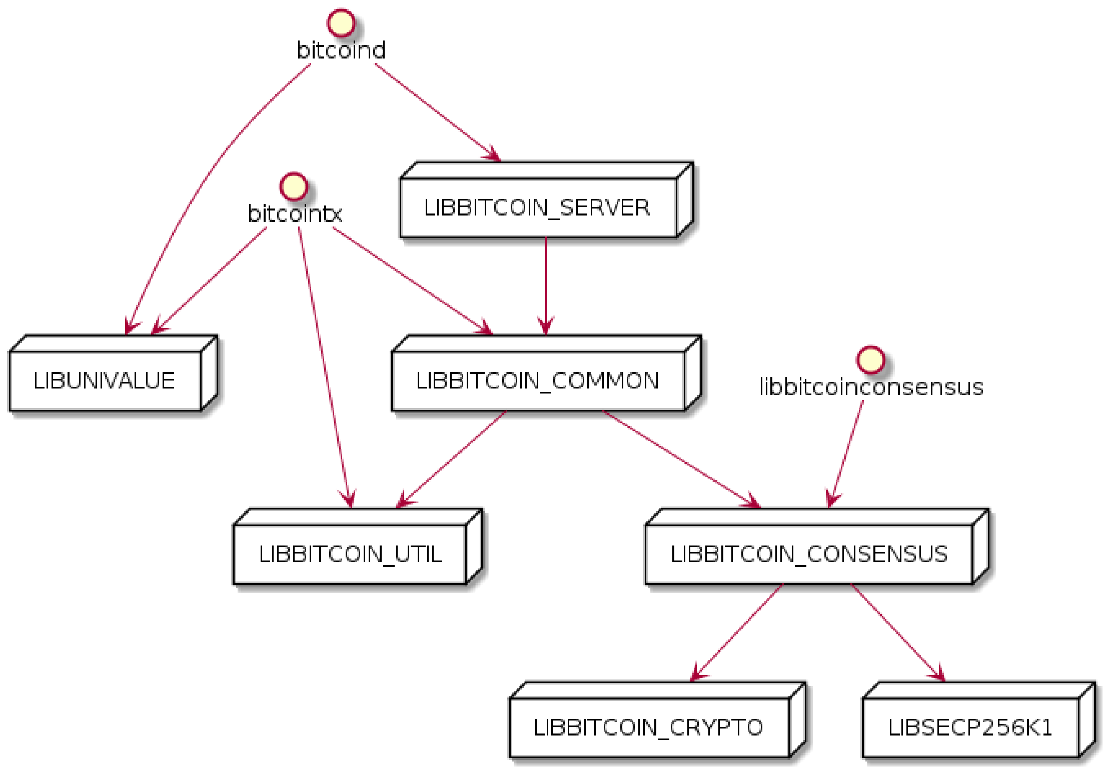

Figure 2.3: Simplified Current Bitcoin Core architecture excluding cli, qt, zmq and wallet.

3 Phase 2: Expose `VerifyHeader()`

As a next function to expose in libconsensus, we can use `VerifyHeader()`. Although it relies on chain storage (`CBlockIndex` class in Bitcoin Core), it is probably the simplest verify function to expose. SPV wallets could call this functions for fully verifying the header (including difficulty adjustments) instead of only checking the proof of work.

Since we want libbitcoinconsensus to be independent of the storage, CBlockIndex needs to be replaced with some sort of interface compatible with libconsensus' C API. For example, #8493 introduces a C struct `BlockIndexInterface` containing function pointers as fields. These functions are used to access the header index. For libconsensus, each header is just a void pointer used only with the help of `BlockIndexInterface`. Since each caller may have a different structure for their header object, we cannot assume any particular one and we take them as void pointers.

To move the header validation functions and their dependency pow.o to the consensus package, we need to decoupled the from chain.o (where `CBlockIndex` is defined) using this interface. Figure 3.1 shows the result of applying phase 2.

