neonCLUSTER Notes

# Introduction

This document is a bucket for information that may make it into more formal documentation later.

# Deployment

Docker 1.12.0 and later allows swarm mode services to be deployed on both manager and worker nodes. This is reasonable for small clusters, but we recommend that for larger and sensitive production clusters that most workloads be constrained to deploy only to worker nodes. This limits the potential for cluster services to overload manager nodes and potentially disrupt the entire cluster.

# Networks, Proxies, and Routes

neonCLUSTER hosts reserve ports 5000-5499 for internal cluster purposes, including: for services like the neon-registry-cache, the logging Elasticsearch cluster, as well as the neon-proxy-public and neon-proxy-private services described below. The port definitions are defined in the NeonClusterPorts class.

neonCLUSTERs deploy two proxy services that can be used to route TCP and HTTP/S traffic to services and containers attached to the built-in neon-public and neon-private networks. These proxies are based on the [neoncluster/neon-proxy](https://hub.docker.com/r/neoncluster/neon-proxy/) image which relies on [HAProxy](http://www.haproxy.org).

The neon-proxy-public proxy is intended primarily for routing inbound Internet traffic to services and containers on the neon-public network.

neon-proxy-private is intended for routing internal cluster traffic or perhaps internal traffic between datacenters to the neon-private network.

These proxies are used to augment the built-in Docker service discovery and routing capabilities which doesn’t provide for routing external traffic from a router or load balancer to Docker services and also for providing load balancing for stateful services implemented as a group of individual (non-service) containers.

Proxies are configured using the neon-cli by managing routes and TLS certificates via the route and cert commands. The built-in [neon-proxy-manager](https://hub.docker.com/r/neoncluster/neon-proxy-manager/) service watches for these changes and rebuilds the proxy configurations as required.

Note that network, proxy, and route names prefixed by neon-\* are reserved for neonCLUSTER use.

# Docker Secrets

As of version 1.13.0, Docker supports secrets for swarm mode services. Docker secrets are created by piping the secret (text or data) to the docker secret NAME command. This persists the secret in Docker using the NAME passed. The necessary secrets must be made available to cluster services as they are deployed.

Secret names prefixed by neon-\* are reserved for neonCLUSTER services. Here are the current neonCLUSTER secrets:

neon-proxy-manager-credentials Vault credentials for the neon-proxy-manager service.

neon-proxy-public-credentials Vault credentials for the neon-proxy-public service.

neon-proxy-private-credentials Vault credentials for the neon-proxy-private service.

# Vault

neonCLUSTER uses HashiCorp Vault for secret management. This is deployed to the cluster manager nodes and are configured for high-availability. The Vault unseal keys and root token are persisted anywhere in the cluster but instead, are encrypted and persisted on the operator’s workstation by the **neon-cli**.

## Default Vault Configuration

Vault is deployed initially with:

* **Auditing** to **syslog** is enabled using the **AUTH** facility and the **vault** tag.
* **Transit** backend is enabled to provide encryption as a service.
* A secret backend is mounted at **neon-secret** and is reserved for use by neonCLUSTER.
* The standard **/secret** backend is available for general use.
* The **AppRole** authentication backend is enabled.
* neonCLUSTER specific roles and policies are defined with names prefixed by **neon-\***.

## Naming Conventions

We recommend that you adhere to these naming conventions:

* The **neon-secret** secret backend is reserved for neonCLUSTER related services.
* Role, policy, transit or other Vault item names prefixed by **neon-\*** are reserved for use by neonCLUSTER related uses.
* To avoid conflicts, applications and services should prefix their Vault item names using a DNS host name you control (replacing dots with dashes). For example, the owner of the foo.com domain could:  
  + Store secrets at: **secret/foo-com/my-secrets**
  + Name an AppRole: **foo-com-myrole**
  + Name a policy: **foo-com-mypolicy**

## neonCLUSTER Secrets Hierarchy

neonCLUSTER secrets are persisted under the **/neon-secret** mount point using the follow structure.

neon-secret

global  
 dblogin: <credentials>

cert  
 mycert1: <cert + private key PEM encoded>  
 mycert2: <cert + private key PEM encoded>  
  
 hosting  
 options: <cluster hosting options>

service  
 neon-foo  
 foo-secret1  
 foo-secret2  
 ...  
 neon-bar  
 bar-secret1  
 bar-secret2  
 ...  
 vpn  
 ca.zip.encrypted (zipped/encrypted certificate authority files)

Global secrets, like shared API keys or database credentials are saved under **neon-secret/global**.

