ECS with NGINX (OpenResty)

Deployment Reference Guide

August 2017

A Dell EMC Whitepaper

Revisions

Date	Description
August 2017	Initial release

Table of Contents

Revisions....2
Executive Summary....5
1 Introduction....6
1.1 Audience....6
1.2 Scope....6
2 ECS Overview....7
3 NGINX Overview....9
3.1 NGINX Core....9
3.2 OpenResty....9
4 ECS with NGINX Deployments....11
4.1 Single....11
4.2 Highly Available (Redundancy)....12
4.3 Global Load Balancing....12
5 Single Deployment Example....14
5.1 Installation and Setup....14
5.1.1 Server....14
5.1.2 OpenResty....15
5.1.3 Domain Name System (DNS)....20
5.2 OpenResty Configuration....22
5.2.1 Global and Event Context....22
5.2.2 HTTP Context....23
5.2.3 Stream Context....39
5.2.4 Monitoring....43
6 Highly Available Example....51
6.1 Virtual IP....51
6.2 Keepalived....52
6.3 Validation....55
7 Global Load Balancing Example....57
7.1 Setup....57
7.2 OpenResty Configuration Files....59
7.2.1 Geo-Pinning (gslb.lua)....62
7.2.2 Monitoring....62

7.3 Validation 66
8 Best Practices 67
9 Conclusions 68
A Attachment – NGINX-SAMPLE.ZIP 69
A.1 Install 70
A.2 SLB 70
A.3 Keepalived 70
A.4 GSLB 70
B Troubleshooting 71
B.1 Build Issues 71
B.2 Logs 71
B.3 Firewall Issues 71
C ECS Dedicated Cloud 72
D Technical Support and Resources 73
D.1 Related resources 73

Executive Summary

Elastic Cloud Storage (ECS) is the third generation object platform from Dell EMC. It is designed for traditional and next-generation applications. It is known for its flexible deployment, resiliency and simplicity. ECS does not have a specific requirement for a particular load balancer. However, in an optimal configuration, a load balancer is recommended to distribute the load across the nodes within ECS and ECS clusters in different geographic locations. NGINX (OpenResty) is a free and reliable open source web server software solution that offers load balancing and other proxy features such as mail, HTTP, and TCP/UDP. It provides a cost-effective solution for customers who are interested in utilizing a load balancer with ECS.

1 Introduction

This white paper is a reference guide into deploying NGINX load balancer with Dell EMC Elastic Cloud Storage (ECS). It provides example configurations and highlights best practices when utilizing the load balancing feature of the NGINX (OpenResty) software solution with ECS.

1.1 Audience

This document is targeted for customers and Dell EMC personnel interested in a reference deployment of ECS with NGINX (OpenResty) web platform.

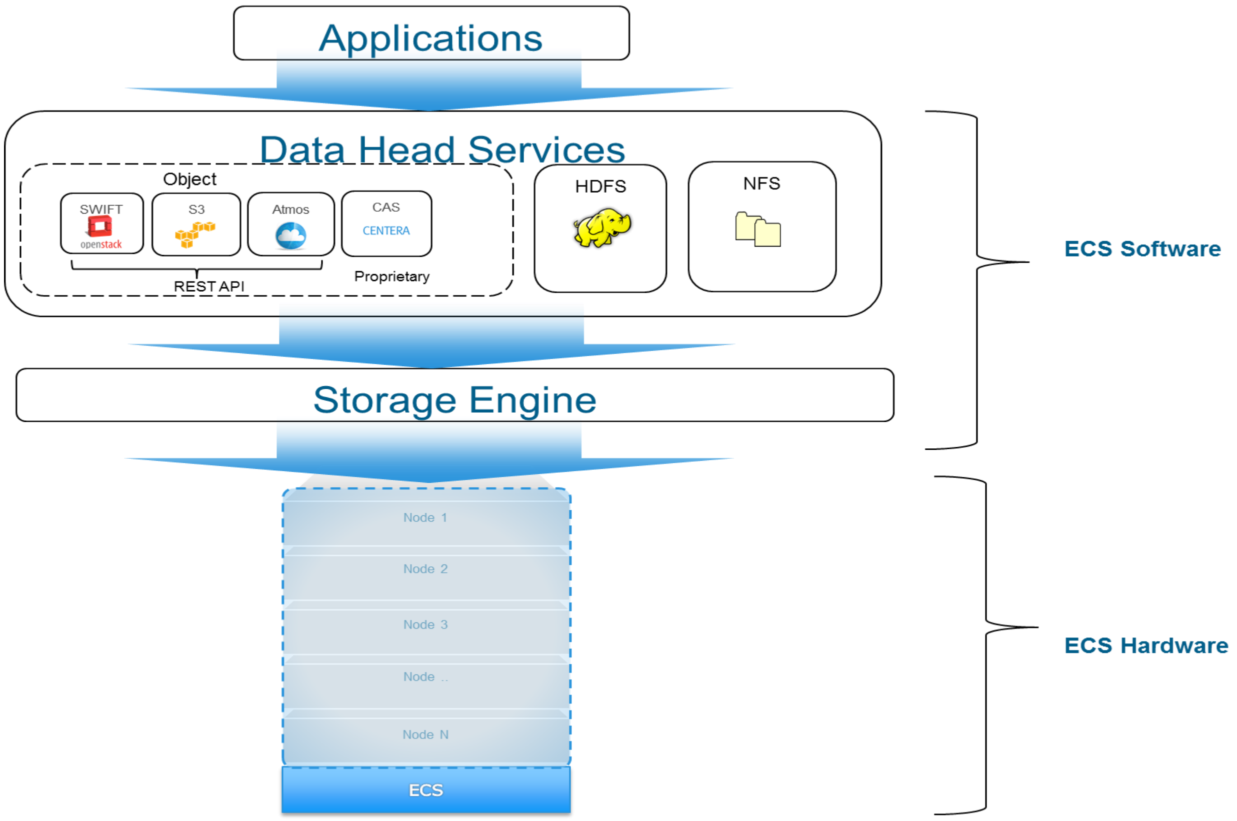
1.2 Scope

This whitepaper is meant to be a reference deployment guide for customers who would like to use NGINX with their ECS deployment. Its intent is to provide a "reference" or an example for customers to deploy ECS with NGINX. Load balancing is optional and not part of ECS. A quick overview of ECS is covered in this whitepaper. For a more in-depth overview of ECS Architecture and description of how ECS handles and processes object and file access, refer to ECS Architecture and Overview whitepaper.

2 ECS Overview

ECS is a consortium of software, hardware nodes with disks and switches working together seamlessly to provide access to object storage data. ECS provides object and file storage. Object access via S3, Atmos, and Swift on ECS storage platform is achieved via REST APIs. Objects are written, retrieved, updated and deleted via HTTP or HTTPS calls of GET, POST, PUT, DELETE, and HEAD. For file access, ECS provides NFS version 3 natively. Applications accessing ECS send requests to the ECS data head services that are responsible for taking client requests, extracting required information, and passing it to the storage engine and hardware for further processing (e.g. read, write, etc.) as shown in Figure 1.

Figure 1 - ECS High Level View



Each of the supported protocols communicates to ECS via specified ports as highlighted in Table 1. ECS also supports CAS protocol; however, a load balancer is not required since the Centera SDK has a built-in load balancer.

Table 1 - Ports assignments per ECS protocol.

ECS Protocol	Transport Protocol or Daemon Service	Port
S3	HTTP	9020
	HTTPS	9021

Atmos	HTTP	9022
	HTTPS	9023
Swift	HTTP	9024
	HTTPS	9025
NFS	mountd,nfsd	2049
	portmap	111
	lockd	10000

These ports are important when configuring ECS with NGINX and need to be open in your firewall in order to access objects using the above protocols. For more information on ECS ports refer to the ECS Security Configuration Guide.

3 NGINX Overview

NGINX is a free open source web server software solution having numerous features such as HTTP and reverse proxy, mail proxy, TCP/UDP proxy and load balancing capabilities. It is well known for its scalability and flexible configuration. Binaries and source can be downloaded from the NGINX site (https://nginx.org/). NGINX has been tested on several platforms and operating systems such as Linux, Solaris, FreeBSD, Windows and AIX.

With the growth of the NGINX developer community, numerous add-on modules have been created to give NGINX additional functionality. One such open-source community that has branched out from NGINX core is OpenResty® (https://openresty.org/). OpenResty is an open-source full web platform that is based on NGINX with added support for Lua, extended modules and other libraries developed by the community. The example deployments defined in this whitepaper utilizes the OpenResty variant of NGINX customized for use with ECS.

3.1 NGINX Core

Utilizing NGINX load balancing features allows requests to be distributed to several ECS systems to enhance performance, scalability and reliability. NGINX core has support for the following:

- Load Balancing Modes
  - Layer 7 (http) - evaluate the HTTP headers and forward to backend servers based on content of user request.
  - Layer 4 (tcp) - allows all data traffic to be forwarded directly to backend servers streamlining user requests.
- Load Balancing Algorithms – options available include:
  - Round-Robin – default algorithm which selects servers in a rotating basis.
  - Least Connected– selects servers based on the least number of connections
  - IP-Hash – selects servers based on a hash of the source IP such as the user IP address to ensure request goes to the same server until something changes in the hash (i.e. one backend server goes down) .
- Weighted load balancing – assigning weight values to particular servers such that requests are distributed based on the weight assigned. For instance, a server with an assigned weight of 3 handles three times the requests than the other servers.
- Sticky Sessions – enables persistence in order for applications to connect to same backend server to process requests.
- Health Check – simple checks to validate if a backend is available and if not, then it is automatically removed from the rotation to process requests until it is restored or becomes healthy.
- Monitoring –simple statistics relating to NGINX are available for monitoring purposes but are limited.

3.2 OpenResty

OpenResty (https://openresty.org) is an open source web platform which bundles together NGINX core, LuaJIT (Just-In-Time Lua Compiler), Lua libraries, and other external libraries. It extends the core functionality of NGINX core via the Lua (lightweight scripting language) modules and other third party NGINX modules to provide developers a mechanism to customize a web server. Some of these modules are static such that a re-compile of NGINX is required in order to utilize them.

Included with OpenResty is a LuaJIT compiler and Lua libraries. Lua is a scripting programming language mostly used for embedded systems and clients. The main designs goal of Lua is to provide a simple and flexible mechanism to extend features and functionality. It runs on all types of operating systems (Unix and Windows), platforms from mobile to mainframes, and embedded processors (ie. ARM, Rabbit). There are samples in this whitepaper which exemplify the use of Lua scripts to optimize reads and writes for ECS in a global environment, conduct health checks on ECS, and collect metrics.

4 ECS with NGINX Deployments

NGINX can be deployed in a single, highly available or global fashion. In all deployments, NGINX main configuration file, *nginx.conf*, contains the directives on where to forward the requests, health checks, and how to distribute the requests across the servers. The configuration file is organized in a modular way in which blocks of definitions or directives are encompassed in a set of braces { }. Definitions within these braces are referred to as “contexts”. The configuration files can be simple in which a simple context or multiple nested contexts are specified or embedded Lua scripts are used for more complex processing.

