ethercattle Documentation

Release 0.0.0

Austin Roberts

Feb 05, 2018

CONTENTS:

CHAPTER

ONE

INTRODUCTION

There is a notion in Systems Administration that services are better when they can be treated as Cattle, rather than Pets. That is to say, when cattle gets badly injured its owner will typically have it slaughtered and replace it with a new animal, but when a pet gets badly injured its owner will typically do everything within reason to nurse the animal back to health. We want services to be easily replaceable, and when a service begins to fail healthcecks we want to discard it and replace it with a healthy instance.

For a service to be treated as cattle, it typically has the following properties:

- It can be load-balanced, and any instance can serve any request as well as any other instance.
- It has simple health checks that can indicate when an instance should be removed from the load balancer pool.
- When a new instance is started it does not start serving requests until it is healthy.
- When a new instance is started it reaches a healthy state quickly.

Unfortunately, existing Ethereum nodes don't fit well into this model:

- Certain API calls are stateful, meaning the same instance must serve multiple successive requests and cannot be transparently replaced.
- There are numerous ways in which an Ethereum node can be unhealthy, some of which are difficult to determine.
  - A node might be unhealthy because it does not have any peers
  - A node might have peers, but still not receive new blocks
  - A node might be starting up, and have yet to reach a healthy state
- When a new instance is started it generally starts serving on RPC immediately, even though it has yet to sync the blockchain. If the load balancer serves request to this instance it will serve outdated information.
- When new instances are started, they must discover peers, download and validate blocks, and update the state trie. This takes hours under the best circumstances, and days under extenuating circumstances.

As a result it is often easier to spend time troubleshooting the problems on a particular instance and get that instance healthy again, rather than replace it with a fresh instance.

The goal of this initiative is to create enhanced open source tooling that will enable DApp developers to treat their Ethereum nodes as replaceable cattle rather than indespensable pets.

CHAPTER

TWO

DESIGN GOALS

The primary goal of the Ether Cattle intiative is to provide access to Ethereum RPC services with minimal operational complexity and cost. Ideally this will be achieved by enhancing an existing Ethereum client with capabilities that simplify the operational challenges.

2.1 Health Checks

A major challenge with existing Ethereum nodes is evaluating the health of an individual node. Generally nodes should be considered healthy if they have the blockchain and state trie at the highest block, and are able to serve RPC requests relating to that state. If a node is more than a couple of blocks behind the network, it should be considered unhealthy.

2.2 Service Initialization

One of the major challenges with treating Ethereum nodes as disposable is the initialization time. Conventionally a new instance must find peers, download the latest blocks from those peers, and validate each transaction in those blocks. Even if the instance is built from a relatively recent snapshot, this can be a bandwidth intensive, computationally intensive, disk intensive, and time consuming process.

In a trustless peer-to-peer system, these steps are unavoidable. Malicious peers could provide incorrect information, so it is necessary to validate all of the information received from untrusted peers. But given several nodes managed by the same operator, it is generally safe for those nodes to trust eachother, allowing us to avoid some of the computationally intensive and disk intensive steps that make the initialization process time consuming.

Ideally node snapshots will be taken periodically, new instances will launch based on the most recent available snapshot, and then sync the blockchain and state trie from trusted peers without having to validate every successive transaction. Assuming relatively recent snapshots are available, this should allow new instances to start up in a matter of minutes rather than hours.

Additionally, during the initialization process services should be identifiable as still initializing and excluded from the load balancer pool. This will avoid nodes serving outdated information during initialization.

2.3 Load Balancing

Given reliable healthchecks and a quick initialization process, one challenge remains on loadbalancing. The Ethereum RPC protocol supports a concept of “filter subscriptions” where a filter is installed on an Ethereum node and subsequent requests about the subscription are served updates about changes matching the filter since the previous request. This requires a stateful session, which depends on having a single Ethereum node serve each successive request relating to a specific subscription.

For now this can be addressed on the client application using Provider Engine's Filter Subprovider. The Filter Subprovider mimics the functionality of installing a filter on a node and requesting updates about the subscription by making a series of stateless calls against the RPC server. Over the long term it might be beneficial to add a shared database that would allow the load balanced RPC nodes to manage filters on the server side instead of the client side, but due to the existence of the Filter Subprovider that is not necessary in the short term.

