Introduction

This document outlines how to get an Ether Cattle cluster up and running in an infrastructure agnostic manner. If you are interested in running Ether Cattle in Amazon Web Services, we have a CloudFormation template that can get you up and running with minimal effort. If you want to host Ether Cattle in another environment, or want to understand what the CloudFormation template is doing behind the scenes, this document is for you.

System Requirements

This documentation assumes that your Ether Cattle cluster will run on Linux servers. It is likely possible to run Ether Cattle on other operating systems, but that is outside the scope of this document.

Master

vCPU: 2
RAM: 8 GB
Disk: >250 GB SSD

The master needs high performance disks to be able to sync the blockchain. Because of the replication process, the master's disks have higher requirements than a typical disk.

Replicas

vCPU: 1
RAM: 2 GB
Disk: >250 GB HDD

For basic operation, Replicas have very low resource requirements. Synchronizing from the master does not require high performance disks, however heavy RPC request workloads may necessitate faster disks.

Kafka

Cluster: 3 Nodes
vCPU: 1
RAM: 2 GB
Disk: >500 GB HDD

EtherCattle's use of Kafka is not very taxing by Kafka's standards, and so your Kafka cluster's specs do not need to be very high. HDDs are fine, as Kafka performs primarily sequential read / writes. It is possible to host multiple replica clusters on a single Kafka cluster, in which case you may need additional disk space.
Managing Kafka can be somewhat challenging, and its importance for the stability of an EtherCattle cluster cannot be understated. Depending on your organizations needs, you may do well to find a managed Kafka provider.

Snapshotting

Ether Cattle relies on the ability to make volume level snapshots of Geth's chaindata folder. These snapshots need to be atomic, but if necessary that can be achieved by stopping a synced replica. Snapshots need to be taken more frequently than Kafka's retention period (default: 7 days), so that new replicas can spin up with a snapshot, then sync from Kafka.

Depending on your snapshotting solution, you should be aware that volumes freshly created from snapshots may not perform as well as normal volumes. See if your snapshotting solution documents options for "warming" volumes, as this may be necessary when creating a new master from a snapshot.

Basic Installation

Prerequisites

- **Kafka**: This document does not cover the installation of Kafka. Refer to Apache's documentation for Kafka installation: https://kafka.apache.org/quickstart

  Recommended Configuration Options:
  - *default.replication.factor* should be at least 3 to ensure you are tolerant to broker failures.
  - *min.insync.replicas* should be set to at least 2 to avoid loss of data
  - *compression.type* should be *snappy* to avoid using too much disk.

Generating An Initial Snapshot

Before you can stand up the replicas or masters for an EtherCattle cluster, you will need a snapshot of the Geth chaindata folder. For this, you can simply run the Geth client until it is synced, stop the Geth process with `kill -HUP $GethPID` then snapshot the ~/.ethereum folder. Note that the syncing process can take a long time, and even on a highly performant system with lots of peers, you should allow at least 48 hours for this process.

Master Setup

System Configuration

The master should be started with the initial snapshot described in the previous section. You can mount this where you like, but this document will assume it is mounted at `/var/lib/ethereum`

Software Configuration

The master server needs the latest build of the Ether Cattle fork of geth. You can check out the code from Github here: https://github.com/NoteGio/go-ethereum and build with the command for your platform. This is likely one of:

```
make geth-linux-amd64
```

```
make geth-darwin-amd64
```

```
make geth-windows-amd64
```

Preparation

Before starting the master, it is recommended to sync the database from Kafka so that the master will always start syncing from the network at the same point as the replicas. This can be achieved by running:

```
geth replica --kafka.broker=$KafkaHostname --datadir=$DataDir
--kafka.topic=$KafkaTopic --replica.syncshutdown
```

- `--kafka.broker=$KafkaHostname` - Should point to a broker from your Kafka cluster.
- `--kafka.topic=$KafkaTopic` - Should designate the Kafka topic your master will write to, and your replicas will sync from
- `--datadir=$DataDir` - Should correspond to the directory where your geth database snapshot is mounted
- `--replica.syncshutdown` - Tells geth to terminate with a 0 exit code when the database is caught up.