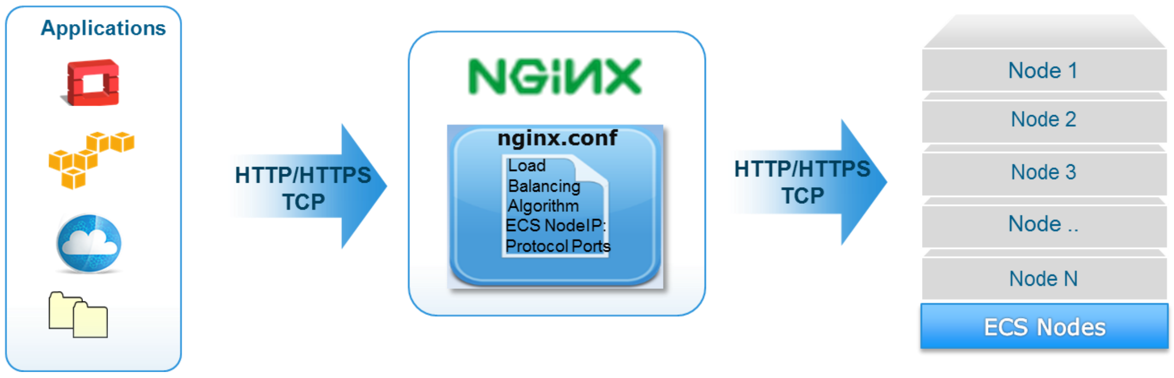
The ***main*** context contains details on how HTTP requests and TCP requests are handled and forwarded to a pool of backend servers in addition to any health checking and monitoring. If the ***http*** context is defined, the HTTP headers are analyzed and forwarded based on the content of request. If context is ***stream***, requests are forwarded directly to pool of backend servers for handling. Within the http or stream context is a ***server*** context that defines the port NGINX is listening to, a load balancing algorithm to distribute requests among the resources, as well as health checks or other directives on how to process requests. As previously mentioned, NGINX provides round-robin, least connect and ip-hash balancing algorithms. Either a domain name system (DNS) names or virtual IPs of NGINX web servers are presented to clients.

The example images of NGINX with ECS deployments in this section highlight object and file protocols access. Objects are accessed via HTTP/HTTPS and NFS via TCP. For NFS, it is recommended that a load balancer be used for high availability purposes only and not for balancing load across the ECS nodes. More detailed information on how to employ NGINX with ECS when using NFS is described in a later section of this whitepaper.

4.1 Single

In a single NGINX deployment, the *http* and *stream* contexts defines an *upstream* context that specifies the IP address of the ECS nodes with specified ports of ECS protocol (e.g. 10.246.50.129:9020) and load balancing algorithm. Figure 2 illustrates an example of a single deployment. This is the simplest of configurations; however, the single load balancer is also a single point of failure and not recommended in production environments.

Figure 2 - Example of ECS with a Single NGINX



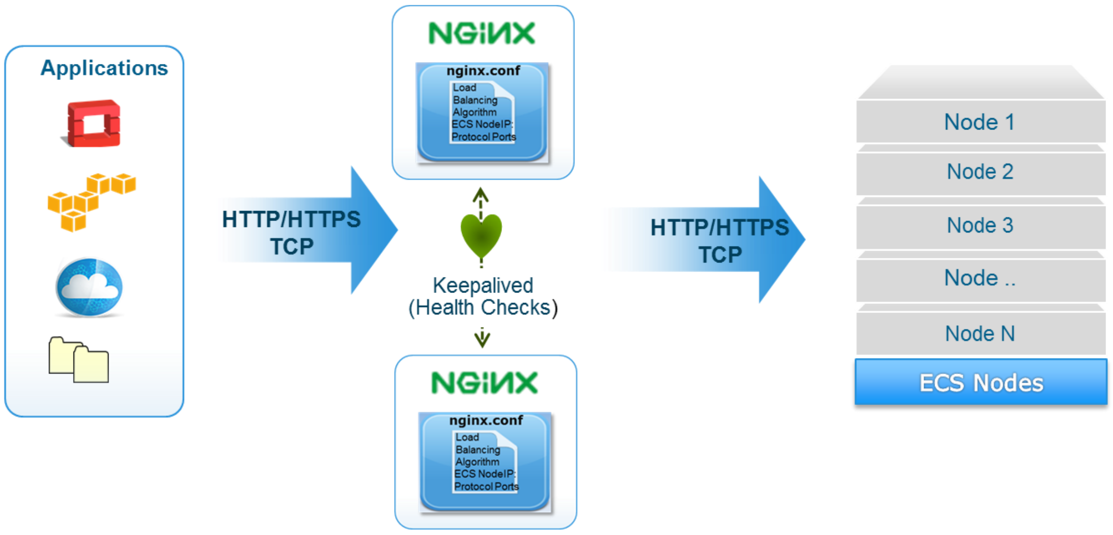
4.2 Highly Available (Redundancy)

In order to not have a single point of failure, it is best practice to setup NGINX in a “highly available” configuration by setting up two NGINX web servers. This provides redundancy such that in case of failure another load balancer is available to handle requests. A mechanism such as Linux’s “keepalived” utility can do health checks between the load balancers to inform NGINX when one of the web servers is not available. In a highly available setup, there are two ways to configure the redundant NGINX:

- **Active/Passive** – one NGINX act as a primary and the other is only activated when the primary fails or is unresponsive.
- **Active/Active** – both NGINX are active and either one can process client requests.

There are advantages and disadvantages of each method that should be considered. Since both load balancers are available for use in the active/active setup, the performance level is higher than in an active/passive setup. However, in an active/passive, there is a consistent performance level when one fails whereas in active/active when one fails, performance level can drop by as much as half as perceived by clients. If certain “levels of service” are expected, then consistency is an important criterion to consider. Upsizing the servers hosting the NGINX in an active/passive setup can improve performance; however, it may not be as cost-effective. Understanding the tradeoffs and requirements is important in developing a deployment best suited for your needs. Figure 3 provides an example of redundant NGINX web server in front of a pool of ECS Nodes with a virtual IP presented to clients.

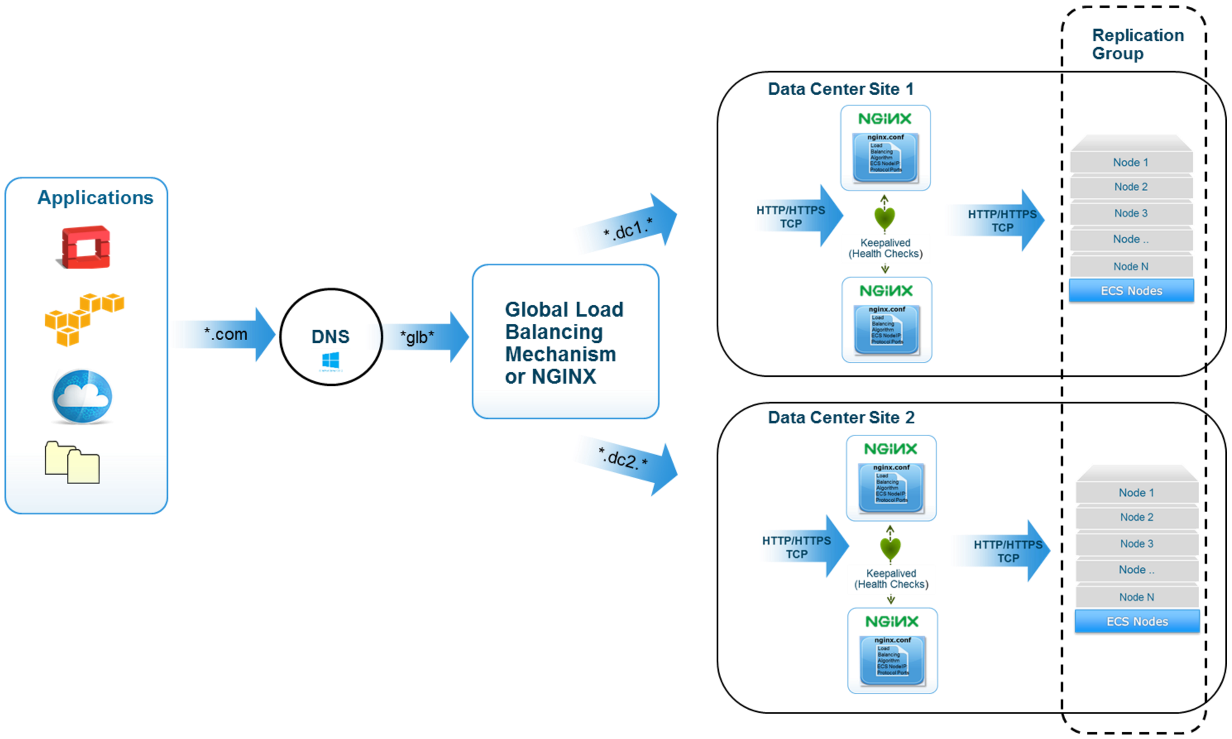
Figure 3 - Example of ECS with Redundant NGINX for High Availability



4.3 Global Load Balancing

When there are two or more geographically dispersed ECS sites within a replication group, a mechanism to load balance across the nodes between sites is recommended. This is especially pertinent in three or more sites where it becomes key for taking advantage of ECS storage efficiency achieved via ECS XOR feature. Another advantage is when one site is unavailable; requests are automatically forwarded to the surviving site(s), providing disaster recovery and high availability. Global load balancing can be achieved by either using DNS, network routing (i.e. OSPF: Open Shortest Path First, BGP: Border Gateway Protocol, etc), a global server load balancer (GLSB) or combination of these techniques. Figure 4 provides an example of client requests being sent to a Domain Name System (DNS) which have an entry for a global load balancing mechanism or NGINX. The global load balancing techniques used forwards requests to a pool of NGINX web servers which then forwards requests to a pool of ECS nodes within a replication group.

Figure 4 - Example of Global Load Balancing with NGINX in a Geo-Replicated ECS Deployment



When considering a global load balancing mechanism, it is important to understand that a read to ECS involves checking with the owner of the object to validate if it has the latest copy locally. If data is not local or in the site cache, it retrieves the data from the site that has the latest version. Thus when architecting a global load balancing solution, it is advised to send or direct the read requests to the owning site if possible. This may depend on the workflow and application.

5 Single Deployment Example

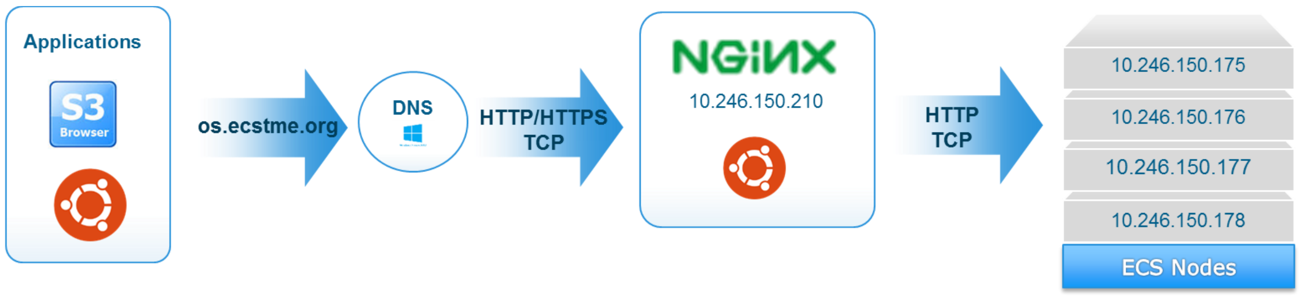
An example of how to setup ECS with OpenResty is detailed in the following sections. It provides a base setup in which the reader can enhance the deployment to implement further. In this example, information on how to configure OpenResty for load balancing object (S3 and Swift) access and NFS access in a single deployment are described. For object access, http context is used and for NFS stream context is used. Virtual machines running Ubuntu 16.04, was used as the server to host OpenResty. DNS installed on a Windows Server 2012 was used to map the OpenResty IP addresses to various names.

To customize OpenResty for ECS deployment, sources for OpenResty are downloaded and compiled with additional modules. Sample Lua scripts to conduct ECS health checks and provide ECS metrics are described and utilized in this example. In the global deployment of OpenResty, a sample “geo-pinning” Lua script has been developed and is described in the Global Load Balancing Example section.

