Filecoin component architecture

@anorth
October 2019
Status: Proposal, seeking feedback

Background

The go-filecoin application is built as a monolithic binary, encapsulating core blockchain capabilities along with storage mining and market application functionality. It includes the Rust sectorbuilder module as a static-linked library, called via FFI. The sectorbuilder controls the sector layout, packing, and proof scheduling routines.

The new (Oct. 2019) specification identifies some clear component boundaries, separating the core blockchain consensus functionality from the storage mining, storage market and piece data transfer subsystems. While not strict requirements for a compliant Filecoin implementation, at a high level these component boundaries provide sound architectural guidance, and a common language for talking about Filecoin implementation internals.

We expect large-scale miner operators to customize their storage architecture to achieve economies of scale. This requires configuration or customisation of the storage mining and market software, but not so much the blockchain node software. Such customisation is currently difficult, due to the monolithic and poorly-componentized internal architecture of Go-filecoin.

Some parts of a Filecoin node are more critical to the security of the network than others. These tend to be the core blockchain components rather than the miner-customized components. We intend and expect there to be multiple Filecoin implementation, and are aware of some in early stages of development. The network security benefits of implementation diversity derive from the core blockchain components more than of the storage mining components. Indeed, *a fully validating node needs no storage mining capability at all*.

Goals

Separating an implementation into a blockchain component and one or more mining and market components presents an opportunity to **encourage implementation diversity** while re-using non-security-critical components, and also greatly **ease miner-operator customisations**, even while blockchain node implementation is ongoing.

Goals of this effort include:

- Enhance **network security** by supporting the development of multiple blockchain node implementations, while re-using application-level storage mining, market, and data transfer functionality.
- Support significant **operator customization** by decoupling the components embodying most miner-operator architecture and policy decisions from the core blockchain implementation.
- Support numerous **combinations of node and module implementations** to be combined into a full system by defining clear boundaries and APIs between modules.
- Segregation of core blockchain network endpoints from storage-client deal, upload and retrieval endpoints.
- Flexibility for implementations to link shared modules directly into a monolith, or compose a system of interacting processes.
- Provide an out-of-the-box monolithic go-filecoin build for simple small-scale operations.

Non-goals

- Direct support for **highly scalable and configurable systems**.
- Shared modules written in any language other than Go.
- Re-writing the Rust sectorbuilder in Go.
- Transparent invocation of RPC vs linked modules (though this would be nice)

Design ideas

Modules and processes

A **module** means a code library that may be compiled and linked against. A **component** means a distinct process presenting or consuming RPC API.

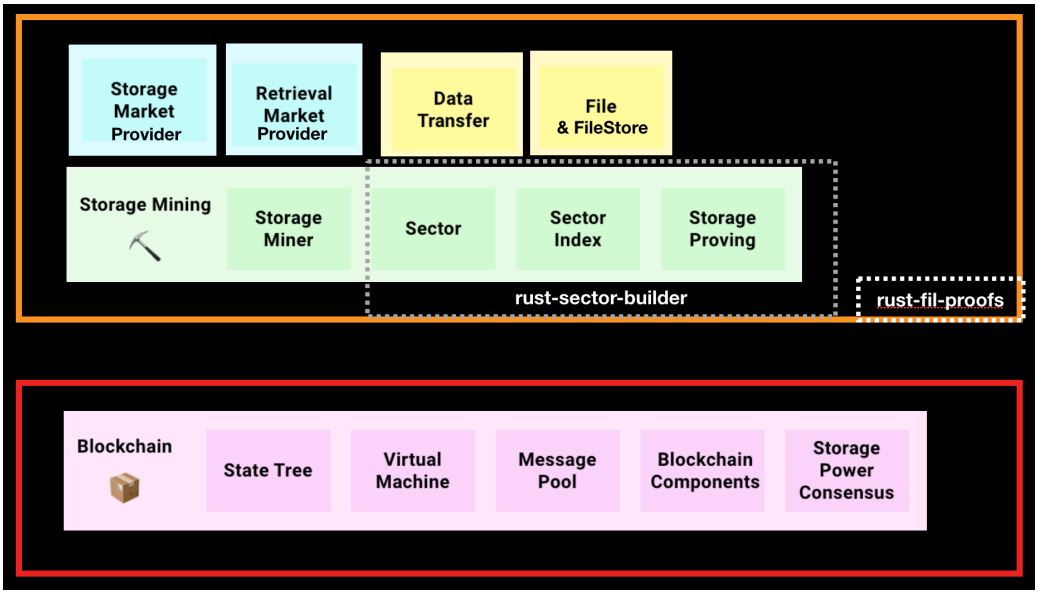
Informed by the spec, we identify the following modules, which may be linked into node binary or other component:

- Storage mining (sector sealing, commitments, storage, and proofs)
- Storage market (storage deal exchange, retrieval negotiation)
- Data transfer (ingestion, retrieval)

These modules may be extracted into one or more shared repositories. In order to be usable by multiple implementations, these ***modules may not depend on go-filecoin*** (or any other node implementation). The types used by these modules must live with the module, and be depended on in turn by node implementations; it may be necessary for some common structures to be duplicated. Modules should be suitable for directly linking in to a Go application. Module APIs must also be suitable for bridging via RPC (no shared client/server resources). Modules may contain state, run threads, and require filesystem access.