Named TLS certificates are saved under **neon-secret/cert** with the value being the certificate concatenated with the private key, both PEM encoded.

Service specific secrets are saved under **neon-secret/service** using the service name. The hierarchy beneath this is defined by the service.

## neonCLUSTER Policies

neonCLUSTER policy names are prefixed by **neon-\***. Here’s the current list of policies initialized for a cluster:

**neon-reader** Can read/list any secret under **neon-secret/\***

**neon-reader** Can read/list any secret under **neon-secret/\***

**neon-cert-reader** Can read/list certificates.

**neon-cert-writer** Can read/write/list certificates.

**neon-hosting-reader** Can read/list cluster hosting information/credentials.

**neon-hosting-writer** Can read/write/list cluster hosting information/credentials.

**neon-service-reader** Can read/list any secret under **neon-secret/service/\***

**neon-service-writer** Can read/write/list secrets under **neon-secret/service/\***

**neon-global-reader** Can read/list any secret under **neon-secret/global/\***

**neon-global-writer** Can read/write/list secrets under **neon-secret/global/\***

TODO: Should we define policies for specific services or just set these explicitly in the AppRole when the service is deployed?

## neonCLUSTER AppRoles

neonCLUSTER role names are prefixed by **neon-\*** and typically match the name of the corresponding service. These roles are currently created when a cluster is provisioned:

**neon-proxy-public** Granted the **neon-cert-reader** policy to access the cluster certificates.

**neon-proxy-private** Granted the **neon-cert-reader** policy to access the cluster certificates.

# Consul

neonCLUSTER uses HashiCorp Consul as its primary key/value store. Consul servers are deployed to the cluster manager nodes and are configured for high-availability. Consul proxy agents are deployed to all worker nodes[[1]](#footnote-1) such that host processes and containers can always query the local proxy to access Consul.

## Layout

neonCLUSTER reserves the use of all keys prefixed by **neon\***.

We recommend that user-deployed applications and services prefix their keys using a DNS domain you control for example, the owner of the **foo.com** domain could prefix keys for the **bar** service like: **foo.com/bar/\***. Potential conflicts will be avoided by following this convention.

## Calling Consul

Consul can be accessed directly from each host using the CLI tool. The **CONSUL\_HTTP\_ADDR** environment variable and the local hosts file are configured such that no additional parameters are required. These settings can be propagated to containers and services that mount the host **/etc/neoncluster/env-host** script file and then execute it during startup so that the CLI can be used when installed into the container.

Other tools like **curl** may be used to query Consul. These tools should use the **CONSUL\_HTTP\_FULLADDR** environment variable because it includes the URL scheme (http:// or https://).

## neonCLUSTER Consul Hierarchy

neonCLUSTER reserves the use of all **neon\*** key prefixes. Here’s structure beneath:

neon:  
 cluster:  
 definition.deflate – (json/compressed) the current cluster definition  
 definition.hash - MD5 hash of the definition (base64)

lock:

event:  
 cert-changed

service  
 neon-proxy-public  
 ...  
 neon-proxy-private  
 ...  
 neon-proxy-manager  
 ...

Non-sensitive cluster information that needs to be shared across multiple services will be persisted in keys under **cluster**. Keys used for global locks or semaphores will persisted under **lock**, and keys used to signal global events will be persisted under **event**.

Service specific keys will be located under **service/<service name>**.