5.1 Installation and Setup

Components configured in this example to employ ECS with OpenResty include a server to host OpenResty running Ubuntu 16.04 LTS, OpenResty (source, NGINX core, libraries, LuaJIT,and additional modules), Domain Name System (DNS) and an ECS U300 appliance with ECS 3.0HF1 installed. Ubuntu and Windows client servers were used to validate OpenResty with ECS deployment for both object and file access. S3 Browser for Windows and Linux curl was used to validate the setup of OpenResty with ECS for object access and a regular Linux mount command was used to test NFS. Figure 5 illustrates the components in this example.

Figure 5 - Example of Single NGINX Deployment Setup with ECS



5.1.1 Server

A physical server (bare metal) or a virtual machine can be used for deployment of OpenResty. The server should be sized (CPU, memory, network cards, etc.) based upon the following criteria:

- Workload or amount of traffic expected
- If using physical server or virtual machines
- Deploying multiple instances of OpenResty in active/passive or active/active mode.
- Expected service level agreements

A UNIX or Windows operating system is also installed on the server or virtual machine. Refer to the specific operating system website to get minimum server requirements. OpenResty can also be encapsulated using container technology such as Docker for flexibility and automation in deployment.

5.1.2 OpenResty

As mentioned, OpenResty is an open-source web platform that bundles NGINX core, LuaJIT, and other modules and libraries contributed by the developer community. For this example, certain versions of the OpenResty components and a build (compilation) from the sources are required in order to utilize third-party modules for ECS health checks, monitoring and metrics collection. Tables 2 provides the components, versions and download locations of OpenResty, modules, and Linux libraries and tools used in this sample deployment.

Table 2 - Versions and Download Locations of OpenResty Components, Third Party Modules, and Libraries

Type	Components	Download Location
OpenResty	OpenResty 1.11.2.2	https://openresty.org/download/openresty-1.11.2.2.tar.gz
	NGINX 1.11.5	http://nginx.org/download/nginx-1.11.5.tar.gz
	LUARocks 2.3.0	http://luarocks.org/releases/luarocks-2.3.0.tar.gz
	OpenSSL 1.0.2j	https://www.openssl.org/source/openssl-1.0.2j.targ.gz
	PCRE 8.39	https://ftp.csx.cam.ac.uk/pub/software/programming/pcre/pcre-8.39.tar.gz
3rd Party Modules for Monitoring	nginx-module-sts	https://github.com/vozlt/nginx-module-sts/archive/master.zip
	nginx-module-stream-sts	https://github.com/vozlt/nginx-module-stream-sts/archive/master.zip
	nginx-module-vts	https://github.com/vozlt/nginx-module-vts/archive/master.zip
3rd Party Modules for Health Checks	ngx_stream_upstream_check_module	https://github.com/lusis/ngx_stream_upstream_check_module/archive/master.zip
Libraries and Tools	build-essential ca-certificates curl libgd-dev libgeoip-dev libncurses5-dev libperl-dev libreadline-dev libxslt1-dev make perl unzip zlib1g-dev wget apt-utils	Installed from Ubuntu Repositories (apt-get install)

Lua Scripts	prometheus.lua healthcheck.lua config.lua strutils.lua s3/urllib.lua	Developed by Dell EMC DevOps team and attached to this whitepaper (nginx-sample.zip)

The basic steps to install OpenResty include:

1. Install Ubuntu libraries and tools.
2. Download and unzip 3rd party modules
3. Download and unzip OpenResty and sources
4. Build OpenResty with the add-on third party modules.
5. Install OpenResty

Several libraries and tools are required to setup and build OpenResty. Figure 6 provides a sample command to install these libraries and tools.

Figure 6 - Install Ubuntu Libraries and Tools

```
# apt-get install -y --no-install-recommends \
                   build-essential \
                   ca-certificates \
                   curl \
                   libgd-dev \
                   libgeoip-dev \
                   libncurses5-dev \
                   libperl-dev \
                   libreadline-dev \
                   libxslt1-dev \
                   make \
                   perl \
                   unzip \
                   zlib1g-dev \
                   iproute \
                   iptables \
                   wget \
                   apt-utils
```

OpenResty has limited built in modules related to health checks, monitoring, and metric collection (read latency, write latency, etc). Several modules from the Github repositories are available to provide this additional functionality and include:

- **nginx-module-sts-master and nginx-module-stream-sts-master** – provides stream server traffic status information such as requests per second, responses, and total traffic received and sent. If stream context is not defined, no information will be shown.
- **nginx-module-vts-master –** provides HTTP server traffic status information.
- **ngx_stream_upstream_check_module** – provides support for health checks in NGINX streams.

Figure 7 shows sample commands to download and unzip these modules. The normal directory location to place these modules is in “/usr/local/src”.

Figure 7- Step 2 Download and Unzip 3rd Party Add-on Modules

```
# cd /usr/local/src

# wget -O nginx-module-sts-master.zip https://github.com/vozlt/nginx-module-
sts/archive/master.zip

# unzip nginx-module-sts-master.zip

# wget -O nginx-module-stream-sts-master.zip https://github.com/vozlt/nginx-module-stream-
sts/archive/master.zip

# unzip nginx-module-stream-sts-master.zip

# wget -O nginx-module-vts-master.zip https://github.com/vozlt/nginx-module-
vts/archive/master.zip

# unzip nginx-module-vts-master.zip

# wget -O ngx_stream_upstream_check_module.zip
https://github.com/lusis/ngx_stream_upstream_check_module/archive/master.zip

# unzip ngx_stream_upstream_check_module.zip
```

To verify if you have downloaded correctly, Figure 8 has the sizes of the files after download. Depending on versions or release downloaded, these sizes may differ.

Figure 8 - Sizes of Third Party Modules Downloaded.

```
# ls -l *.zip
-rw-r--r-- 1 root root  30266 Jun  1 10:49 nginx-module-stream-sts-master.zip
-rw-r--r-- 1 root root 357012 Jun  1 10:51 nginx-module-sts-master.zip
-rw-r--r-- 1 root root 396273 Jun  1 10:29 nginx-module-vts-master.zip
-rw-r--r-- 1 root root 154616 Jun  1 10:29 ngx_stream_upstream_check_module.zip
```

Additional definitions are needed for the *ngx_stream_upstream_check_module-master* module so a patch is executed to modify the modules with these new definitions. This patch is attached to this whitepaper. Figure 9 shows an example of how this patch is applied and output when applied.

Figure 9 - Patching of ngx_stream_upstream_check_module-master

```
# cd ngx_stream_upstream_check_module-master
# patch -p1 < /tmp/stream_upstream_module.patch
patching file ngx_stream_upstream_check_module.c
patching file ngx_stream_upstream_check_module.h
```

The OpenResty bundled sources in the main website does not include certain versions of NGINX, Lua, PCRE (Perl Compatible Regular Expression), and OpenSSL components that are needed for some of the Lua scripts and third party add-on modules. Figure 10 illustrates how to replace the bundled version of OpenResty with the necessary versions needed for this sample deployment.

Figure 10 - Step 3 Download and unzip of OpenResty and Components

```
# cd /tmp

# curl -kfSL https://www.openssl.org/source/openssl-1.0.2j.tar.gz -o openssl-1.0.2j.tar.gz

# tar xzf openssl-1.0.2j.tar.gz

# curl -kfSL https://ftp.csx.cam.ac.uk/pub/software/programming/pcre/pcre-8.39.tar.gz -o
pcre-8.39.tar.gz

# tar xzf  pcre-8.39.tar.gz

# curl -kfSL https://openresty.org/download/openresty-1.11.2.2.tar.gz -o openresty-
1.11.2.2.tar.gz

# tar xzf openresty-1.11.2.2.tar.gz

# curl -kfSL http://nginx.org/download/nginx-1.11.5.tar.gz -o nginx-1.11.5.tar.gz

# tar xzf nginx-1.11.5.tar.gz
```

Check for failures or errors during download and when unpacking the tarballs of the OpenResty components. To verify if you have downloaded correctly, Figure 11 has the sizes of the files after download. Depending on versions or release downloaded, these sizes may differ.

Figure 11 - Sizes of Downloaded OpenResty Components.

```
# ls -l /tmp/*.tar.gz
-rw-r--r-- 1 root root  956517 Jun  1 10:34 nginx-1.11.5.tar.gz
-rw-r--r-- 1 root root 4244348 Jun  1 10:34 openresty-1.11.2.2.tar.gz
-rw-r--r-- 1 root root 5307912 Jun  1 10:34 openssl-1.0.2j.tar.gz
-rw-r--r-- 1 root root 2062258 Jun  1 10:34 pcre-8.39.tar.gz
```

For the proper functioning of the ngx_stream_upstream_check_module, a patch needs to be applied to NGINX. Also,the nginx-1.11.5 version needs to be copied over to the OpenResty bundled directory. Since version 1.11.5 is utilized, NGINX 1.11.2 is removed from the bundle. Figure 12 provides the commands to do this.

Figure 12 - Patch of NGINX 1.11.5 with Output and Added to OpenRestyBundle

```
# cd nginx-1.11.5/
# patch -p0 < /tmp/patch-1.11.x.patch
patching file src/stream/ngx_stream_upstream_hash_module.c
Hunk #3 succeeded at 558 (offset 3 lines).
patching file src/stream/ngx_stream_upstream_least_conn_module.c
Hunk #2 succeeded at 146 (offset -1 lines).
patching file src/stream/ngx_stream_upstream_round_robin.c
Hunk #2 succeeded at 102 with fuzz 1 (offset 1 line).
Hunk #3 succeeded at 462 with fuzz 1 (offset 11 lines).
Hunk #4 succeeded at 562 (offset 4 lines).
patching file src/stream/ngx_stream_upstream_round_robin.h

# cd /tmp
# mv nginx-1.11.5 /tmp/openresty-1.11.2.2/bundle/
# rm -rf /tmp/openresty-1.11.2.2/bundle/nginx-1.11.2
```

Afterwards, the bundle should now include the nginx-1.11.5 as highlighted in Figure 13.

Figure 13 - OpenResty Bundle with NGINX 1.11.5

```
# ls  /tmp/openresty-1.11.2.2/bundle
array-var-nginx-module-0.05          lua-resty-limit-traffic-0.01        ngx_lua-0.10.7
drizzle-nginx-module-0.1.9           lua-resty-lock-0.04                 ngx_lua_upstream-0.06
echo-nginx-module-0.60               lua-resty-lrucache-0.04             ngx_postgres-1.0rc7
encrypted-session-nginx-module-0.06  lua-resty-memcached-0.14            opm-0.0.2
form-input-nginx-module-0.12         lua-resty-mysql-0.17                pod
headers-more-nginx-module-0.32       lua-resty-redis-0.26                rds-csv-nginx-module-0.07
iconv-nginx-module-0.14              lua-resty-string-0.09               rds-json-nginx-module-0.14
install                              lua-resty-upload-0.10               redis2-nginx-module-0.13
lua-5.1.5                            lua-resty-upstream-healthcheck-0.04 redis-nginx-module-0.3.7
lua-cjson-2.1.0.4                    lua-resty-websocket-0.06            resty-cli-0.16
LuaJIT-2.1-20161104                  memc-nginx-module-0.17              resty.index
lua-rds-parser-0.06                  nginx-1.11.5                        set-misc-nginx-module-0.31
lua-redis-parser-0.12                nginx-no_pool.patch                 srcache-nginx-module-0.31
lua-resty-core-0.1.9                 ngx_coolkit-0.2rc3                  xss-nginx-module-0.05
lua-resty-dns-0.18                   ngx_devel_kit-0.3.0
root@nlb2:/tmp#
```

With all the sources, add-on modules, patches, and libraries downloaded and setup, the last step is to build and install the OpenResty binaries in their appropriate directories. Before a “make” is initiated to build OpenResty, a ***“configure”*** is performed to set the flags and include all modules that are part of the OpenResty binary. Figure 14 illustrates the configure command with the options, the build and install commands. If an “Error” is encountered during configure, check that all libraries and other dependencies in the previous steps have been installed in the correct directories. An appropriate error message is printed out to indicate reason of error. A successful “configure” outputs the following when completed:

Type the following commands to build and install:

make

make install

The “*make*” and *“make install”* commands should also not produce an Error message and an error message is in the output to provide some clue on the error. Pre-built packages are available and can be downloaded; however, the third party modules used in this example are not included.