Runtime

The primary process for running a master will sync with the peer-to-peer network and write its change log to Kafka. This can be launched with:

```
geth --gcmode=archive --kafka.broker=$KafkaHostname
--kafka.topic=$KafkaTopic --datadir=$DataDir
```

- `--gcmode=archive` - tells geth to flush the state trie to disk every block. Without this the state trie on the master will exist only in memory, and will not be available to replicas until it periodically gets flushed.
- Other flags are the same as in the previous section

Transaction Broadcasting

Replicas do not connect directly to the peer-to-peer network, thus when a user sends a transaction to the replica via RPC the replica cannot broadcast it directly. Optionally, you can set up a Kafka topic where replicas publish transactions they receive. The `txrelay` service follows this topic and rebroadcasts transactions to the network. If your application is read-only and does not require broadcasting transactions, you may skip this step, otherwise, run the relay with:

```
geth txrelay --kafka.broker=$KafkaHostname --kafka.tx.topic=$TxTopic
--kafka.tx.consumergroup=$ConsumerGroup /var/lib/ethereum/geth.ipc
```

- `--kafka.broker=$KafkaHostname` - Should match the broker used by the master and replicas
- `--kafka.tx.topic=$TxTopic` - Is the topic on which replicas will submit transactions. This can be any string, but must batch between the `txrelay` service and replicas. Note that if you run multiple Ether Cattle clusters, you can use the same topic
- and broadcast transactions through multiple masters to help ensure prompt delivery.
- `--kafka.tx.consumergroup=$ConsumerGroup` - Identifies this instance of txrelay to Kafka, so that if the service gets restarted it does not rebroadcast every transaction on the topic. This can be any string.
- `/var/lib/ethereum/geth.ipc` - The endpoint the txrelay should broadcast transactions to. In this case, we're using the master's IPC endpoint, but this could also be an HTTP(S) based RPC endpoint.

Putting It Together

The following systemd service files will run the master and txrelay services, restarting if they exit in error:

geth.service:

```
[Unit]
Description=Ethereum go client
```

```
After=syslog.target network.target

[Service]
User=geth
Group=geth
Environment=HOME=/var/lib/ethereum
Type=simple
ExecStartPre=geth replica --kafka.broker=${KafkaHostname}
--datadir=/var/lib/ethereum --kafka.topic=${KafkaTopic}
--replica.syncshutdown
ExecStart=geth ${MasterExtraFlags} --gcmode=archive
--kafka.broker=${KafkaHostname} --datadir=/var/lib/ethereum
--kafka.topic=${KafkaTopic}
KillMode=process
KillSignal=SIGINT
TimeoutStartSec=86400
TimeoutStopSec=90
Restart=on-failure
RestartSec=10s

[Install]
WantedBy=multi-user.target
```

Geth-txrelay.service:

```
[Unit]
Description=Ethereum go client transaction relay
After=syslog.target network.target geth

[Service]
User=geth
Group=geth
Environment=HOME=/var/lib/ethereum
Type=simple
ExecStart=geth txrelay --kafka.broker=${KafkaHostname}
--kafka.tx.topic=${NetworkId}-tx
--kafka.tx.consumergroup=${KafkaTopic}-cg /var/lib/ethereum/geth.ipc
KillMode=process
KillSignal=SIGINT
TimeoutStopSec=90
Restart=on-failure
RestartSec=10s
```

```
[Install]
WantedBy=multi-user.target
```

Firewall

Master

The Master must enable the following ingress for the public internet:

- 30303/udp
- 30303/tcp
- 30301/udp

It must allow public egress on UDP and TCP for syncing with the blockchain, and must be able to contact every server in your Kafka cluster.