# Logging

neonCLUSTER relies on TD-Agent (from [Fluentd](http://www.fluentd.org)), Elasticsearch, and Kibana for its logging infrastructure. We choose TD-Agent for log capture rather than Logstash because TD-Agent was much lighter weight then Logstash which required the Java runtime.

neonCLUSTER deploys several components to pull this all together:



The neon-log-host image is deployed as a local container on every cluster node (both managers and workers). It’s responsible for reading events from the systemd journal and receiving local syslog events as well as logs from local Docker containers configured to use the fluent log driver. The neon-log-host container filters out logs from itself, adds some datacenter, cluster, and node properties to each event and forwards the events on to the neon-log-collector service for further processing and storage

The neon-log-collector service is responsible for receiving raw events from the hosts, normalizing them into a standard format, and then persisting them to Elasticsearch.

## Host and Docker Container Statistics

neonCLUSTER uses Elastic [Metricbeat](https://www.elastic.co/products/beats/metricbeat) to capture Docker host node as well as Docker container statistics such as CPU, memory, disk I/O, etc. This is accomplished by deploying the [neoncluster/metricbeat](https://hub.docker.com/r/neoncluster/metricbeat/) image as a container on all cluster nodes.

## Log Record Formats

neonCLUSTERs persist four basic types of log events to the log Elasticsearch cluster: service, proxy, vault-audit, and Metricbeat. Service events capture typical status events emitted by services. These are persisted using the standard ELK stack Logstash format. Proxy events are emitted by HAProxy to describe network traffic handled by the cluster’s public, private, and Vault network proxies. Metricbeat events describe the status of a Docker host node or container at a point in time.

The following subsections describe the fields for the service and proxy events. You can examine the Elastic Metricbeat documentation for more information on those events.

### Service Event Fields

This section describes the standard neonCLUSTER fields for service events. The log pipeline is responsible for transforming raw received events into this format.

activity\_id Optional globally unique activity ID.

audit Optional field added for audit related events (see below).

cluster Identifies the source cluster within the datacenter.

container\_id Short container ID or empty if not a container.

datacenter Identifies the source datacenter.

environment Identifies the operational environment, one of: dev, test, stage, prod, or other.

index Optional long index of the generated event since the service was started. This is useful for displaying events in the actual order they were emitted by services that support this.

json Original raw string for event messages formatted as structured JSON.

level Event level/priority. See the description below for the possible values.

location Optional field that describes a related physical location. See the section below for more information.

message Optional event message text.

module Identifies the source component or application area.

node Docker host node name as specified in the cluster definition.

node\_dnsname Docker host node DNS name (if any).

node\_ip Docker host node’s IP address on the local cluster network.

node\_role Docker host node role, one of: manager or worker.

proxy Optional field added for proxy traffic events (see below).

service Names the service or container.

service\_host Identifies the how the service is hosted: system, docker,…

service\_type Identifies the underlying service type, e.g. elasticsearch, couchbase, mssql,… This may be the same as the service field in situations where the application code is single use.

tag The Fluentd event tag at the time when the event was persisted to Elasticsearch.

@timestamp Unix time (seconds) when the event was emitted or captured.

The possible level field values are inspired by the standard [syslog levels](https://en.wikipedia.org/wiki/Syslog#Severity_level):

emergency System is unusable (not emitted by services).

alert System is seriously degraded (not emitted by services)..

critical Service has failed (maps from fatal for Log4Net oriented application logging).

error Service has encountered an error.

warn Indicates that an error may occur if actions are not taken.

notice Something unusual has occurred but is not an error.

info Normal operational messages that require no action.

debug Developer/diagnostic information.

unknown Log level could not be determined.

### Location Details

Some events may be associated with a physical location. A common example are proxy traffic events where a reverse geographical lookup has been performed on the client IP address.

Note that geoname\_id fields refer to places in the [geonames.org](http://www.geonames.org/about.html) database.