Figure 14 - Build and Install of OpenResty

```
# cd /tmp/openresty-1.11.2.2/

# ./configure -j1 --with-openssl=/tmp/openssl-1.0.2j --with-pcre=/tmp/pcre-8.39 --with-file-aio
--with-http_addition_module --with-http_auth_request_module --with-http_dav_module --with-
http_flv_module --with-http_geoip_module=dynamic --with-http_gunzip_module --with-
http_gzip_static_module --with-http_image_filter_module=dynamic --with-http_mp4_module --with-
http_random_index_module --with-http_realip_module --with-http_secure_link_module --with-
http_slice_module --with-http_ssl_module --with-http_stub_status_module --with-http_sub_module
--with-http_v2_module --with-http_xslt_module=dynamic --with-ipv6 --with-mail --with-
mail_ssl_module --with-md5-asm --with-pcre-jit --with-sha1-asm --with-stream     --with-
stream_ssl_module --with-threads --add-module=/usr/local/src/nginx-module-sts-master --add-
module=/usr/local/src/nginx-module-stream-sts-master --add-module=/usr/local/src/nginx-module-
vts-master --add-module=/usr/local/src/ngx_stream_upstream_check_module-master
```

5.1.3 Domain Name System (DNS)

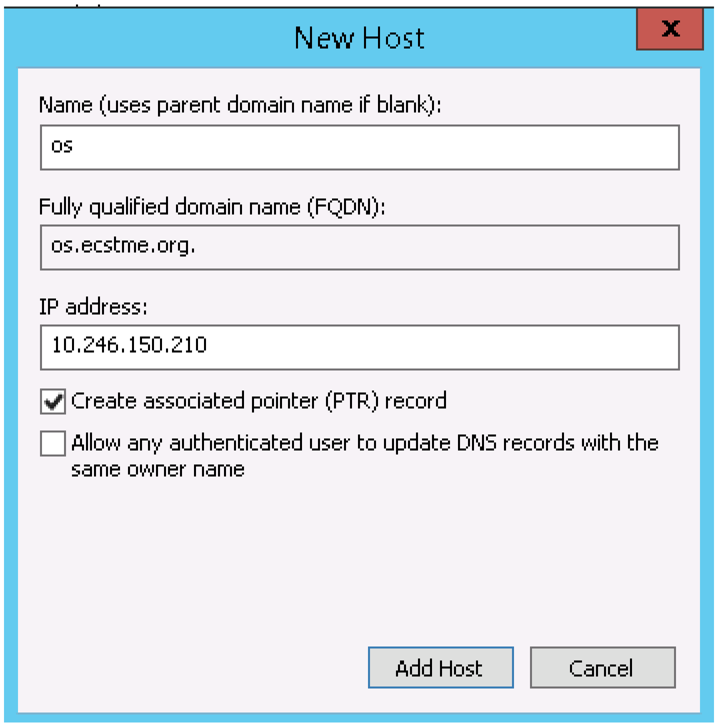
In this example a DNS is setup on a Windows 2012 server and accessible from the server hosting OpenResty. Table 3 shows the DNS entries created. Adding DNS entries allows mapping of “names” to IP addresses. In this example, DNS is used as a mechanism for translating the object protocol (S3 or Swift) the client is using and allows OpenResty to direct request to the appropriate pool of ECS nodes based on protocol name. The names associated with each object protocol are mapped to one IP address associated with OpenResty and translate it to a pool of ECS nodes. An “A-record “ is created in DNS which maps a name to the IP address of OpenResty server and CNAME provides an alias for each protocol.

Table 3 - DNS Entries Example

DNS Record	Record Type	Record Data	Comments
os.ecstme.org	A	10.246.150.210	NGINX external IP Address and also used for S3 protocol access (os=object store)
*.os.ecstme.org	CNAME	os.ecstme.org	Used for S3 virtually hosted buckets, i.e. mybucket.os.ecstme.org
swift.ecstme.org	CNAME	os.ecstme.org	Endpoint for clients using the Swift protocol

From the Windows Server, start up the DNS Manager and add DNS entries of “New Host” for A-Record and “New Alias” in your domain zone as described in above table. Sample screenshots of this are pictured in Figures 15 and 16.

Figure 15 - DNS A-Record for "os".


New Host

Name (uses parent domain name if blank): os

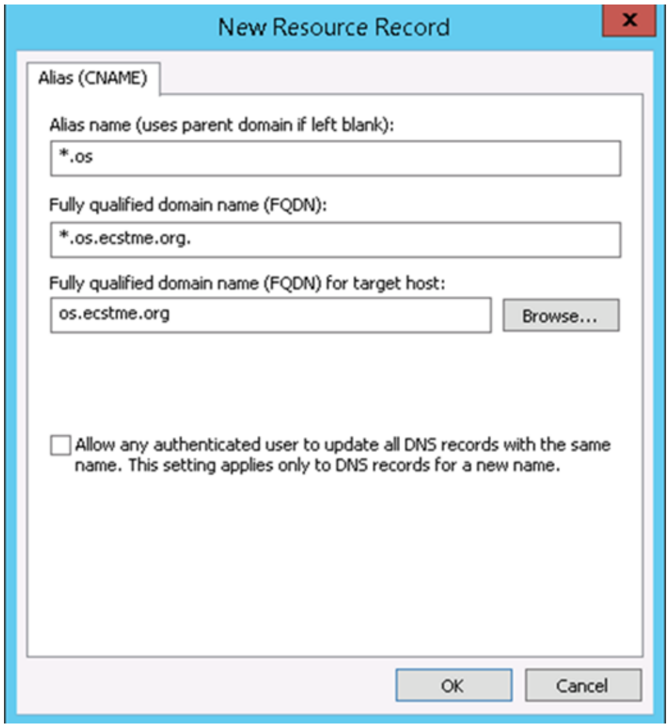
Fully qualified domain name (FQDN): os.ecstme.org.

IP address: 10.246.150.210

[✓] Create associated pointer (PTR) record

[ ] Allow any authenticated user to update DNS records with the same owner name

Figure 16 - DNS Alias Record for "*.os".



Alias name (uses parent domain if left blank): *.os

Fully qualified domain name (FQDN): *.os.ecstme.org.

Fully qualified domain name (FQDN) for target host: os.ecstme.org

☐ Allow any authenticated user to update all DNS records with the same name. This setting applies only to DNS records for a new name.

5.2 OpenResty Configuration

Configuration file(s) are used to define how OpenResty load balances requests to a pool of servers. The NGINX configuration file, *nginx.conf*, is located in "*/usr/local/openresty/nginx/conf*" directory. The ngnix.conf file is structured in blocks of directives surrounded by braces {}.   The examples in this whitepaper predominately define the following contexts or blocks:

- **global or main** – any directives that are defined outside of the braces {} and affects all contexts within file.
- **events** – connection handler
- **http** – defines how HTTP or HTTPS requests are processed
- **stream** – defines how TCP or UDP requests are processed. Similar to http context, it contains global variables and nested server and upstream contexts.

This section describes each context and the embedded Lua scripts in step-wise fashion and can be culminated together to create an nginx.conf file. For reference, the nginx.conf file used in this example is also attached to this whitepaper. Refer to Appendix (A) section.

5.2.1 Global and Event Context

The global context is the directives that applies to the entire web platform and applies to all contexts. It is defined outside of the braces {}. One of the primary directives in the global context is *worker_processes*. NGINX has a "master process" responsible for maintaining worker processes. The *worker_processes* directive in the configuration file defines how many worker processes the master spawns to handle requests. This number is usually based on the number of CPU cores on the host running NGINX. In this simple example, *work_processes* is set to 1. This value can be tuned and optimized based on your server and workload requirements.

The event context block specifies the directives related to the connection processing. It can include the following:

- **worker_connections** - number of clients that can be served per worker process
- **epoll** – an event notification mechanism to streamline I/O and improve scalability of NGINX for processing requests.
- **multi-accept** – if "on", it accepts as many connections as possible

These values can also be tuned based on workload requirements. Figure 17 exemplifies these directives.

Figure 17 - Global and Event Context Directives

```
worker_processes  1;

events {
        worker_connections 2048;
        use epoll;
        multi_accept on;
        }
```

5.2.2 HTTP Context

The http context includes all the directives and nested blocks or contexts (i.e. server, location, etc) that define how OpenResty processes HTTP client requests. Inside this context are the following:

- **general** – definitions utilized within http context such as error and access logs, proxy buffers, proxy buffering, error and access log files, paths, log format, etc.
- **server** – handler for a specific type of HTTP client requests and contains listening ports, ssl certificates, etc.
- **upstream** – defines a named pool of backend servers that requests are sent to and the load balancing algorithm
- ***lua*** – embedded Lua scripts to handle healthchecks and metric collection.

5.2.2.1 General Directives

The definitions or directives defined within the http context that are not surrounded by braces, {}, apply to all the nested contexts or blocks within the http context. As a best practice, if there are directives that are applicable to one or more nested context it should be defined in the higher context or outside of the nested context. In this example shown in Figure 18, directives declared outside of nested context are related to logging, performance optimizations, and connection time out values. Lua and third party modules directives can also be specified at this level if it is needed prior to defining a context or block.

The first set of directives defined in this example is the log format for the access log and error log. The logs are generally located relative to install directory of OpenResty (i.e. */usr/local/openresty/nginx/logs)* or as specified in the *access_log* and *error_log* directives.

OpenResty by default reads and writes to internal buffers. These buffers contain data received and sent to and from clients and data received and sent to and from proxy servers. If these buffers are set too low, then temporary files are used to store the data. The purpose of the buffering directives is to optimize performance. The values of the buffering directives can be tuned based on the workload.   There are several optimizations that are utilized in this example. The buffers relating to client requests include the following:

- *client_max_body_size* – maximum size of the body of client request. Set to 0 to disable checking of body size.
- *client_body_buffer_size* –the client buffer size (ie. a POST action)
- *client_header_buffer_size* – maximum header size for a client request.

Another set of the optimizations relates to internal buffering of requests and responses from proxied servers. In this example, *proxy_request_buffering* is disabled so that the request body is immediately sent to the proxied servers. As for responses, they are buffered and the *proxy_buffers* specify the number of allocated buffers and size for a request. Also, the *proxy_buffer_size* is defined to indicate the buffer size to store the first part of the response from a proxied server. This is a separate buffer and can be smaller than the buffers for the rest of the response.

Further optimizations can be implemented by using the tcp_nodelay, tcp_nopush and sendfile directives. These directives work together to optimize how data is sent at the system (kernel) and TCP layer. The directive *tcp_nodelay* sends data in the socket buffer immediately without delay. If not enabled, there can be up to 0.2 seconds before data is sent. Historically the TCP stack set this delay to avoid network congestion and to handle very small amount of data that did not fill up the packet. However, today, the packet is usually full and enabling *tcp_nodelay* turns off this behavior. As for *tcp_nopush*, this ensures that packets are full or at a certain optimal level before they are sent. The *sendfile* directive reduces the internal overhead when transferring static files. It utilizes file descriptors and mmap to directly send data without extra context switching and copying of data. As for the *keepalive_timeout* value, it specifies how long a client connection is open on the server side.