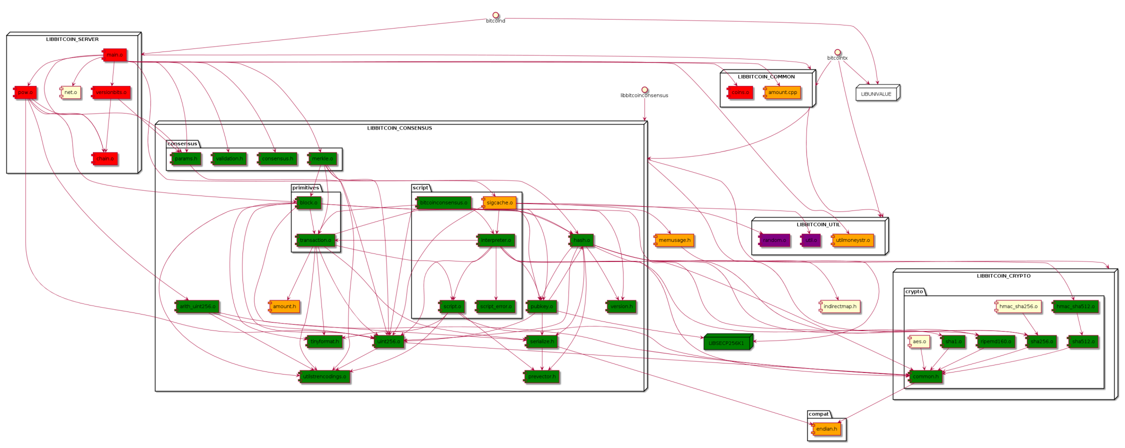

Figure 2.4: Detail of the current state of libconsensus

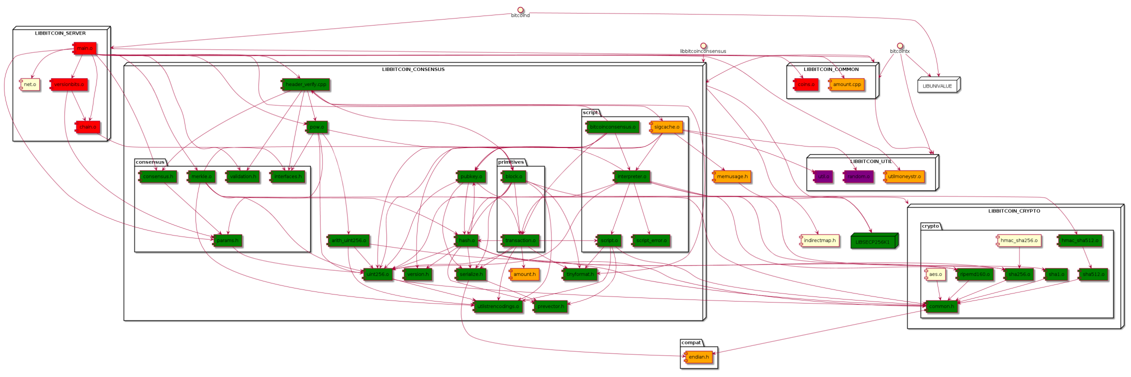

Figure 3.1: Phase 2: Expose `VerifyHeader()`

4 TODO Phase 3: Expose `GetConsensusFlags()`

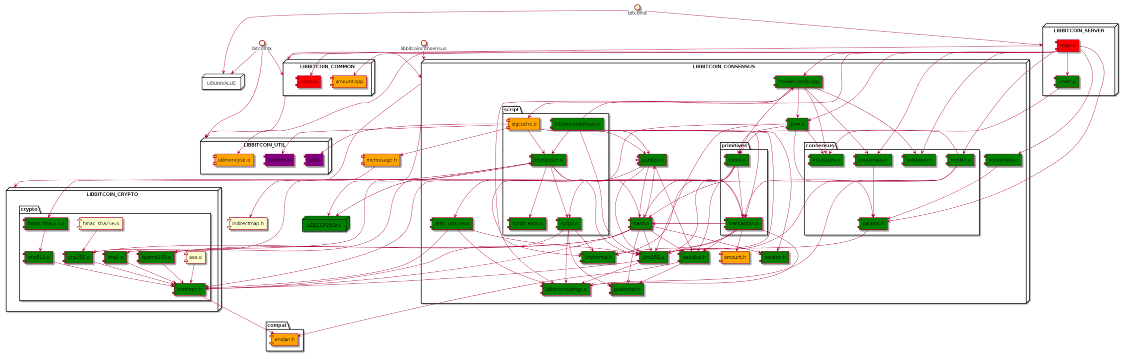

Figure 4.1: Phase 3: Expose `GetConsensusFlags()`

5 TODO Phase 4: Expose `VerifyTx()`

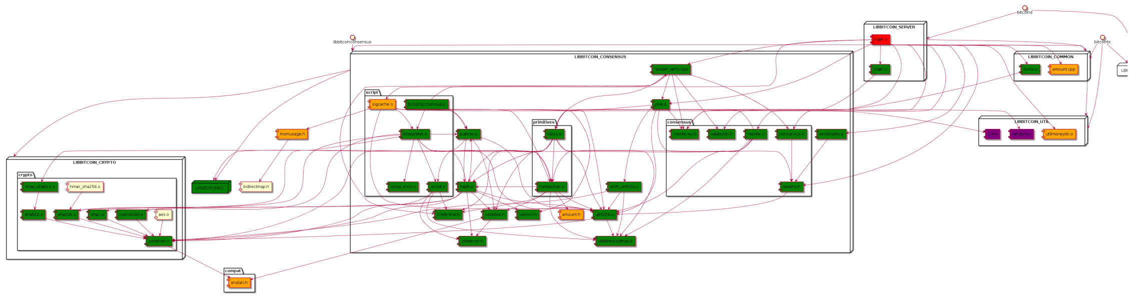

Figure 5.1: Phase 4: Expose `VerifyTx()`

6 TODO Phase 5: Complete libconsensus API (expose `VerifyBlock()`)