Replica

The Replica does not need any connectivity to the public internet for its normal operation. It requires:

- 8545/tcp - Only needs to be reachable by the load balancer

It must also be able to contact every server in your Kafka cluster.

Kafka

Kafka does not need any connectivity to the public internet for normal operation. It requires:

- 9092/tcp - Must be reachable by master, replicas, and other Kafka brokers

Replica Setup

System Configuration

The replica should be started with the initial snapshot described earlier. You can mount this where you like, but this document will assume it is mounted at `/var/lib/ethereum`. It is important that for the initial setup, the same snapshot is used for both the master and replicas. When building or rebuilding systems after initialization, any snapshot can be used so long as it is within the retention period of Kafka (default 7 days).

Runtime

The replica process will connect with Kafka, syncing data from the master and serving RPC requests. This can be launched with:

```
geth replica --kafka.broker=$KafkaHostname --datadir=$DataDir
--kafka.topic=$KafkaTopic --kafka.tx.topic=$TxTopic
--replica.startup.age=45 --replica.offset.age=62
--replica.block.age=240
```

- `--kafka.broker=$KafkaHostname` - Should point to a broker from your Kafka cluster.
- `--kafka.topic=$KafkaTopic` - Should designate the Kafka topic your master will write to, and your replicas will sync from
- `--datadir=$DataDir` - Should correspond to the directory where your geth database snapshot is mounted
- `--kafka.tx.topic=$TxTopic` - Should match the topic used with the master's txrelay, if applicable (otherwise this flag may be omitted).
- `--replica.startup.age=45` - When the replica begins syncing, it will not start serving RPC requests until it is up-to-date with Kafka, and has a block younger than the number of seconds specified here. Setting to 0 or omitting the flag disables this limit, and the replica will start serving RPC requests as soon as it is in sync with Kafka. Do not set this value too low, as the time it takes to mine a block and propagate it through the network means that a replica may rarely see a block that is only a few seconds old.
- `--replica.offset.age=62` - If the replica has not received any communication from the master via Kafka in this amount of time, shut down. The master sends a heartbeat message every 30 seconds, the recommended value of this flag indicates at least two missed heartbeats. Setting to 0 or omitting this flag disables shutting down in the event of missed heartbeats. This flag should only be used if:
    - You have multiple clusters, and other replicas will be able to pick up the load if your master goes down, and
    - You are using a process manager such as systemd or docker, which will restart the process after it shuts down to resume the synchronization process.
- `--replica.block.age=240` - If the replica's most recent block is older than this number of seconds, shutdown. This should be significantly higher than `--repilca.offset.age`. Do not set this value too low, or normal deviations in block time could cause your replicas to shutdown. This is in place to protect against a master that is sending heartbeats, but failing to sync with the network. This flag should only be used if:
    - You have multiple clusters, and other replicas will be able to pick up the load if your master goes down, and

    - You are using a process manager such as systemd or docker, which will restart the process after it shuts down to resume the synchronization process.

Snapshotting Process

As noted earlier, in order to run a cluster effectively, your infrastructure must support snapshotting of hard disks. In order to prepare a snapshot, we recommend the following process:

1. Start a new server containing the most recent snapshot and the Ether Cattle fork of Geth.
2. Run `geth replica --kafka.broker=$KafkaHostname --datadir=$DataDir --kafka.topic=$KafkaTopic --replica.syncshutdown` (as described above) to sync the latest data from Kafka on top of the last snapshot.
3. Take a snapshot of your $DataDir with your snapshotting system.
4. Shutdown the server you launched to create the snapshot.
5. In the future when you stand up new instances (either replicas or masters) start them with the most recent snapshot. You do not need to replace existing servers - they have all the information contained in the snapshot.