Figure 18 - HTTP context General Directives

```
http {
        log_format main   '$remote_addr - $remote_user [$time_local] "$request" '
                       '$status $bytes_sent $body_bytes_sent "$http_referer" '
                       '"$http_user_agent" ua=$upstream_addr us=$upstream_status '
                           'rt=$request_time rl=$request_length
        uct=$upstream_connect_time '
                       'uht=$upstream_header_time urt=$upstream_response_time';

        access_log logs/access.log main buffer=32k flush=5s;
        error_log  logs/error.log warn;

        client_max_body_size 0;
        client_body_buffer_size 8M;
        client_header_buffer_size 32K;

        proxy_http_version 1.1;
        proxy_request_buffering off;
        proxy_buffering off;
        proxy_buffer_size 2M;
        proxy_buffers 100 2M;

        sendfile       on;
        tcp_nopush     on;
        tcp_nodelay    on;

        keepalive_timeout  60;

        ...
        ...
        ...
}
```

5.2.2.2 Upstream Context

The upstream context declares the pool of backend servers or ECS nodes that OpenResty sends HTTP requests for handling. Based upon the load balancing algorithm specified would determine how the requests are balanced across the servers. Supported load balancing algorithm include round –robin (default), least connected and ip-hash. The upstream context is defined outside of the server context. The name of the upstream context is referenced within the server or location blocks. In Figure 19, the name of the upstream context is *ecs_9020* and *ecs_9024* and the IP addresses of each ECS node and protocol port are preceded by a *server* directive. For S3 object requests, port 9020 is specified and for Swift, port 9024 is specified. The other ECS protocols communicating over HTTP would be in their own *named* upstream block and have similar definitions but different port numbers. If there are a lot of upstream servers, it is recommended to create a separate file for the upstream directives for readability and better manageability. Then, within the http context, add an “include” directive with name of file inside of nginx.conf. If no load balancing algorithm is defined in the context, the default load balancing algorithm, round robin is used. The keepalive directive indicates the number of idle keepalive connections to upstream server that remain open.

Figure 19 - Upstream Context within HTTP context

```
upstream ecs_9020 {
     server 10.246.150.175:9020;
     server 10.246.150.176:9020;
     server 10.246.150.177:9020;
     server 10.246.150.178:9020;
      keepalive 32;
}

upstream ecs_9024 {
      server 10.246.150.175:9024;
      server 10.246.150.176:9024;
      server 10.246.150.177:9024;
      server 10.246.150.178:9024;
      keepalive 32;
 }
```

5.2.2.3 Health Check Lua Script

The value of OpenResty is the inclusion of LuaJIT and other lua modules that allow developing of scripts to customize the web platform. Dell EMC DevOps team has developed several sample Lua scripts. One of the Lua scripts is the health checks of ECS nodes defined in the *"init_worker_by_lua_block"* context. In Figure 20, there are several http context directives defined prior to the Lua blocks which include:

- **lua_code_cache** – indicates whether Lua code is cached or not.
- **lua_shared_dict** – declares a named shared memory zone, *healthcheck* , of certain size, *1M*, and is shared by all NGINX worker processes.
- **lua_socket_log_errors** – enables or disables error logging when a failure occurs for the TCP or UDP cosockets.

These directives are used by several of the Lua scripts and thus declared outside of the Lua block(s).

Figure 20 - General Directives for Lua Blocks

```
lua_package_path "${prefix}/lua/?.lua;;";
lua_code_cache on;

lua_shared_dict healthcheck 1m;
lua_socket_log_errors off;
```

The *"init_worker_by_lua_block"* implements the health checks for the ECS nodes and uses the built-in lua-resty-upstream-check module. It runs on every NGINX worker process startup if master process is enabled or when the NGINX configuration file is loaded if master process is disabled. It sends an HTTP “ping” request to validate the ECS nodes and services are up and available to handle requests. The upstream context named ecs_9020 and ecs_9024 indicates the pool of ECS nodes. As shown in the comments in Figure 21, this runs a check every 5 seconds and considers a node down if the ping fails once. Errors in spawning this Lua script are logged in error.log file in the install directory of OpenResty (i.e. */usr/local/openresty/nginx/logs*).

Figure 21 - ECS Health Check Lua Script within HTTP Context

```
    init_worker_by_lua_block {
        local hc = require "resty.upstream.healthcheck"

        local ok, err = hc.spawn_checker {
            shm = "healthcheck",      -- defined by "lua_shared_dict"
            upstream = "ecs_9020",    -- defined by "upstream"
            type = "http",

            http_req = "GET /?ping HTTP/1.0\r\nHost: ecs_9020\r\n\r\n",
                    -- raw HTTP request for checking

            interval = 5000,  -- run the check cycle every 5 secs
            timeout = 30000,   -- 30 secs is the timeout for network operations
            fall = 1,  -- # of successive failures before turning a peer down
            rise = 2,  -- # of successive successes before turning a peer up
            valid_statuses = {200, 302},  -- a list valid HTTP status code
            concurrency = 2,  -- concurrency level for test requests
        }

        if not ok then
            ngx.log(ngx.ERR, "failed to spawn http health checker: ", err)
            return
        end


        local ok, err = hc.spawn_checker {
            shm = "healthcheck",      -- defined by "lua_shared_dict"
            upstream = "ecs_9024",    -- defined by "upstream"
            type = "http",

            http_req = "GET /auth/v1.0 HTTP/1.0\r\nHost: ecs_9024\r\n\r\n",
                    -- raw HTTP request for checking

            interval = 5000,  -- run the check cycle every 5 secs
            timeout = 60000,   -- 30 secs is the timeout for network operations
            fall = 3,  -- # of successive failures before turning a peer down
            rise = 1,  -- # of successive successes before turning a peer up
            valid_statuses = {401},  -- a list valid HTTP status code
            concurrency = 2,  -- concurrency level for test requests
        }

        if not ok then
            ngx.log(ngx.ERR, "failed to spawn http health checker: ", err)
            return
        end
    }
```

5.2.2.4 Server Context

The server context is defined within the HTTP context and includes the following directives:

- **listen** – IP address and/or port combination that this context is listening to.
- **location** – defines how certain requests that match a specified criteria (ie. header) are processed. Multiple location contexts can be specified.

Figure 22, indicates that this server is listening on port 9020 and passes HTTP requests to the proxied servers (ECS nodes) of upstream context name, name *ecs_9020* for handling of S3 requests. Similarly, a server context is defined for Swift on port 9024and passes to upstream context named *ecs_9024*. There can be multiple server contexts defined with different listening ports and set of proxied servers. This allows for handling of different types of client requests. The proxy_set_header adjusts the headers for Swift connections. For some applications, for instance Cyberduck, the proxy_pass URL is being specified in the response header and by setting the proxy_set_header, it is corrected.

Figure 22 - Server Context within HTTP Context

```
        server {
                listen 9020;
                location / {
                    proxy_pass http://ecs_9020;
                }
        }


        server {
                listen 9024;

                location / {
                    proxy_pass http://ecs_9024;
                   proxy_set_header Host $host:$server_port;
                }
        }
```

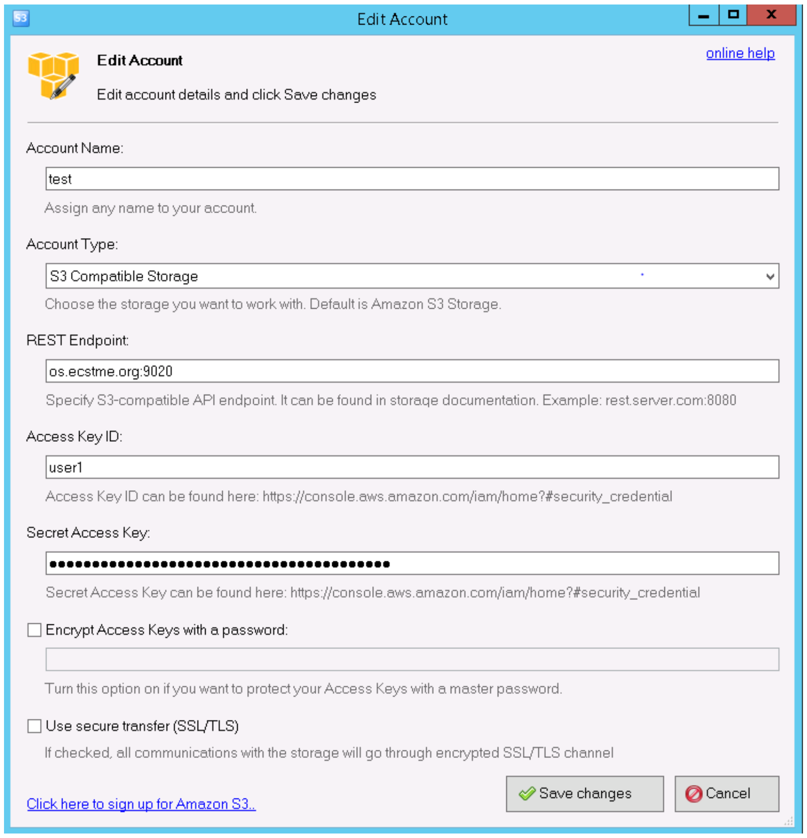
After the nginx.conf has been modified, check the validity of the configuration file. If configuration file is valid, then start OpenResty service to put in effect the directives in nginx.conf. Figure 23 exhibits how to check and start OpenResty.

Figure 23 - Example Command to Check nginx.conf and Start OpenResty

```
# /usr/local/openresty/bin/openresty -t -c /usr/local/openresty/nginx/conf/nginx.conf

nginx: the configuration file /usr/local/openresty/nginx/conf/nginx.conf syntax is ok
nginx: configuration file /usr/local/openresty/nginx/conf/nginx.conf test is successful

#  /usr/local/openresty/bin/openresty
```

To validate the OpenResty configuration, install S3 Browser or curl and set the IP or name of host running OpenResty. Figure 24 provides an example of the settings for S3 Browser.   Since HTTP is being tested, verify that the S3 Browser has the “Use secure transfer (SSL/TLS)” box unchecked. Also specify port 9020 on REST endpoint since, OpenResty is listening on this port for HTTP S3 requests.

Figure 24 - S3 Browser Settings Example Using OpenResty



Edit Account

Edit account details and click Save changes

Account Name: test

Assign any name to your account.

Account Type: S3 Compatible Storage

Choose the storage you want to work with. Default is Amazon S3 Storage.

REST Endpoint: os.ecstme.org:9020

Specify S3-compatible API endpoint. It can be found in storage documentation. Example: rest.server.com:8080

Access Key ID: user1

Access Key ID can be found here: https://console.aws.amazon.com/iam/home?#security_credential

Secret Access Key: ••

Secret Access Key can be found here: https://console.aws.amazon.com/iam/home?#security_credential

☐ Encrypt Access Keys with a password:

Turn this option on if you want to protect your Access Keys with a master password.

☐ Use secure transfer (SSL/TLS)

If checked, all communications with the storage will go through encrypted SSL/TLS channel

The Linux curl command can also be utilized to test both S3 and Swift. In this example, a curl command is utilized to test Swift. The DNS alias, “swift.ecstme.org” and port 9024 for the server settings and the Swift credentials (e.g. user as suser1 and password for Swift user) as shown in Figure 25 are passed to curl command.   An HTTP response of 204 and ECS token are returned to indicate command succeeded.