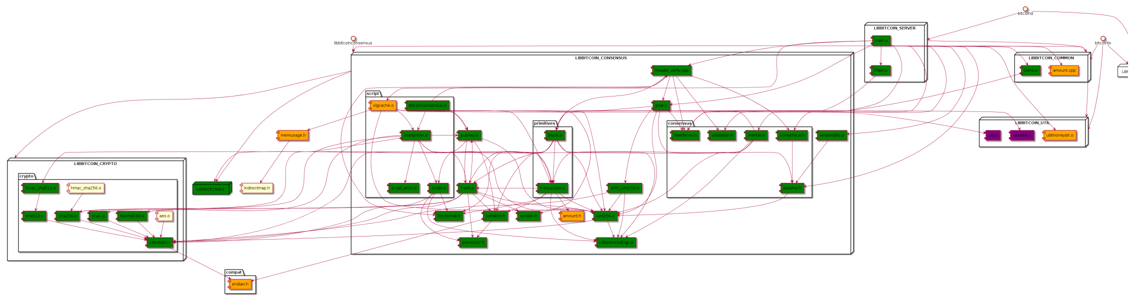

Figure 6.1: Phase 5: Expose `VerifyBlock()`

7 TODO Phase 6: Separate libconsensus to its own repository

7.1 Phase 6.1: Remove non-consensus code.

7.2 Phase 6.2: Move all consensus code to the same directory

7.3 Phase 6.3: Create a sub-repository or subtree

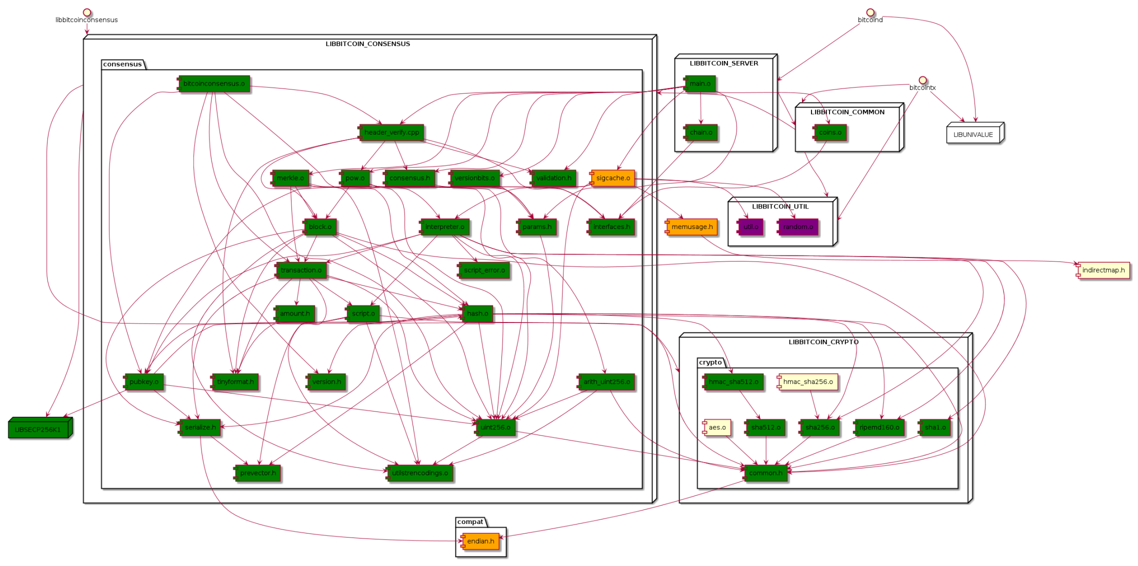

Figure 7.1: Phase 6: Separate libconsensus to its own repository

8 TODO Phase 7: Make Bitcoin Core eat its own dog food

9 TODO Final proposed complete libconsensus' C API