The possible location fields:

latitude Latitude coordinate.

longitude Longitude coordinate.

metro\_code (USA only). [Nielson Designated Market Area ID](https://support.google.com/richmedia/answer/2745487?hl=en).

postal\_code Postal code.

time\_zone Timezone name.

continent Continent related fields.

code Two character continent code.

geoname\_id Name ID.

name Continent name.

country Country related fields.

geoname\_id Name ID.

iso\_code Three character ISO code.

name Country name.

city City related fields.

geoname\_id Name ID.

name City name.

subdivisions Array of regional subdivisions between country and city (e.g. State, County,…)

geoname\_id Name ID.

iso\_code ISO code.

name Subdivision name.

### Proxy Traffic Details

neonCLUSTERs deploy three proxy services: neon-proxy-vault, neon-proxy-public and neon-proxy-private. These services are based on HAProxy and handle TCP and HTTP load balancing for situations where it’s not possible to use the Docker ingress network. The logging pipeline recognizes log events from these services and adds the proxy field with traffic statistics to the event.

Common Fields for TCP and HTTP Traffic

Here are the proxy common subfields across TCP and HTTP traffic:

browser Optional field describing known browsers.

bot Indicates whether the browser is a known bot (true/false).

device Device name.

name Name of a known browser.

os\_name Operating system (platform) name.

os\_version Operating system (platform) version.

version Version of a known browsers.

bytes\_received Number of bytes received by the proxy from the client.

bytes\_sent Number of bytes sent by the proxy to the client.

client\_ip Client IP address.

conn\_proxy Total number of connections being handled by the proxy at the time the event was logged.

conn\_frontend Number of requests handled by the proxy frontend at the time the event was logged.

conn\_backend Number of proxy connections to internal HAProxy backend at the time the event was logged.

conn\_server Number of proxy connections to backend servers at the time the event was logged.

mode Identifies the type of proxy: tcp or http.

queue\_server Number of operations queued to be delivered to the current backend server at the time the event was logged.

queue\_backend Number of operations queued to the current proxy across all backend servers at the time the event was logged.

retries Number of times the operation was retried after an initial failure.

route Identifies the neonCLUSTER proxy route.

server Identifies the backend server. This can be configured in the route or will auto generated by the neon-proxy-manager.

server\_ip Backend server IP address.

server\_port Backend server port.

time\_queue Total seconds the operation waited in all queues.

time\_connect Seconds the operation waited for a connection to a backend server.

time\_session Seconds for the overall session.

termination Details how the session was terminated. See the HAProxy documentation [here](http://cbonte.github.io/haproxy-dconv/1.7/configuration.html#8.5) for more information.

tls\_version TLS version or the empty string if the connection is not secured.

tls\_cypher TLS cypher or the empty string if the connection is not secured.

Extended Fields for HTTP Traffic

Here are the extended proxy common subfields for and HTTP traffic:

http\_host HTTP Host header.

http\_method HTTP request method, like: GET, PUT, POST,…

http\_status HTTP response status code.

http\_time\_active Seconds from the request time (@timestamp) until the response was transmitted.

http\_time\_idle Seconds waiting idle for the first byte of the HTTP.

http\_time\_request Seconds to receive the full HTTP request from the first byte.

http\_time\_response Seconds the backend server took to process the request and return the response line and headers. This does not include the time to return all of the response contents.

http\_uri Relative request URI excluding query strings.

http\_uri\_query Request URI query string.

http\_user\_agent HTTP User-Agent header.

http\_version The HTTP version, like: HTTP/1.1

### Audit Details

TODO: DESCRIBE THESE

cluster Identifies the source cluster within the datacenter.

datacenter Identifies the source datacenter.

details Vault audit event details.

environment Identifies the operational environment, one of: dev, test, stage, prod, or other.

node Docker host node name as specified in the cluster definition.

node\_dnsname Docker host node DNS name (if any).

node\_ip Docker host node’s IP address on the local cluster network.

node\_role Docker host node role, one of: manager or worker.

service Names the service or container.

tag The Fluentd event tag at the time when the event was persisted to Elasticsearch.

