ethercattle Documentation

Release 0.0.0

Austin Roberts

Sep 20, 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 discard it 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 If the load balancer serves request to this instance it will serve outdated information.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.

1.1 Publicly Hosted Ethereum RPC Nodes

Many organizations are currently using publicly hosted Ethereum RPC nodes such as Infura. While these services are very helpful, there are several reasons organizations may not wish to depend on third party Ethereum RPC nodes.

First, the Ethereum RPC protocol does not provide enough information to authenticate state data provided by the RPC node. This means that publicly hosted nodes could serve inaccurate information with no way for the client to know. This puts public RPC providers in a position where they could potentially abuse their clients' trust for profit. It also makes them a target for hackers who might wish to serve inaccurate state informatino.

Second, it means that a fundamental part of an organization's system depends on a third party that offers no SLA. RPC hosts like Infura are generally available on a best effort basis, but have been known to have significant outages. And should Infura ever cease operations, consumers of their service would need to rapidly find an alternative provider.

Hosting their own Ethereum nodes is the surest way for an organization to address both of these concerns, but currently has significant operational challenges. We intend to help address the operational challenges so that more organizations can run their own Ethereum nodes.

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 individual nodes 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
- Delete
- Reset
- 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 will 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 messaging systems.

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 specifying 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 service 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

In *go-ethereum/internal/ethapi/backend.go*, a Backend interface is specified. Objects filling this interface can be passed to *ethapi.GetAPIs()* to return *[]rpc.API*, which can be used to serve the Ethereum RPC APIs. Presently there are two implementations of the Backend interface, one for full Ethereum nodes and one for Light Ethereum nodes that depend on the LES protocol.

This project will implement a third backend implementation, which will provide the necessary information to ethapi.GetAPIs() to in turn provide the RPC APIs.

4.1 Backend Functions To Implement

This section explores each of the 26 methods required by the Backend interface. This is an initial pass, and attempts to implement these methods may prove more difficult than described below.

4.1.1 Downloader()

Downloader must return a **go-ethereum/eth/downloader.Downloader* object. Normally the *Downloader* object is responsible for managing relationships with remote peers, and synchronizing the block from remote peers. As our replicas will receive data directly via Kafka, the Downloader object won't see much use. Even so, the *PublicEthereumAPI* struct expects to be able to retrieve a *Downloader* object so that it can provide the *eth_syncing* API call.

If the Backend interface required an interface for a downloader rather than a specific Downloader object, we could stub out at Downloader that provided the *eth_syncing* data based on the current Kafka sync state. Unfortunately the Downloader requires a specific object constructed with the following properties:

- *mode SyncMode* - An integer indicating whether the SyncMode is Fast, Full, or Light. We can probably specify "light" for our purposes.
- *stateDb ethdb.Database* - An interface to LevelDB. Our backend will neeed a Database instance, so this should be easy.
- *mux *event.TypeMux* - Used only for syncing with peers. If we avoid calling Downloader.Synchronize(), it appears this can safely be nil.
- *chain BlockChain* - An object providing the downloader.BlockChain interface. If we only need to support Downloader.Progress(), and we set SyncMode to LightSync, this can be nil.
- *lightchain LightChain* - An object providing the downloader.LightChain interface. If we only need to support Downloader.Progress(), and we set SyncMode to LightSync, we will need to stub this out and provide CurrentHeader() with the correct blocknumber.
- *dropPeer peerDropFn* - Only used when syncing with peers. If we avoid calling Downloader.Synchronize(), this can be *func(string) {}*

Constructing a *Downloader* with the preceding arguments should provide the capabilities we need to offer the *eth_progress* RPC call.

4.1.2 ProtocolVersion()

This just needs to return an integer indicating the protocol version. This tells us what version of the peer-to-peer protocol the Ethereum client is using. As replicas will not use a peer-to-peer protocol, it might make sense for this to be a value like *-1*.