2.4 Reduced Computational Requirements

As discussed in *Service Initialization*, a collection of nodes managed by a single operator do not have the same trust model amongst themselves as nodes in a fully peer-to-peer system. RPC Nodes can potentially decrease their computational overhead by relying on a subset of the nodes within a group to validate transactions. This would mean that a small portion of nodes would need the computational capacity to validate every transaction, while the remaining nodes would have lower resource requirements to serve RPC requests, allowing flexible scaling and redundancy.

CHAPTER

THREE

APPROACH

3.1 Change Data Capture

After considering several different approaches to meet our *Design Goals*, we settled on a Change Data Capture approach (CDC). The idea is to hook into the database interface on one node, capture all write operations, and write them to a transaction log that can be replayed by other nodes.

3.1.1 Capturing Write Operations

In the Go Ethereum codebase, there is a *Database* interface which must support the following operations:

- Put
- Get
- NewBatch
- Has
- Delete
- Close

and a Batch interface which must support the following operations:

- Put
- Write
- ValueSize

We have created a simple CDC wrapper, which proxies operations to the standard databases supported by Go Ethereum, and records *Put*, *Delete*, and *Batch.Write* operations through a *LogProducer* interface. At present, we have implemented a *KafkaLogProducer* to record write operations to a Kafka topic.

The performance impact to the Go Ethereum server is minimal. The CDC wrapper is light weight, proxying requests to the underlying database with minimal overhead. Writing to the Kafka topic is handled asynchronously, so write operations are unlikely to be delayed substantially due to logging. Read operations be virtually unaffected by the wrapper.

While we have currently implemented a Kafka logger, we have defined an abstract interface that could theoretically support a wide variety of

3.1.2 Replaying Write Operations

We also have a modified Go Ethereum service which uses a *LogConsumer* interface to pull logs from Kafka and replay them into a local LevelDB database. The index of the last written record is also recorded in the database, allowing the service to resume in the event that it is restarted.

Preliminary Implementation

In the current implementation we simply disable peer-to-peer connections on the node and populate the database via Kafka logs. Other than that it functions as a normal Go Ethereum node.

The RPC service in its current state is semi-functional. Many RPC functions default to querying the state trie at the "latest" block. However, which block is deemed to be the "latest" is normally determined by the peer-to-peer service. When a new block comes in it is written to the database, but the hash of the latest block is kept in memory. Without the peer-to-peer service running the service believes that the "latest" block has not updated since the process initialized and read the block out of the database. If RPC functions are called speciying the target block, instead of implicitly asking for the latest block, it will look for that information in the database and serve it correctly.

Despite preliminary successes, there are several potential problems with the current approach. A normal Go Ethereum node, even one lacking peers, assumes that it is responsible for maintaining its database. Occasionally this will lead to replicas attempting to upgrade indexes or prune the state trie. This is problematic because the same operations can be expected to come from the write log of the source node. Thus we need an approach where we can ensure that the read replicas will make no effort to write to their own database.

Proposed Implementation

Go Ethereum offers a standard *Backend* interface, which is used by the RPC interface to retrieve the data needed to offer the standard RPC function calls. Currently there are two main implementations of the standard Backend interface, one for full Ethereum nodes, and one for light Ethereum nodes.

We propose to write a third implementation for replica Ethereum nodes. We believe we can offer the core functionality required by RPC function calls based entirely on the database state, without needing any of the standard syncing capabilities.

Once that backend is developed, we can launch it as a separate service, which will not attempt to do things like database upgrades, and which will not attempt to establish peer-to-peer connections.

Under the hood, it will mostly leverage existing APIs for retrieving information from the database. This should limit our exposure to changes in the database breaking our code unexpectedly.

3.2 Other Models Considered

This section documents several other approaches we considered to achieving our *Design Goals*. This is not required reading for understanding subsequent sections, but may help offer some context for the current design.