The Filecoin specification breaks some of these modules down even further. Reflecting this internal architecture is left to the discretion of implementers.

These modules may then be combined into one or more components, interacting via RPC. For operational simplicity, this document proposes a **single component** containing the storage mining, storage market, and data transfer modules, but a system could further break these components down into smaller ones. A filecoin node implementation may then interact with these modules via the component RPC, rather than direct linking. This permits the component to run on distinct hardware, for multiple components to interact with a single node binary, and similar architectural options.



An engineering team may combine these modules into components in other configurations. For example, go-filecoin will probably link all the modules directly into the node binary in order to provide an “it just works” experience. Another componentization may separate the network-exposed components (storage/retrieval market, data transfer) from the storage mining component.

The filecoin implementation teams intend to create **only the simplest reasonable implementation** of the shared modules. Where multiple policy choices are available, only a single simple one will be implemented. **We expect sophisticated miner operators to fork or re-implement these modules** to support more scalable, distributed architectures. By providing clear and stable APIs between these components, we expect that such customized modules may be integrated into a complete system without much friction.

Storage mining module

The storage mining module (spec) encapsulates everything about sealing and proving storage sectors. It allocates pieces to sectors, seals them, and maintains the physical storage of sectors on disk and associated metadata. The module also schedules PoSt computation, in response to information provided to it about the blockchain state.

A storage mining module instance corresponds to **a single StorageMinerActor**. The storage mining module owns the miner actor's worker's private key. This means it also encapsulates a few small of logic associated with that key: drawing an election ticket, and signing messages and blocks with that key. While logically distinct from the work of proving sector storage, these worker key operations are packaged in this module reflecting the 1:1 relationship with on-chain actors. With this architecture, a single blockchain node may service multiple storage mining modules each servicing different miner actors.

The storage mining module will encapsulate the sectorbuilder, which need no longer be linked into the node implementation (nodes will still link rust-fil-proofs for verification calls).

Design doc: Storage mining module

Storage & retrieval market modules

The storage and retrieval market modules encapsulate order and deal exchange for storage and retrieval deals. In collaboration with the data transfer module, they arrange for piece data to be obtained from storage clients and provided to the storage miner (via a filesystem abstraction), and subsequently made available for retrieval.

Design doc: Storage and retrieval market module

Data transfer module

The data transfer module encapsulates exchange protocols for the exchange of piece data between storage clients and miners, both when consummating a storage deal and when retrieving the piece later.

Design doc: Filecoin data transfer module

Storage component

The storage component bundles the storage mining, market, and data transfer modules into a single binary. This component wires the modules together and contains basic workflow, configuration and policy logic for customising the behaviour of those modules. **Component separation is optional**: a system may instead link all the modules into a single binary (as go-filecoin will likely do).

In a multi-process architecture, the storage component would form the miner operator's entry point for all mining and market operations (but not basic node operations) pertaining to a single storage miner actor. It depends on a node to mediate blockchain interactions. **The storage component drives these interactions**. If viewed as a system of services, the storage component is the consumer of a service provided by a node. Thus, the **storage component will depend on an RPC API be provided by a node**. This API is likely to include streaming operations in order to provide continually changing blockchain state to the component. The mimblewimble/grin project is another example of this multi-process node/miner architecture.

The storage component is likely to need its own CLI (or UI) for operator interaction.

Storage client component

A storage client component bundles the storage market, retrieval market, and data transfer components into an application for a storage client, i.e. a user/process acting as the client of a storage deal.

Open questions

RPC system
Componentization requires a shared RPC facility for interacting components. Depending on detailed component design, this facility will likely require uni- or bi-directional streaming capability.

Systems may use any convenient RPC facility providing the necessary features. Module APIs and RPC need not necessarily be coupled.

REST API
A monolithic system, linking the above modules with the node into a single binary, can present a single REST API for all node and storage operations. Go-filecoin will probably do this.

A multi-process system must do a little more work to present a single API endpoint. Options Include:

- an API service acting as a reverse proxy to a node plus one or more storage components, which routes calls to the node or mining component's "internal" APIs;
- presenting the REST API from the mining component, routing blockchain calls through to the blockchain node to which the mining component is connected.

Repository layout

At one extreme, we could build the shared modules and the storage component in a single repository. This could ease dependency-wrangling and the sharing of code between components.

We could separate any or all of the component and modules into additional repositories. This would draw more clear boundaries and dependencies, but may add friction propagating changes.