4.1.3 SuggestPrice()

Should return a *big.Int* gas price for a transaction. This can use **go-ethereum/eth/gasprice.Oracle* to provide the same values a stanard Ethereum node would provide. Note, however, that gasprice.Oracle requires a Backend object of its own, so implementing SuggestPrice() will need to wait until the following backend methods have been implemented:

- *HeaderByNumber()*
- *BlockByNumber()*
- *ChainConfig()*

4.1.4 ChainDb()

Our backend will need to be constructed with an *ethdb.Database* object, which will be it's primary source for much of the information about the blockchain and state. This method will return that object.

For replicas, it might be prudent to have a wrapper that provides the *ethdb.Database* interface, but errors on any write operations, as we want to ensure that all write operations to the primary database come from the replication process.

4.1.5 EventMux()

This seem to be used by peer-to-peer systems. I can't find anything in the RPC system that depends on *EventMux()*, so I think we can return *nil* for the Replica backend.

4.1.6 AccountManager()

This returns an **accounts.Manager* object, which manages access to Ethereum wallets and other secret data. This would be used by the Private Ethereum APIs, which our Replicas will not implement. Services that need to manage accounts in conjunction with replica RPC nodes should utilize client side account managers such as Web3 Provider Engine.

In a future phase we may decide to implement an AccountManager service for replica nodes, but this would require serious consideration for how to securely store credentials and share them across the replicas in a cluster.

4.1.7 SetHead()

This is used by the private debug APIs, allowing developers to set the blockchain back to an earlier state in private environments. Replicas should not be able to roll back the blockchain to an earlier state, so this method should be a no-op.

4.1.8 HeaderByNumber()

HeaderByNumber needs to return a **core/types.Header* object corresponding to the specified block number. This will need to get information from the database. It might be possible to leverage in-memory caches to speed up these data lookups, but it must not rely on information normally provided by the peer-to-peer protocol manager.

This should be able to use *core.GetCanonicalHash()* to get the blockhash, then *core.GetHeader()* to get the Block Number.

4.1.9 BlockByNumber()

BlockByNumber needs to return a **core/types.Block* object corresponding to the specified block number. This will need to get information from the database. It might be possible to leverage in-memory caches to speed up these data lookups, but it must not rely on information normally provided by the peer-to-peer protocol manager.

This should be able to use *core.GetCanonicalHash()* to get the blockhash, then *core.GetBlock()* to get the Block Number.

4.1.10 StateAndHeaderByNumber()

Needs to return a **core/state.StateDB* object and a **core/types.Header* object corresponding to the specified block number.

The header can be retrieved with *backend.HeaderByNumber()*. Then the stateDB object can be created with *core/state.New()* given the hash from the retrieved header and the ethdb.Database.

4.1.11 GetBlock()

Needs to return a **core/types.Block* given a *common.Hash*. This should be able to use *core.GetBlockNumber()* to get the block number for the hash, and *core.GetBlock()* to retrieve the **core/types.Block*.

4.1.12 GetReceipts()

Needs to return a *core/types.Receipts* given a *common.Hash*. This should be able to use *core.GetBlockNumber()* to get the block number for the hash, and *core.GetBlockReceipts()* to retrieve the *core/types.Receipts*.

4.1.13 GetTd()

Needs to return a **big.Int* given a *common.Hash*. This should be able to use *core.GetBlockNumber()* to get the block number for the hash, and *core.GetTd()* to retrieve the total difficulty.

4.1.14 GetEVM()

Needs to return a **core/vm.EVM*.

This requires a *core.ChainContext* object, which in turn needs to implement:

- *Engine()* - A conensus engine instance. This should reflect the conensus engine of the server the replica is replicating. This would be Ethash for Mainnet, but may be Clique or eventually Casper for other networks.
- *GetHeader()* - Can proxy *backend.GetHeader()*

Beyond the construction of a new *ChainContext*, this should be comparable to the implementation of eth/api_backend.go's *GetEVM()*

4.1.15 Subscribe Event APIs

The following methods exist as part of the Event Filtering system.