3.2.1 Higher Level Change Data Capture

Rather than capturing data as it is written to the database, one option we considered was capturing data as it was written to the State Trie, Blockchain, and Transaction Pool. The advantage of this approach is that the change data capture stream would be reflective of high level operations, and not dependent on low level implementation details regarding how the data gets written to a database. One disadvantage is that it would require more invasive changes to consensus-critical parts of the codebase, creating more room for errors that could effect the network as a whole. Additionally, because those changes would have been made throughout the Go Ethereum codebase it would be harder to maintain if Go Ethereum does not incorporate our changes. The proposed implementation requires very few changes to core Go Ethereum codebase, and primarily leverages APIs that should be relatively easy to maintain compatibility with.

3.2.2 Shared Key Value Store

Before deciding on a change-data-capture replication system, one option we considered was to use a scalable key value store, which could be written to by one Ethereum node and read by many. Some early prototypes were developed under this model, but they all had significant performance limitations when it came to validating blocks. The Ethereum State Trie requires several read operations to retrieve a single piece of information. These read operations are practical when made against a local disk, but latencies become prohibitively large when the state trie is stored on a networked key value store on a remote system. This made it infeasible for an Ethereum node to process transactions at the speeds necessary to keep up with the network.

3.2.3 Extended Peer-To-Peer Model

One option we explored was to add an extended protocol on top of the standard Ethereum peer-to-peer protocol, which would sync the blockchain and state trie from a trusted list of peers without following the rigorous validation procedures. This would have been a substantially more complex protocol than the one we are proposing, and would have put additional strain on the other nodes in the system.

3.2.4 Replica Codebase from Scratch

One option we considered was to use Change Data Capture to record change logs, but write a new system from the ground-up to consume the captured information. Part of the appeal of this approach was that we have developers interested in contributing to the project who don’t have a solid grasp of Go, and the replica could have been developed in a language more accessible to our contributors. The biggest problem with this approach, particularly with the low level CDC, is that we would be tightly coupled to implementation details of how Go Ethereum writes to LevelDB, without having a shared codebase for interpreting that data. A minor change to how Go Ethereum stores data could break our replicas in subtle ways that might not be caught until bad data was served in production.

In the proposed implementation we will depend not only on the underlying data storage schema, but also the code Go Ethereum uses to interpret that data. If Go Ethereum changes their schema *and* changes their code to match while maintaining API compatibility, it should be transparent to the replicas. It is also possible that Go Ethereum changes their APIs in a way that breaks compatibility, but in that case we should find ourselves unable to compile the replica without fixing the dependency, and shouldn’t see surprises on a running systme.

Finally, by building the replica in Go as an extension to the existing Go Ethereum codebase, there is a reasonable chance that we could get the upstream Go Ethereum project to integrate our extensions. It is very unlikely that they would integrate our read replica extensions if the read replica is a separate project written in another language.

CHAPTER

FOUR

IMPLEMENTATION

4.1 Backend Functions To Implement

4.1.1 Downloader()

Downloader()

4.1.2 ProtocolVersion()

ProtocolVersion()

4.1.3 SuggestPrice()

SuggestPrice()

4.1.4 ChainDb()

ChainDb()

4.1.5 EventMux()

EventMux()

4.1.6 AccountManager()

AccountManager()

4.1.7 SetHead()

SetHead()

4.1.8 HeaderByNumber()

HeaderByNumber()

4.1.9 BlockByNumber()

BlockByNumber()

4.1.10 StateAndHeaderByNumber()

StateAndHeaderByNumber()

4.1.11 GetBlock()

GetBlock()

4.1.12 GetReceipts()

GetReceipts()

4.1.13 GetTd()

GetTd()

4.1.14 GetEVM()

GetEVM()

4.1.15 SubscribeChainEvent()

SubscribeChainEvent()

4.1.16 SubscribeChainHeadEvent()

SubscribeChainHeadEvent()

4.1.17 SubscribeChainSideEvent()

SubscribeChainSideEvent()

4.1.18 SendTx()

SendTx()

4.1.19 GetPoolTransactions()

GetPoolTransactions()

4.1.20 GetPoolTransaction()

GetPoolTransaction()

4.1.21 GetPoolNonce()

GetPoolNonce()

4.1.22 Stats()

Stats()

4.1.23 TxPoolContent()

TxPoolContent()

4.1.24 SubscribeTxPreEvent()

SubscribeTxPreEvent()

4.1.25 ChainConfig()

ChainConfig()

4.1.26 CurrentBlock()

CurrentBlock()