@timestamp Unix time (seconds) when the event was emitted or captured.

## Custom Event Processing

By default, neonCLUSTER log pipeline attempts to extract some fields such as the timestamp, log level, module, and the remaining message from log events passing through the pipeline. This can work for many applications, but sometimes logs require custom processing.

To accomplish this, you’ll need to tag the application events with a log source identifier as the leading tag segment. The typical case will be to specify --log-driver=fluentd and then --log-opt tag=<value> as one of these strings when deploying a Docker service or container. This gives the neon-log-collector service enough information to customize event parsing for the specific application.

Here are the predefined identifiers supported by neonCLUSTER out of the box. You may specify custom tags and then extend the neon-log-collector image to support other applications.

neon-common Many NeonResearch applications emit a common log message format that include an optional timestamp, optional log-level, and optional sections in that order, followed by the message text. Each of the optional sections are surrounded by square brackets and are formatted like:  
  
[2017-01-27T19:04:11.000+00:00] [info] [module:main] [activity:<guid>] The message.

elasticsearch Elasticsearch cluster node.

# Activity Correlation

It is often useful to be able to correlate information about the actions performed across multiple systems and services to perform a high-level activity. neonCLUSTER handles this via globally unique activity IDs. By default, the neonCLUSTER HAProxy services will add an X-Activity-ID header to each proxied request, if this header isn’t already present. Standard services will be coded such that activity IDs from received requests will be forwarded on to downstream service calls and also included in the service logs.

Activity IDs are to be considered to be opaque strings. IDs generated by HAProxy for received requests without an activity ID will look like:  
  
C140001:C1B8\_AC140014:2327\_55F1BFF3\_0003

IDs generated by neonCLUSTERs related services will have .NET standard GUIDs like:  
  
2A45FAE8-9902-4F8B-BF58-0E4785D7519F

Other services may generate activity IDs with differing formats, so you should never depend on a specific format.

# Cluster Networks

neonCLUSTER assumes that the cluster servers are deployed on a network behind a firewall and/or router. The services will be assigned static IP addresses that cannot be changed after the cluster is deployed.

The neon tool configures and controls the servers by connecting directly to cluster servers using SSH/SCP and issuing commands. It also used HTTP/S to perform operations against various servers. This implies that the tool must have access to the cluster network. This can be achieved by:

1. Running the tool from a workstation on the cluster network.
2. Deploying a custom VPN solution.
3. Using the Built-in neonCLUSTER Management VPN.

Solution #1 works well for development and test environments, for very secure production environments where physical access is required or where a [jumpbox](https://en.wikipedia.org/wiki/Jump_server) is available for operators to log into remotely to administer the cluster. This will be problematic for other environments such as public clouds, where direct connection is impossible and jumping is inconvenient.

Solution #2 is certainly possible, effective, and secure but can be difficult and expensive to configure, especially for smaller environments. This can be accomplished using dedicated hardware and software as well as VPN services provided by the cloud environments.

Solution #3 is the easiest way to get started. This uses [OpenVPN](https://openvpn.net/) to establish a point-to-point VPN connection between the operator workstation and the cluster and the neon tool handles all of the provisioning!

## neonCLUSTER VPN

For cloud deployments, the neon tool will need to configure a VPN for cluster management purposes. This is not currently supported for private data center deployments, but will be in the future). This is based on [OpenVPN](https://openvpn.net/) and is intended to allow cluster operators to connect to the cluster network to perform operations there. This is not intended to serve as a general-purpose site-to-site VPN (although I suppose, it could be used that way in a pinch).

The management VPN can be enabled at cluster deployment time by setting the cluster configuration property:

vpn.enabled = true

The VPN servers are deployed to the cluster manager nodes. This implies that these nodes must have IP addresses that are reachable from the operator’s network. For cloud environments, the **neon** tool configures a load balancer with NAT rules to route VPN traffic through to the servers.

# Cluster DNS

neonCLUSTER includes a programmable DNS server that provides the following capabilities:

* Dynamically register and lookup DNS names
* Endpoint health checking
* Cluster node specific name resolution.