Figure 25- Curl Command with Swift Settings Example Using OpenResty

```
# curl -i -H "X-Storage-User: suser1" -H "X-Storage-Pass: dangerous"
http://swift.ecstme.org:9024/auth/v1.0 -v
* About to connect() to swift.ecstme.org port 9024 (#0)
*   Trying 10.246.150.210... connected
* Connected to swift.ecstme.org (10.246.150.210) port 9024 (#0)
> GET /auth/v1.0 HTTP/1.1
> User-Agent: curl/7.19.7 (x86_64-redhat-linux-gnu) libcurl/7.19.7 NSS/3.18 Basic ECC
zlib/1.2.3 libidn/1.18 libssh2/1.4.2
> Host: swift.ecstme.org:9024
> Accept: */*
> X-Storage-User: suser1
> X-Storage-Pass: dangerous
>
< HTTP/1.1 204 No Content
HTTP/1.1 204 No Content
< Server: nginx/1.11.5
Server: nginx/1.11.5
< Date: Mon, 12 Jun 2017 12:32:32 GMT
Date: Mon, 12 Jun 2017 12:32:32 GMT
< Connection: keep-alive
Connection: keep-alive
< X-Storage-Url: http://ecs_9024/v1/testns
X-Storage-Url: http://ecs_9024/v1/testns
< X-Auth-Token: ECS_05df3a250eba4685b43a66908e097914
X-Auth-Token: ECS_05df3a250eba4685b43a66908e097914
< x-emc-request-id: 0af696b5:15c59ff8277:e415:0
x-emc-request-id: 0af696b5:15c59ff8277:e415:0

<
* Connection #0 to host swift.ecstme.org left intact
* Closing connection #0
```

5.2.2.5 SSL Certificates

Developing trust between two entities is established via Secure Socket Layer (SSL) and SSL certificates. The purpose of SSL and certificates is encryption and identification to ensure that communication exchange between two parties is secure and trustworthy. For identification of who to trust, a certificate would need to be generated and installed on OpenResty or ECS nodes. Certificates usually contain information about the owner of the certificate (i.e. company, organization, city, state, email address of owner), duration of validity, resource location (i.e. Fully Qualified Domain Name or common name), public key, and hash. The certificate generated can be signed by either:

- Certificate Authority (CA) – trusted organizations that can verify the identity and validity of the entity requesting the certificate.
- Self-signed - authenticated by the system where the certificate resides.

As a best practice, using a Certificate Authority to sign certificate is preferred over issuing self-signed certificates. Almost all client systems come with a list of trusted root CA certificates that automatically validates your CA-signed certificate. If you opt to use a self-signed certificate, that certificate will need to be installed on all client systems as a “trusted” certificate. If your organization has an internal Certificate Authority, you can use that CA to sign your certificates; just make sure that any client systems have your CA’s root certificate installed. Also, utilizing OpenResty to offload and terminate SSL is a best practice in order to not add extra load on ECS nodes to establish SSL sessions. Thus, the certificate generated in this example is to be installed on host running OpenResty. Some organizations have security policies that specify that the connection to ECS must be encrypted all the way to ECS. In those cases you need to terminate SSL on the ECS appliance itself. If SSL termination is required on ECS nodes, then use Layer 4 (tcp) load balancing mode to pass through the SSL traffic to ECS nodes for handling. In this scenario, the certificates would need to be installed on ECS. For information on how to generate certificates for ECS Nodes, refer to ECS System and Administration Guide.

Certificate Generation for OpenResty Example

OpenSSL is used in this example to generate the certificates. Note that you do not need to generate the certificates on ECS; any system with suitable tools like OpenSSL can generate certificates. By default, OpenSSL is installed on most Linux releases. General steps to create a certificate for OpenResty using OpenSSL include:

1. Generate a private key.
2. Modify configuration file to add Subject Alternative Names (SANs).
3. Create a certificate request to submit to CA or generate a self-signed certificate.
4. Combine the private key and certificate and place in OpenResty directory.

When generating certificates, the hostname of where the certificate is used needs to be specified. For compatibility with the S3 protocol, the Common Name (CN) on the certificate should point to the wildcard DNS entry used by S3 because S3 utilizes virtually hosted-style URL buckets where the bucket name is in the hostname. There can only be one wildcard entry on an SSL certificate and it must be under the CN. If specifying IP addresses or other DNS entries for Atmos and Swift protocols, Subject Alternative Names (SANs) should be registered on the certificate. Some organizations do not allow the use of wildcard certificates. In this case, you need to make sure that all of your S3 applications use “path-style” access so they can use the base hostname of S3 (e.g. os.ecstme.org) instead of the default method of adding the bucket to the hostname (e.g. bucket.os.ecstme.org).

Step 1: Generate a Private Key

A private key is required for self-signed and CA requests certificates. This key is combined with the certificate generated. An example of how to generate the private key is shown in Figure 26. Permissions are also changed on the generated key to safeguard from accidental modification or deletion.

Figure 26 - Example Command to Create Private Key Using OpenSSL

```
# openssl genrsa -des3 -out server.key 2048
Generating RSA private key, 2048 bit long modulus
............................................................+++
.......+++
e is 65537 (0x10001)
Enter pass phrase for server.key: <enter a password>
Verifying - Enter pass phrase for server.key: <enter a password>

# chmod 0400 server.key
```

Step 2: Modify the Configuration File with SANs

OpenSSL does not allow passing of SANs through the command line so a configuration file is created to define them. A sample configuration file for openssl can be used as a reference and is located in /usr/lib/ssl/openssl.cnf. Copy the openssl.cnf file to a temporary directory where certificates are generated and placed as pictured in Figure 27.

Figure 27- Copying of Openssl Configuration File

```
# cp /usr/lib/ssl/openssl.cnf request.conf
```

Edit the request.conf file to include the SAN by adding the IP addresses or DNS entries mapping of the OpenResty web-platform. Figure 28 is an example of the SAN setting for both DNS entries and IP addresses.

Figure 28 – SANs Setting in Configuration File

```
[ alternate_names ]
DNS.1 = os.ecstme.org
DNS.2 = swift.ecstme.org
IP.1 = 10.246.150.210
```

In the [req] section, add the following lines if not present in the configuration file as shown in Figure 29.

Figure 29 - Indicate Parameter(s) for Extensions

```
x509_extensions = v3_ca # The extentions to add to the self-signed cert
req_extensions = v3_ca  # For cert signing req
```

In the [v3_ca] section, add the following lines as shown in Figure 30. This indicates that there are alternate names provided.

Figure 30 - Specify [v3_ca] Parameters for SAN

```
        [ v3_ca ]
        subjectAltName    = @alternate_names
        basicConstraints = CA:FALSE
        keyUsage = nonRepudiation, digitalSignature, keyEncipherment
        extendedKeyUsage = serverAuth
```

Also in [v3_ca] section, if creating a certificate signing request, comment out "authorityKeyIdentifier" as illustrated in Figure 31. There is no need to comment this out for self-signed certificates.

Figure 31 – Comment out authorityKeyIdentifier paramteter in [v3_ca] Section

```
#authorityKeyIdentifier=keyid:always,issuer
```

Finally in section [CA_default], uncomment or add the copy_extension line as pictured in Figure 32.

Figure 32 - Parameter to Add at [CA_default] Section

```
copy_extension=copy
```

Step 3a: Creation of a Certificate Signing Request for CA Submission

Figure 33 provides an example ***openssl*** command of how to create a certificate signing request. The command requires the private key, *"server.key"* created in Step 1, and the modified configuration file, request.conf containing the subject alternate names as described in Step 2. Several user inputs are expected such as location and organization information, email address and Common Name.   As previously mentioned, the Common Name should be set to the wildcard DNS entry used by S3 which is "*.os.ecstme.org" in this example.

Figure 33 - Example Command to Generate a Certificate Signing Request

```
# openssl req -new -key server.key -config request.conf -out server.csr

Enter pass phrase for server.key: <your passprhase from above>

You are about to be asked to enter information that will be incorporated into your
certificate request. What you are about to enter is what is called a Distinguished Name
or a DN. There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]: <Enter value>
State or Province Name (full name) [Some-State]: <Enter value>
Locality Name (eg, city) []: <Enter value>
Organization Name (eg, company) [Internet Widgits Pty Ltd]: <Enter value>
Organizational Unit Name (eg, section) []: <Enter value>
Common Name (e.g. server FQDN or YOUR name) []: *.os.ecstme.org
Email Address []: <admin email>

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []: <optional>
An optional company name []: <optional>
```

Step 3b: Creation of a Self-Signed Certificate

The command to create self-signed certificates is similar to certificate request except for the addition of *"-x509"* option. Figure 34 provides an example command to generate the self-signed certificate. Also, the Common Name is set to *"*.os.ecstme.org"* for the S3 wildcard DNS entry. The validity of this certificate by default is one month, if more days are desired, specify command with "-days <# of days> (i.e." –days 366").

Figure 34 - Example Command for Creation of Self-Signed Certificate

```
# openssl req -x509 -new -key server.key -config request.conf -out server.crt

Enter pass phrase for server.key: <your passprhase from above>

You are about to be asked to enter information that will be incorporated into your
certificate request. What you are about to enter is what is called a Distinguished Name
or a DN. There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]: <Enter value>
State or Province Name (full name) [Some-State]: <Enter value>
Locality Name (eg, city) []: <Enter value>
Organization Name (eg, company) [Internet Widgits Pty Ltd]: <Enter value>
Organizational Unit Name (eg, section) []: <Enter value>
Common Name (e.g. server FQDN or YOUR name) []: *.os.ecstme.org
Email Address []: <admin email>

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []: <optional>
An optional company name []: <optional>
```

Step 3c: Validation of SANs in Certificate Request and Self Signed Certificate

In either certificate generation, user inputs and the Subject Alternate Names provided are included in the certificate. An openssl command to output certificate in text format is highlighted in Figure 35 and Figure 36 for each type of certificates.