- *SubscribeChainEvent()*
- *SubscribeChainHeadEvent()*
- *SubscribeChainSideEvent()*
- *SubscribeTxPreEvent()*

As discussed in *Load Balancing*, the initial implementation of the replica service will not support the filtering APIs. As such, these methods can be no-ops that simply return *nil*. In the future we may implement these methods, but it will need to be a completely new implementation to support filtering on the cluster instead of individual replicas.

4.1.16 SendTx()

As replica nodes will not have peer-to-peer connections, they will not be able to send transactions to the network via conventional methods. Instead, we propose that the replica will simply queue transactions onto a Kafka topic. Independent from the replica service we can have consumers of the transaction topic emit the transactions to the network using different methods. The scope of implementing *SendTx()* is limited to placing the transaction onto a Kafka topic. Processing those events and emitting them to the network will be discused in *Transaction Emitters*.

4.1.17 Transaction Pool Methods

The transaction pool in Go Ethereum is kept in memory, rather than in the LevelDB database. This means that the primary log stream will not include information about information about unconfirmed transactions. Additionally, the primary APIs that would make use of the transaction pool are the filtering transactions, which we established in *Subscribe Event APIs* will not be supported in the initial implementation.

For the first phase, this project will not implement the transaction pool. In a future phase, depending on demand, we may create a separate log stream for transaction pool data. For the first phase, these methods will return as follows:

- GetPoolTransactions() - Return an empty *types.Transactions* slice.
- GetPoolTransaction() - Return nil
- GetPoolNonce() - Use *statedb.GetNonce* to return the most recent confirmed nonce.
- Stats() - Return 0 transactions pending, 0 transactions queued
- TxPoolContent() - Return empty *map[common.Address]types.Transactions* maps for both pending and queued transactions.

4.1.18 ChainConfig()

The ChainConfig property will likely be provided to the Replica Backend as the backend is contructed, so this will return that value.

4.1.19 CurrentBlock()

This will need to look up the block hash of the latest block from LevelDB, then use that to invoke *backend.GetBlock()* to retrieve the current block.

In the future we may be able to optimize this method by keeping the current block in memory. If we track when the *LatestBlock* key in LevelDB gets updated, we can clear the in-memory cache as updates come in.

4.2 Transaction Emitters

Emitting transactions to the network is a different challenge than replicating the chain for reading, and has different security concerns. As discussed in *SendTx()*, replica nodes will not have peer-to-peer connections for the purpose of broadcasting transactions. Instead, when the *SendTx()* method is called on our backend, it will log the transaction to a Kafka topic for a downstream Transaction Emitter to handle.

Different use cases may have different needs from transaction emitters. On one end of the spectrum, OpenRelay needs replicas strictly for watching for order fills and checking token balances, so no transaction emitters are necessary in the current workflow. Other applications may have high volumes of transactions that need to be emitted.

The basic transaction emitter will watch the Kafka topic for transactions, and make RPC calls to transmit those messages. This leaves organizations with several options for how to transmit those messages to the network. Organizations may choose to:

- Not to run a transaction emitter at all, if their workflows do not generate transactions.
- Run a transaction emitter pointed to the source server that is feeding their replica nodes.
- Run a transaction emitter pointed to a public RPC server such as Infura.
- Run a separate cluster of light nodes for transmitting transactions to the network

4.2.1 Security Considerations

The security concerns relating to emitting transactions are different than the concerns for read operations. One reason for running a private cluster of RPC nodes is that the RPC protocol doesn't enable publicly hosted nodes to prove the authenticity of the data they are serving. To have a trusted source of state data an organization must have trusted Ethereum nodes. When it comes to emitting transactions, the peer-to-peer protocol offers roughly the same assurances that transactions will be emitted to the network as RPC nodes. Thus, some organizations may decide to transmit transactions through APIs like Infura and Etherscan even though they choose not to trust those services for state data.

