Protocol Upgrade Table Design

@acruikshank, July 2019
Status: Proposal

Background

Changes to any Filecoin software that alters on-chain data structures or the code that validates them currently require that we reset the network and that all participants upgrade to a new release before reconnecting to the network. This is growing untenable in our development networks and will be completely unacceptable in testnet or mainnet.

For the purposes of this document, **protocol upgrade** is any change to go-filecoin code that affects block validation. This includes block and message data structures, their validation rules and all state transition code implemented by actor methods. The protocol version does not cover wire protocols for transmitting blocks or messages. These wire protocols will likely evolve independently, and require a separate versioning scheme which will not be discussed here.

In order to minimize forks around protocol upgrades, implementations including go-filecoin must switch to the new protocol at the same block height. We need a mechanism to track and schedule protocol upgrades so that we can write code that changes its behavior across protocol version boundaries.

This document focuses on how to schedule protocol changes and provide access to that schedule throughout the codebase. A subsequent design document will discuss how that schedule could be used to ease development of protocol transitions.

Goals and Assumptions

We start with the following **assumptions**:

- Some external process will generate consensus among Filecoin operators that a specific set of protocol changes will go into effect at a specific block height.
- Operators of go-filecoin nodes that agree to a protocol upgrade and hope to stay compatible with the spec must download a new binary or new release of the source code prior to the version transition.
- We can make some contributions to the Filecoin specification to accomplish these goals. Specifically we can propose new built in actors.

With this in mind, we establish the following **goals**:

- All solutions must be compatible with the Filecoin spec.

- A schedule of protocol versions and their starting blockheights will be easily available throughout the codebase so that protocol implementation can make decisions about how to behave at the current block height.
- Every network (devnets, nightly networks, usernets, testnets, mainnet, etc.) may have its own protocol upgrade schedule.
- The go-filecoin code is testable at all protocol versions and transitions from one protocol to another can also be tested.

The following are **non-goals** of this document:

- This document does not discuss any aspect of communicating upgrades to the network or reaching consensus that they will be implemented.
- This document does not attempt to solve problems related to a high percentage of miners failing to install protocol upgrades.
- This document only concerns breaking protocol changes (hard forks). It does not describe how to implement backwards compatible changes without forcing a protocol upgrade.

Summary of Proposals

This document will propose the following changes to the go-filecoin codebase:

1. A Protocol Version identifier will be a string conceptually and implemented as an enumeration.
2. A network name will be stored in actor state in the genesis block.
3. The network name will be used in all network protocol names to prevent communication between networks.
4. A protocol upgrade comprises a network name, protocol version and block height from which the version takes effect on the network.
5. A protocol upgrade table will hard-code all relevant protocol upgrades into it.
6. A node will use the upgrade table to provide access to the protocol version at a given block height on the current network.

Protocol Version

The protocol version is an identifier for a specific of changes that affect the protocol. From now until after mainnet launch we will have a variety of motivations for protocol upgrades. Initially we will upgrade on a regular schedule as the implementation approaches the spec. The spec is still a work in progress, so spec changes will also drive upgrades. Eventually we will have a process for the community to agree upon protocol changes (probably referred to as Filecoin Improvement Proposals or FIPs).

Version identifiers occupy the same meta-protocol category of information that also includes the Filecoin specification. As such, the version identifier should never appear on chain. We

should maintain the identifiers in the codebase to map an implementation at a certain block height with the protocol version that specifies it.

proposal
Versions will be maintained as an enumeration in the codebase for the following reasons:

1. Version IDs should be typed so the compiler will catch mistakes.
2. Version IDs should be descriptive because we need to categorize the source of the change. For example, we should be able to distinguish a pre-testnet upgrade from a FIP.
3. Version IDs should be non-numeric (or not simply numeric) because the source of the protocol will often have its own numbering. For example, it will be confusing for FIP-2 to correspond to protocol upgrade 34.

We will begin by labeling upgrades DR1, DR2, DR3, etc. for development release, to signal that the specific upgrades are determined by the priorities of the Filecoin development team and the timing of release, rather than any specific change to the protocol.

alternatives
This could be a number or a string. This has some small advantage, because, as we retire old upgrades, the underlying numbers of an enumeration will become unstable in a way that integers or string versions would not. Also either type would be easier to reference from outside the source code than enumerations. I do not anticipate the first issue to be a problem, and, if we need to reference upgrades at all outside the codebase, we can represent it as a string relatively easily.

Network Identifier

We will be operating many different networks in the near term. We currently operate a nightly network and a user network. We can also create ad hoc developer networks. Our functional tests instantiate their own networks. In the long term, we expect multiple test networks to run side by side with mainnet.

These networks are primarily defined by their genesis block, since having the same genesis block is a requirement for nodes to participate on the network together. We will need to tie protocol upgrade schedules to networks. We will also need to run the same binary on multiple networks, so we cannot hardcode the network identifier in the source code.

proposal
We embed the network identifier within the genesis block. Specifically, we store it in actor state in a new singleton Config actor. This has the following advantages:

1. Embedding the network name in the genesis block ensures every node on a network will use the same network name.

2. Storing the name in actor state means no additional fields in block headers, and the network name will still alter the block CID via the state root.
3. Mechanisms for generating actors in genesis and retrieving information from actor state already exist.