Figure 35 - Example Text Output of Certificate Request

```
# openssl req -in server.csr -text -noout
       Certificate Request:
           Data:
               Version: 0 (0x0)
               Subject: C=US, ST=CA, L=SJ, O=Dell, OU=TME,
CN=*.os.ecstme.org/emailAddress=example@dell.com
               Subject Public Key Info:
                   Public Key Algorithm: rsaEncryption
                       Public-Key: (2048 bit)
                       Modulus:
                           00:ca:02:a9:4f:88:eb:9b:bf:07:b5:e7:a5:03:c7:
                           59:2d:e1:53:1e:df:fa:9d:6b:cd:4a:22:10:01:ed:
                           ca:92:66:28:f3:dc:b3:1c:8f:dd:1c:7c:b2:f0:4c:
                           18:63:ff:0e:47:00:35:6f:f8:ae:9b:19:88:2d:f3:
                           f7:de:ed:7b:bc:20:41:ff:d8:c2:28:00:65:a1:38:
                           71:66:10:f1:3f:39:23:f7:40:65:9d:f5:3f:85:33:
                           b0:d3:b3:6a:32:2f:cc:48:50:2e:57:ec:28:19:f2:
                           85:01:d1:32:9c:51:df:2d:5f:0c:93:97:ad:cd:48:
                           1d:fe:50:5d:cc:44:03:15:48:20:cb:cf:b6:77:a4:
                           8c:11:71:57:68:34:9c:b8:3b:30:2e:0e:17:3e:78:
                           34:f4:66:bc:1c:99:9c:bb:ae:50:7f:89:53:f4:1f:
                           b2:7b:21:9c:34:42:01:66:eb:42:47:9d:af:ab:91:
                           6e:16:49:3c:cc:d4:51:14:96:1e:98:cc:c0:08:d1:
                           a8:71:a4:ab:aa:c6:a5:c4:b7:91:74:20:de:bf:e2:
                           71:b8:65:23:3e:3f:f5:21:c7:10:c3:d5:21:0a:52:
                           c6:a6:89:c7:ec:6e:ee:0f:78:58:3f:28:1a:92:b0:
                           40:a9:a8:a2:84:74:e0:72:b4:3e:c4:19:0c:d4:31:
                           6d:37
                       Exponent: 65537 (0x10001)
               Attributes:
               Requested Extensions:
                   X509v3 Subject Alternative Name:
                       DNS:os.ecstme.org, DNS:swift.ecstme.org, IP Address:10.246.150.210
                   X509v3 Key Usage:
                       Digital Signature, Non Repudiation, Key Encipherment
                   X509v3 Extended Key Usage:
                       TLS Web Server Authentication
                   X509v3 Subject Key Identifier:
                       A6:60:1C:05:50:ED:09:7D:BB:6D:1A:87:1D:43:C0:A9:B2:D3:79:7B
                   X509v3 Basic Constraints:
                       CA:TRUE
           Signature Algorithm: sha256WithRSAEncryption
                a9:af:af:42:07:97:14:7b:06:be:0c:c0:65:eb:c0:ee:8d:c5:
                5f:1c:70:fc:1a:1a:f4:83:fb:6b:63:1f:23:4f:0c:26:21:1f:
                cd:6d:5d:94:63:dc:20:0c:89:0a:ff:8a:cc:db:e0:35:18:bf:
                67:0e:2d:ab:10:51:27:6b:77:27:2e:88:6e:0a:a9:ec:25:e2:
                25:d4:b6:bb:f0:17:a7:76:6c:bb:df:c4:59:56:07:9c:2b:68:
                7a:ee:66:ba:32:9d:ea:04:3e:fc:bc:ac:d4:d0:80:10:b7:d7:
                b1:19:de:fe:56:f9:5d:4e:ed:78:c5:ff:ff:cd:7a:d1:89:6f:
                77:47:f4:6e:7b:2b:16:ba:01:28:54:e7:fa:31:dd:87:d0:2e:
                2a:65:d8:ad:42:4d:5d:eb:d7:ab:54:ac:fe:3b:79:91:39:53:
                26:e7:a4:4a:ce:9a:fa:60:0f:35:a1:3f:43:20:6f:6b:4c:29:
                4e:dc:5a:74:d5:a4:55:f5:28:a8:07:e4:e2:7d:bb:ce:ef:ba:
                1c:9a:3f:22:a8:d1:9a:43:e1:75:07:1e:27:61:4e:ac:58:85:
                05:e2:97:7c:8f:9a:e2:5c:15:48:9d:4e:c4:f0:4c:2d:ad:70:
                8e:25:d4:42:7c:a0:1f:85:f7:c3:ae:b3:2c:3f:fd:90:6e:48:
                c0:f0:e0:28
```

Figure 36 – Sample Self-Signed Certificate Output after Generation

```
# openssl x509 -in server.crt -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 12752130309296882435 (0xb0f8aaf3df220303)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, ST=CA, L=SJ, O=Dell, OU=TME,
CN=*.os.ecstme.org/emailAddress=example@dell.com
        Validity
            Not Before: Dec 22 00:37:35 2016 GMT
            Not After : Jan 21 00:37:35 2017 GMT
        Subject: C=US, ST=CA, L=SJ, O=Dell, OU=TME,
CN=*.os.ecstme.org/emailAddress=example@dell.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:ca:02:a9:4f:88:eb:9b:bf:07:b5:e7:a5:03:c7:
                    59:2d:e1:53:1e:df:fa:9d:6b:cd:4a:22:10:01:ed:
                    ca:92:66:28:f3:dc:b3:1c:8f:dd:1c:7c:b2:f0:4c:
                    18:63:ff:0e:47:00:35:6f:f8:ae:9b:19:88:2d:f3:
                    f7:de:ed:7b:bc:20:41:ff:d8:c2:28:00:65:a1:38:
                    71:66:10:f1:3f:39:23:f7:40:65:9d:f5:3f:85:33:
                    b0:d3:b3:6a:32:2f:cc:48:50:2e:57:ec:28:19:f2:
           .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
                      .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
           .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..


                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Alternative Name:
                DNS:os.ecstme.org, DNS:swift.ecstme.org, IP Address:10.246.150.210
            X509v3 Key Usage:
                Digital Signature, Non Repudiation, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
            X509v3 Subject Key Identifier:
                A6:60:1C:05:50:ED:09:7D:BB:6D:1A:87:1D:43:C0:A9:B2:D3:79:7B
            X509v3 Authority Key Identifier:
                keyid:A6:60:1C:05:50:ED:09:7D:BB:6D:1A:87:1D:43:C0:A9:B2:D3:79:7B

            X509v3 Basic Constraints:
                CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
         56:20:f9:7d:04:10:9e:b1:dd:47:fe:2a:6a:52:ee:af:5a:6c:
         54:21:6c:4f:69:5c:00:b6:4d:ba:69:cf:00:30:5a:13:46:cf:
         33:bd:c8:90:9e:2f:f9:6d:1d:b6:88:4b:12:16:69:95:a9:98:
         f4:ed:2c:57:ec:57:20:b6:98:5b:a3:8f:68:c5:e0:73:8b:d9:
         e3:a0:11:7b:26:a9:a3:03:0c:78:1e:87:15:a6:32:96:87:20:
.. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
        .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
        .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
```

Step 4: Combining the Private Key with Certificate File

After the certificate files have been created, OpenResty requires the private key to be combined with the certificate file. If your certificate was signed by a CA, the intermediate CA files and certificate chain file would need to also be appended to the certificate signing request prior to combining it with the private key. Figure 25 illustrates how to extract the private key by removing the password, concatentating it with a self-signed certificate file to generate a combined file and then placing the combined file in OpenResty directory (i.e. /usr/local/openresty/nginx/ssl)

Figure 37 - Example Commands for Combining the Private Key with Certificate

```
# openssl rsa -in server.key -out server_unsec.key
Enter pass phrase for server.key: <your passprhase from above>

# cat server.crt server_unsec.key > combined.pem
# rm server_unsec.key

# chown root:root combined.pem
# mkdir /usr/local/openresty/nginx/ssl
# cp combined.pem /usr/local/openresty/nginx/ssl
# cp server.key /usr/local/openresty/nginx/ssl
```

Once the certificate generation has been completed, the server context in OpenResty configuration file, “nginx.conf”, is modified with the listening port for S3 HTTPS incoming requests and location of the certificates, combined.pem and server.key. ECS S3 HTTPS port is 9021. Figure 38 provides an example of server context for handling HTTPS requests. In this example the, SSL is terminated at the load balancer and thus certificates are not needed on the ECS nodes. The same backend using the non-SSL ports of the ECS nodes are used as previously defined for upstream context, ecs_9020. Similar definitions are specified for Swift protocol using port 9025; and the proxy_set_header with port 9024 has been defined such that response returned from ECS is set appropriately.

Figure 38 – Add Listening Port and SSL Certificates Directives

```
        server {
                 listen 9020;
                listen 9021 ssl;
                ssl_certificate /usr/local/openresty/nginx/ssl/combined.pem;
                ssl_certificate_key /usr/local/openresty/nginx/ssl/server.key;

                location / {
                    proxy_pass http://ecs_9020;
                }
        }
        server {
                 listen 9024;
                listen 9025 ssl;
                ssl_certificate /usr/local/openresty/nginx/ssl/combined.pem;
                ssl_certificate_key /usr/local/openresty/nginx/ssl/server.key;

                 location / {
                    proxy_pass http://ecs_9024;
                   proxy_set_header Host $host:9024;
                }
        }
```

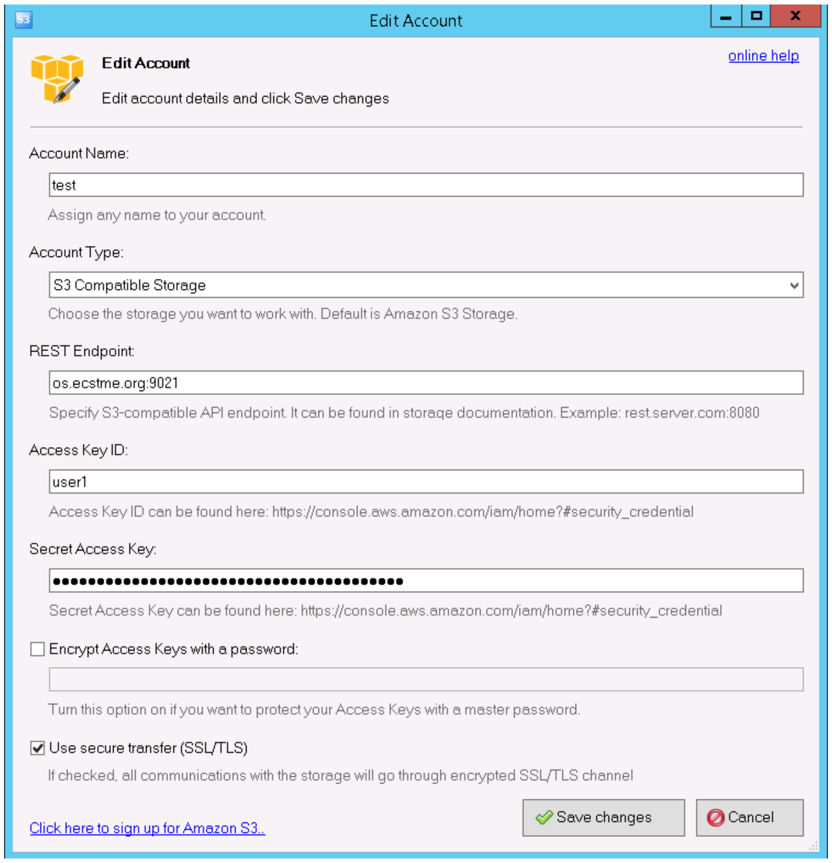
After the nginx.conf has been modified, check the validity of the configuration file. If configuration file is valid, then stop and restart the OpenResty service to put the directives in nginx.conf into effect as shown in Figure 39. When restarting OpenResty, use the pass phrase of server.key (from Figure 33) when prompted for password.

Figure 39 - Sample Commands to Validate nginx.conf and Restart OpenResty

```
#/usr/local/openresty/bin/openresty -t –c /usr/local/openresty/nginx/conf/nginx.conf

nginx: the configuration file /usr/local/openresty/nginx/conf/nginx.conf syntax is ok
nginx: configuration file /usr/local/openresty/nginx/conf/nginx.conf test is successful


# /usr/local/openresty/bin/openresty -s stop
Enter PEM pass phrase:
Enter PEM pass phrase:

# /usr/local/openresty/bin/openresty
Enter PEM pass phrase:
Enter PEM pass phrase:
```

S3Browser can be used to validate that “https” traffic and SSL certificates are going thru OpenResty. For S3 Browser, edit the account and place a checkmark in “Use secure transfer (SSL/TLS) box as pictured in Figure 40. Also specify port 9021 in the REST endpoint since OpenResty is listening to S3 HTTPS request from this port. If using curl to test, use the “-k” option in command and appropriate port, i.e. *“curl -k -i -H "X-Storage-User: suser1" -H "X-Storage-Pass: dangerous" https://swift.ecstme.org:9025/auth/v1.0 -v*”.

Figure 40 - S3 Browser Options to Enable HTTPS


Edit Account

Edit Account

Edit account details and click Save changes

online help

Account Name:

test

Assign any name to your account.

Account Type:

S3 Compatible Storage

Choose the storage you want to work with. Default is Amazon S3 Storage.