CHAPTER

FIVE

OPERATIONAL REQUIREMENTS

The implementation discussed in previous sections relates directly to the software changes required to help operationalize Ethereum clients. There are also ongoing operational processes that will be required to maintain a cluster of master / replica nodes.

5.1 Cluster Initialization

Initializing a cluster comprised of a master and one or more replicas requires a few steps.

5.1.1 Master initialization

Before standing up any replicas or configuring the master to send logs to Kafka, the master should be synced with the blockchain. In most circumstances, this should be a typical Geth fast sync with standard garbage collection arguments.

5.1.2 LevelDB Snapshotting

Once the master is synced, the LevelDB directory needs to be snapshotted. This will become the basis of both the subsequent master and the replica servers.

5.1.3 Replication Master Configuration

Once synced and ready for replication, the master needs to be started with the garbage collection mode of "archive". Without the "archive" garbage collection mode, the state trie is kept in memory, and not written to either LevelDB or Kafka immediately. If state data is not written to Kafka immediately, the replicas have only the chain data and cannot do state lookups. The master should also be configured with a Kafka broker and topic for logging write operations.

5.1.4 Replica Configuration

Replicas should be created with a copy of the LevelDB database snapshotted in leveldb-snapshots. When executed, the replica service should be pointed to the same Kafka broker and topic as the master. Any changes written by the master since the LevelDB snapshot will be pulled from Kafka before the Replica starts serving HTTP requests.

5.2 Periodic Replica Snapshots

When new replicas are scaled up, they will connect to Kafka to pull any changes not currently reflected in their local database. The software manages this by storing the Kafka offset of each write operation as it persists to LevelDB, and when a new replica starts up it will replay any write operations more recent than the offset of the last saved operation. However this assumes that Kafka will have the data to resume from that offset, and in practice Kafka periodically discards old data. Without intervention, a new replica will eventually spin up to find that Kafka no longer has the data required for it to resume.

The solution for this is fairly simple. We need to snapshot the replicas more frequently than Kafka fully cycles out data. Each snapshot should reflect the latest data in Kafka at the time the snapshot was taken, and any new replicas created from that snapshot will be able to resume so long as Kafka still has the offset from the time the snapshot was taken.

The mechanisms for taking snapshots will depend on operational infrastructure. The implementation will vary between cloud providers or on-premises infrastructure management tools, and will be up to each team to implement (though we may provide additional documentation and tooling for specific providers).

Administrators should be aware of Kafka's retention period, and be sure that snapshots are taken more frequently than the retention period, leaving enough time to troubleshoot failed snapshots before Kafka runs out

5.3 Periodic Cluster Refreshes

Because replication requires the master to write to LevelDB with a garbage collection mode of "archive", the disk usage for each node of a cluster can grow fairly significantly after the initial sync. When disk usage begins to become a problem, the entire cluster can be refreshed following the *Cluster Initialization* process.

Both clusters can run concurrently while the second cluster is brought up, but it is important that the two clusters use separate LevelDB snapshots and separate Kafka partitions to stay in sync (they can use the same Kafka broker, if it is capable of handling the traffic).

As replicas for the new cluster are spun up, they will only start serving HTTP requests once they are synced with their respective Kafka partition. Assuming your load balancer only attempts to route requests to a service once it has passed health checks, both clusters can co-exist behind the load balancer concurrently.

5.4 Multiple Clusters

Just as multiple clusters can co-exist during a refresh, multiple clusters can co-exist for stability purposes. Within a single cluster, the master server is a single point of failure. If the master gets disconnected from its peers or fails for other reasons, its peers will not get updates and become stale. A new master can be created from the last LevelDB snapshot, but that will take time during which the replicas will be stale.

With multiple clusters, when a master is determined to be unhealthy its replicas could be removed from the load balancer to avoid stale data, and additional clusters could continue to serve current data.