We recommend snapshotting every 24 hours. This keeps you well within Kafka's retention period, and also reduces the amount of time required to sync a new server from Kafka. You will also want to make sure you have a process in place to clean up old snapshots. We recommend keeping at least 4 days worth of snapshots, so that if a problem is found with a recent snapshot you can revert to an older one.

Monitoring

To maximize the availability of your cluster, there are several aspects that should be monitored

Master

- **Disk Utilization**: Monitoring disk utilization on the master is essential. Both masters and replicas will increase in disk utilization over time, and will eventually need to be increased. We recommend alarms at 95% disk utilization, which should be a couple days warning based on typical growth patterns. As the Master and all Replicas use disk at effectively the same rate, all generally need to be updated at the same time.
- **Memory**: Memory usage on the master should be fairly constant. If you have 8 GB of RAM and the configuration described above, the master should stay consistently below 75% RAM utilization. If it exceeds this threshold for an extended period, we recommend rebooting your master server.

- **CPU**: CPU utilization on the master will be a bit spikey - when new blocks come in it will ramp up to validate the block, then settle back down. With two modern CPUs, the CPU utilization will be around 60% averaged out over time. We recommend monitoring CPU utilization on the master, and if it regularly exceeds 85% consider adding additional CPUs.
- **Peer Count**: A critical metric for master health is its number of peers. If a master has no peers, it will not receive new block information. By default, a master will establish connections with 25 peers. If the master drops below 10 peers, this could be a sign of a problem. Check firewall rules as described above. If a master drops to zero peers and stops syncing blocks, we recommend restarting the master.

  The following Shell command can be used to retrieve the master's current peer count (requires netcat and jq packages installed):

  ```
  echo '{"jsonrpc":"2.0", "method":"net_peerCount", "params":[], "id":64}' | nc -U /var/lib/ethereum/geth.ipc | jq -cr '"ibase=16;obase=A;" + (.result | ltrimstr("0x") | ascii_upcase)' | bc
  ```
- **Block Age**: Every time it processes a block (or group of block, during the initial sync) geth logs a message about the processed blocks. If the processed block is older than 60 seconds, Geth will include the age of that block. We recommend tracking this, and if block times fall consistently behind check the health of the master. High block times can occur under normal circumstances due to the variable nature of the mining process, but consistently high block times is likely indicative of a problem. If you are seeing consistently high block times on a master that was in sync with the network, check network connectivity, peer counts, and if everything else looks okay, restart the master.
- **Block Number**: Because block age metrics are only available when blocks are greater than 60 seconds old, we also recommend monitoring how frequently new blocks are reported. If no new blocks have been received in 120 seconds, follow the same process you would if you were seeing high block ages reported.

Replicas

- **Disk Utilization**: Both masters and replicas will increase in disk utilization over time, and will eventually need to be increased. We recommend alarms at 95% disk utilization, which should be a couple days warning based on typical growth patterns. As the Master and all Replicas use disk at effectively the same rate, all generally need to be updated at the same time.
- **Memory**: Replica RAM utilization depends primarily on the volume of RPC traffic the replica is serving. A replica that is serving only a handful of requests will use less than 1 GB of RAM. A replica under very heavy load will trend up with the load. We recommend setting an alarm at 80% RAM utilization, and scaling out your cluster of replicas to help share the RPC traffic.

- **CPU**: Replica CPU utilization depends primarily on the volume of RPC traffic the replica is serving. A replica that is serving only a handful of requests will have a nearly idle CPU. A replica under heavy load will trend up with the load. We recommend setting an alarm at 75% CPU utilization and scaling out your cluster of replicas to help share the RPC traffic.
- **Block Number**: Every 30 seconds, the replica will log a message with several pieces of information, including the latest block number. If this number does not increase regularly, there is likely a problem with either the master, or communication between the master and replica.
- **Block Age**: Every 30 seconds, the replica will log a message with several pieces of information, including the latest block age. If the block age exceeds `--replica.block.age` the replica will shutdown, so you may want to monitor and be alerted before that happens.
- **Block Lag**: Block lag is a computed metric found by comparing

  ```
  master.blockNumber - replica.blockNumber
  ```