This proposal will require that we create a new built-in actor to hold the network name. We will also need to introduce a new network name parameter to gengen, and have it create the actor with the appropriate configuration in the way it already creates other initial actors.

Once we have a network name available in the instance, we can use this name to make all of our libp2p protocol e.g. the peer DHT and block pubsub topic unique to users on a particular network. We can also include the network name in Hello protocol and explicitly disallow connections from other networks. This is an easy way to prevent communication from other networks from accidentally polluting the current network.

alternatives
We could store the network in config or elsewhere in the repo. Having a network name out of sync with the genesis block could get very messy. This fact should outweigh any considerations related to the difficulty in storing the name in the genesis block.

We could store the network name in an actor that already exists. StorageMarket is a pretty decent candidate, but the network name pertains to more than just the storage market, and that actor already does a lot. The Network actor is an attractive target as well, but that actors role in storing all unclaimed network rewards makes this seem risky.

The `minerAddress` field in the genesis block is probably unused, so we can put a string there and read it directly at node init

The `ticket` field in the genesis block is probably unused, so we can put an integer there (Ethereum identifies networks with integers)

The state tree is just a HAMT. We can put *any key we like* in it. There is no technical necessity that top-level keys can only be actor addresses. So we could store "network" -> "ropsten" in the state tree and read it at node boot.

Protocol Upgrade Schedules

We need a mechanism to define when a protocol upgrade should go into effect on a particular network. This table should update automatically when an operator updates go-filecoin to a version that is aware of a new protocol upgrade. This implies that the upgrade schedule should be checked into the source repository.

We will need to use the same binary on multiple networks, and we don’t expect the block times of upgrade transitions to be consistent across networks. For example, we expect to

perform multiple protocol upgrades during development and test networks, but mainnet will launch with the only the latest protocol version. This implies the schedule should be capable of containing schedules for different network.

The purpose of the schedule will be to allow Go code to query for the active upgrade at a block height on the current network. As such, the schedule must be implemented in Go. The actual configuration could be implemented as a separate file in a data notation like JSON. It is not clear at this time that this will provide any benefit so we propose that the upgrade schedule be hard-coded.

proposal

We will implement an UpgradeSchedule struct with methods to retrieve a protocol upgrade version given a network and a block height. The main schedule will be a populated with a static initializer as follows.

```
type UpgradeVersion uint

const (
  DR1 = UpgradeVersion(iota)
  DR2
  DR3
)

var protocolUpgradeTable ProtocolUpgradeTable

func init() {
  protocolUpgradeTable = NewProtocolUpgradeTable(
    upgrade("nightly", DR3, 0),
    upgrade("user", DR1, 0),
    upgrade("user", DR2, 256000),
    upgrade("user", DR3, 1399000),
    upgrade("test", DR1, 0),
  )
}
```

The UpgradeTable will provide a single method to provide the upgrade name for a network at a specified block height:

```
// Version returns the protocol version active at a block height for the given network
func (put ProtocolUpgradeTable) Version(network string, blockHeight uint64) ProtocolVersion
```

Code that needs to use the table should not be responsible for identifying the current operating network. So we propose adding the following two methods to an object that gets dependency injected by Node:

```
// Network returns the name of the network on which this node is mining
func (e *Expected) Network() string

// ProtocolVersion returns the protocol for this network at the given blockheight
func (e *Expected) ProtocolVersion(blockHeight uint64) ProtocolVersion
```

There are a couple of reasons for choosing Expected for this functionality. The first is that it already has access to the stores it needs to retrieve fetch the network. Expected is also in the consensus package, so code that will need to know the protocol version will probably already depend on it.

Alternatives
There are many implementation options that offer various tradeoffs between complexity and ease of use:

- Upgrades could be populated from a file.
- Upgrades could be specified in Node when the ProtocolUpgradeTable is constructed.
- ProtocolUpgradeTable could be given a network when constructed and accessed directly.
- The active network could be determined before the upgrade table is constructed, and it could be constructed with only the upgrades for the active network.

We are assuming that the proposed design will be the simplest from a code and architectural standpoint, but any of these alternatives may prove better once implementation has begun.

Testing

All code that depends on protocol upgrades should define the ProtocolVersion function in an interface that will be fulfilled by Expected. When unit testing, this function should be mocked or faked to provide the desired version.

Integration and functional testing are a little more complicated. There are a couple of approaches that could work. There is no avoiding the fact that integration testing across protocol boundaries will be challenging.

Tying protocol upgrades to networks allows us to design the upgrade table specifically for integration tests by choosing the network name in genesis blocks used in testing. Conceivably we could create multiple test network names which would allow us to test each protocol version in isolation (i.e. “integration test1” would only have upgrade DR1, “integration test2” would only have DR2, etc).

Gaining specific control over block heights would allow us to write integration tests for protocol transitions. This control would provide many other benefits. The design of this time travel mechanism is worthy of its own design document. This document only asserts that this work should be prioritized.