Let’s examine these capabilities one at a time. First, custom A records can be provisioned in the neonCLUSTER DNS to reference services hosted inside or outside the cluster. By convention, the “.cluster” top-level domain is used to identify generic services within the cluster but the DNS also supports other host names. Multiple destination IP addresses can be registered for a given host.

Health checking can be enabled for host names. This means that neonCLUSTER DNS will periodically perform an HTTP or TCP connection check for each A record registered for a host and then only return health IP addresses. The DNS can also be configured to randomly select the address returned as a primitive form of load balancing.

Some scenarios require that a docker container be able to resolve the IP address of the cluster host node. Specific examples include:

* Reference the local Consul agent
* Reference the local neon-proxy-public and neon-proxy-private load balancers
* Reference the cluster ingress/mesh network

Note: Early releases of neonCLUSTER mapped the /etc/neoncluster/env-host script into containers that would add entries to the container’s /etc/hosts file to support these scenarios. This worked but was kind of a hack, especially for user containers. That also meant that we’d have to manage host files across all of the cluster nodes which would become messy over time.

The neonCLUSTER DNS is implemented using [PowerDNS](http://powerdns.com) [server](https://doc.powerdns.com/authoritative/index.html) and [recursor](https://doc.powerdns.com/recursor/) combined with the neon-dns and neon-dns-health Docker services. Here’s how this looks:



Here’s a brief summary of the components:

PowerDNS Server Open source DNS programmable DNS server that is able to use pluggable backends to resolve host names using custom code. neonCLUSTER uses the remote backend to submit queries to the neon-dns service via HTTP.

PowerDNS Recursor Open source DNS recursor and cache. This handles forwarding lookups to external DNS servers.

Consul Cluster Consul K/V store. Consul servers are provisioned on the manager nodes in a high availability configuration and Consul clients are provisioned on all worker nodes. neonCLUSTER DNS used Consul to persist the custom DNS hosts as well as endpoint health status.

neon-dns Docker service that accepts DNS lookups from PowerDNS Server via HTTP and answers these by querying Consul. This runs as a container exposing its HTTP interface on port 5007 on each cluster node host network.

neon-dns-health Docker service that performs periodic health checks on registered endpoints, updating the status in Consul. Generally, only one instance of this service should be running at any given time.

Here’s how this fits together:

* Custom DNS records are persisted to Consul via the neon-cli or other mechanism.
* DNS records may enable health checks.
* PowerDNS server, recursor, and the neon-dns service are provisioned on all cluster nodes.
* Consul is provisioned on all nodes with the manager nodes acting as servers and the workers as clients.
* The local PowerDNS server instances are configured as the DNS on each node.
* Docker is configured by default such that containers will query the local PowerDNS server.
* PowerDNS is configured to use the remote backend to submit HTTP queries to neon-dns.
* neon-dns returns answers by querying the local Consul instance which may query the Consul leader.
* Names that cannot be resolved by neon-dns will be forwarded to the PowerDNS recursor that uses external DNS servers (Google by default) to resolve the request.
* The neon-dns-health Docker service will be configured to run a single instance to periodically perform health checks on DNS endpoints, updating Consul.

This architecture is pretty clean. Each cluster node essentially acts as a standalone DNS service using Consul to manage DNS state. Here’s the flow:

1. DNS requests from local node processes as well as Docker containers will hit PowerDNS server running on the host node.
2. PowerDNS server queries neon-dns which may return a response from its cache or query Consul.
3. Valid answers from neon-dns will be returned to the caller.
4. When neon-dns doesn’t have an answer, PowerDNS server will forward the request to the recursor who will query the outside world.
5. neon-dns-health runs in the background testing endpoint availability and updating Consul.

Note that external servers may configure the cluster managers as their DNS servers to participate in this scheme.

1. This may change in the future. We may deploy an HAProxy instance to use the Docker ingress network instead. [↑](#footnote-ref-1)