  If this number is higher than 2, that might indicate that the replica is not receiving information from the master. If it stays high or increases steadily, check that Kafka is functioning properly, and try restarting the replica.
- **Replica Offset Age**: Every 30 seconds, the replica will log a message with several pieces of information, including how long it has been since it last got a message from the master. If this number exceeds `--replica.offset.age` the replica will shutdown, so you may want to monitor and be alerted before that happens. Generally this happens if either the master has crashed or Kafka has become unavailable.

Kafka

- **Disk Utilization**: A single Ether Cattle topic with snappy compression enabled and a default 7 day retention period will use about 40 GB of RAM on an ongoing basis. If you are running multiple Ether Cattle clusters on the same Kafka cluster, this will scale accordingly. We recommend leaving plenty of buffer between your expected usage and your disk limits, and monitoring closely to ensure you don't run out of disk.
- **Memory**: Kafka memory utilization is dependent on the volume of data in Kafka topics and the number of active subscribers. With two Ether Cattle clusters, each with two subscribers, we see around 1.5 GB of RAM utilization. We recommend setting an alarm on memory utilization around 85%, and if that threshold is exceeded for an extended period, increase the RAM for your Kafka servers.
- **CPU**: The CPU utilization for Kafka in an Ether Cattle cluster is exceptionally low. We recommend tracking it and scaling appropriately if it is consistently high, but this is unlikely to be a bottleneck for your cluster.

High Availability

A single Ether Cattle cluster provides several operational benefits over running conventional Ethereum nodes, but the master server is still a single point of failure. Using data stored in Kafka, the master can recover much more quickly than a node that needed to sync from peers, but that can still lead to a period of time where the replicas are serving stale data.

To achieve high availability requires multiple clusters with independent masters and their own replicas. Multiple replica clusters can share a high-availability Kafka cluster. The following formula can be used to determine the statistical availability of a cluster:

$$a = 1 - (1 - \frac{mtbf}{mttr + mtbf})^N$$

Where:

- *mtbf* - Mean Time Between Failures - The average amount of time between failures of a master server
- *mttr* - Mean Time To Recovery - The average amount of time it takes to replace a master server after a failure
- *N* - The number of independently operating clusters

The values of *mtbf* and *mttr* will depend on your operational environment. With our AWS CloudFormation templates, we have established an *mttr* of 45 minutes when snapshotting daily. We have not gathered enough data to establish a *mtbf*, but with two independent clusters and a 45 minute *mttr*, EC2's regional SLA becomes the bounding factor of availability if the *mtbf* is greater than two weeks.

This formula focuses only on the availability of masters - it assumes that each master has multiple independent replicas. If a master only has a single replica, that will hurt the *mtbf* of the cluster as a whole.

Upgrade Process

Upgrading an Ether Cattle cluster should be handled carefully to avoid inconsistencies between the master and replica servers.

We recommend an immutable server approach for both masters and replicas. Rather than updating the server in place, stand up a new server, let it sync, and then take down the old server. Note that you should not have two masters running for the same cluster at the same time, so this may involve some downtime.

When upgrading geth, we recommend standing up a whole new cluster with a new Kafka topic. Using the most recent snapshot from your old cluster, stand up a new set of masters and replicas with the new version of geth, communicating over a new topic. Once the new replicas are synced and joined to your load balancer, you can take down the old replicas. This approach avoids any service downtime.

Disk Considerations

Because the master server must run in archival mode, its disk utilization grows faster than a typical geth server - generally around 25 GB per week. To keep disk consumption from getting out of hand, you may want to periodically rerun the **Getting an Initial Snapshot** process and creating a new cluster from that snapshot. This should get you back to the base level of disk usage for a fast synced node, which, at the time of this writing, is around 200 GB.