REST Endpoint:

os.ecstme.org:9021

Specify S3-compatible API endpoint. It can be found in storage documentation. Example: rest.server.com:8080

Access Key ID:

user1

Access Key ID can be found here: https://console.aws.amazon.com/iam/home?#security_credential

Secret Access Key:

●●

Secret Access Key can be found here: https://console.aws.amazon.com/iam/home?#security_credential

☐ Encrypt Access Keys with a password:

Turn this option on if you want to protect your Access Keys with a master password.

☑ Use secure transfer (SSL/TLS)

If checked, all communications with the storage will go through encrypted SSL/TLS channel

5.2.3 Stream Context

ECS NFS implementation utilizes server-side metadata caching and the cache exists locally on each node. Thus, it is important that NFS requests are handled by the same backend ECS node and a load balancer is used predominately for high availability purposes for NFS. If one of the ECS nodes currently serving NFS request fails, then OpenResty redirects requests to another ECS node.

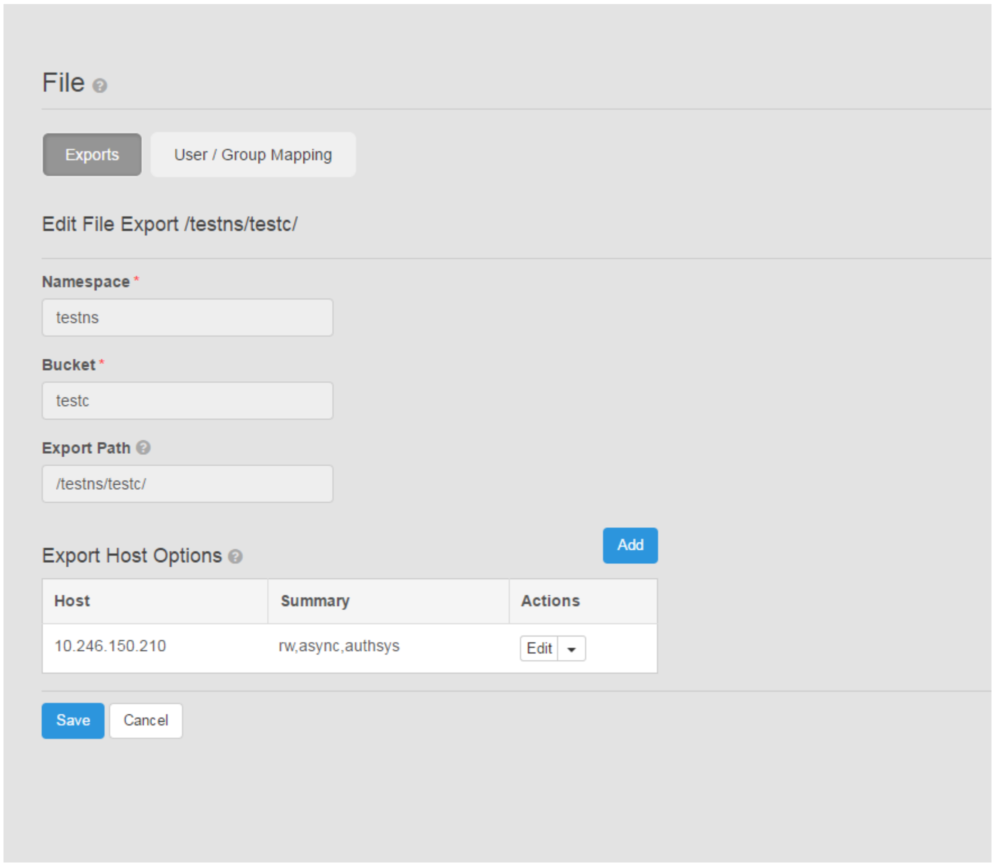
The Linux server that is hosting OpenResty should not be running “rpcbind” otherwise there will be a conflict when setting up OpenResty to forward NFS requests to ECS. If “rpcbind” is installed by default in the operating system, disable or remove. In the OpenResty configuration file, a stream context is defined and requests are directly passed thru to the ECS Nodes for handling. The server context listens for requests on ports 2049, 111, and 10000 where NFS daemons such as nfsd, mountd, nlockmgr and portmapper are listening to on the ECS nodes. As shown in the example directives in Figure 41, there is a stream context and within it multiple upstream contexts. The upstream contexts specify the list of ECS nodes with protocol ports and the load balance algorithm “hash”. The “hash algorithm” selects servers based on a hash of the source IP such as the user IP address to ensure request goes to the same server until something changes in the hash (ie. one backend server goes down). A third-party module, ngx_stream_check_module, is used to provide health checks of stream context via the “check” directive. And server context listens to the NFS protocol ports and proxies it to the named backend. Logging can also be defined in this section as well as tcp and buffer optimizations such as *tcp_nodelay* and *proxy_buffer size.*

Figure 41 – Example Frontend and Backend Definitions for NFS

```
stream {
        log_format main   '$remote_addr - [$time_local] $protocol '
                        '$status $bytes_sent $bytes_received '
                        '$session_time ua=$upstream_addr ubs=$upstream_bytes_sent '
                        'ubr=$upstream_bytes_received uct=$upstream_connect_time';

        access_log logs/access.log main buffer=32k flush=5s;

        tcp_nodelay on;
        proxy_buffer_size 2M;

        upstream nfs1_backend {
                hash $remote_addr;

                server 10.246.150.175:2049 fail_timeout=30s;
                server 10.246.150.176:2049 fail_timeout=30s;
                server 10.246.150.177:2049 fail_timeout=30s;
                server 10.246.150.178:2049 fail_timeout=30s;
         check interval=5000 rise=1 fall=1 timeout=30000 type=tcp;
        }

        upstream nfs2_backend {
                hash $remote_addr;

                server 10.246.150.175:111;
                server 10.246.150.176:111;
                server 10.246.150.177:111;
                server 10.246.150.178:111;
                check interval=5000 rise=1 fall=1 timeout=30000 type=tcp;
        }

        upstream nfs3_backend {
                hash $remote_addr;

                server 10.246.150.175:10000;
                server 10.246.150.176:10000;
                server 10.246.150.177:10000;
                server 10.246.150.178:10000;
         check interval=5000 rise=1 fall=1 timeout=30000 type=tcp;
        }

        server {
                listen 2049;
                proxy_pass nfs1_backend;
        }

        server {
                listen 111;
                proxy_pass nfs2_backend;
        }
        server {
                listen 10000;
                proxy_pass nfs3_backend;
        }
}
```

It is assumed that ECS has been configured to support file and the exports and user and group mappings have been configured appropriately for NFS. The host IP or name running OpenResty should be added to the exports host options in ECS as shown in Figure 42. The user mappings are configured prior to mount as pictured in Figure 43.

Figure 42 - Example of File Export Settings in ECS



Edit File Export /testns/testc/

Namespace *: testns

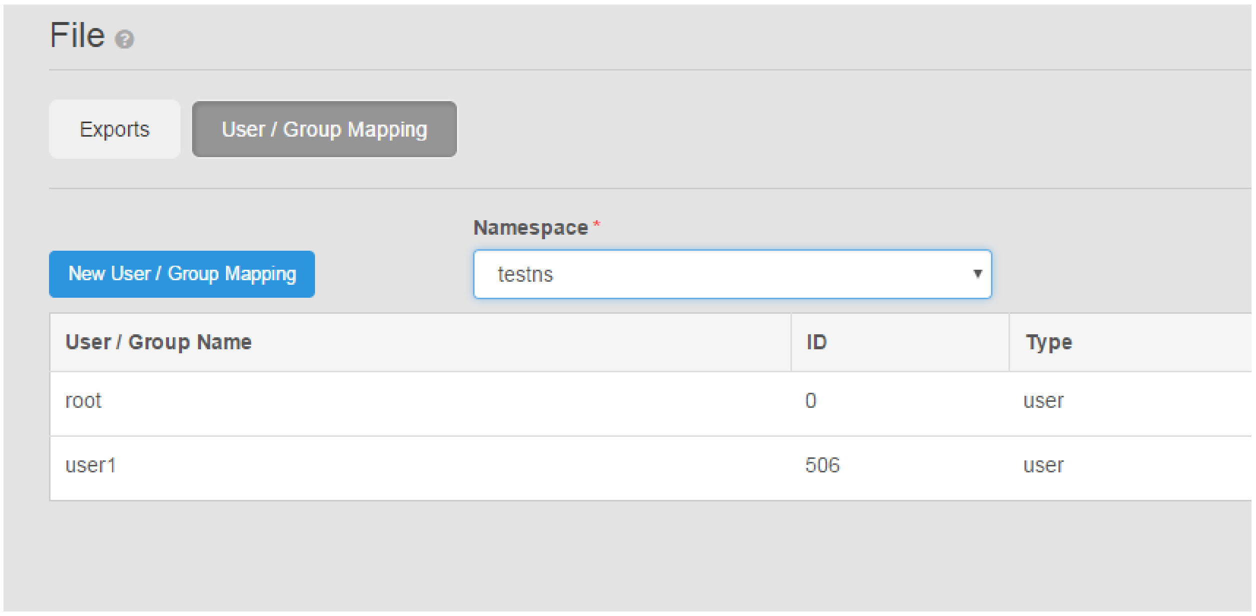
Bucket *: testc

Export Path: /testns/testc/

Export Host Options

Host	Summary	Actions
10.246.150.210	rw,async,authsys	Edit

Figure 43 - Example of User Mapping Settings in ECS



Namespace*: testns

User / Group Name	ID	Type
root	0	user
user1	506	user

After addition of the stream context directives, validate the nginx.conf file and restart OpenResty to pick up the changes to the nginx.conf as shown in Figure 44.

Figure 44 - Validate nginx.conf File and Restart OpenResty

```
#/usr/local/openresty/bin/openresty -t -c /usr/local/openresty/nginx/conf/nginx.conf

nginx: the configuration file /usr/local/openresty/nginx/conf/nginx.conf syntax is ok
nginx: configuration file /usr/local/openresty/nginx/conf/nginx.conf test is successful


# /usr/local/openresty/bin/openresty -s stop
Enter PEM pass phrase:
Enter PEM pass phrase:

# /usr/local/openresty/bin/openresty
Enter PEM pass phrase:
Enter PEM pass phrase:
```

Then from an NFS client, another server different from OpenResty server, issue a mount command specifying the IP or name of OpenResty, port 2049 and NFS version 3. Figure 45 shows the example “mount” command and output of “df” command to illustrate the success of the mount. There should be an equivalent user in the Linux client with the same *userid* (506, in this example) specified as in ECS to access data as shown in Figure 46.

Figure 45 - Example "mount" and "df" Command

```
# mount -o user,vers=3,proto=tcp,port=2049 os.ecstme.org:/testns/testc /mnt

# df
Filesystem              1K-blocks      Used    Available Use% Mounted on
/dev/mapper/vg_vtest-lv_root
                         36645576  13556612     21220780  39% /
tmpfs                     1962340       424      1961916   1% /dev/shm
/dev/sda1                  487652     42509       419543  10% /boot
os.ecstme.org:/testns/testc
                     351063244800 290411520 350772833280   1% /mnt
```

Figure 46 - Example Output of User ID on Linux Client and "ls"

```
[user1@vtest ~]$ id
uid=506(user1) gid=506(user1) groups=506(user1),10(wheel)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023

[user1@vtest ~]$ ls /mnt
foo  fstab  networks  rc1.d
[user1@vtest ~]$
```

5.2.4 Monitoring

Standard NGINX has limited monitoring capabilities; however, when integrated with OpenResty, Lua scripts and additional add-on modules, monitoring is enhanced. Dell EMC DevOps team has developed Lua scripts for monitoring of HTTP requests and connections, HTTP metrics, and health status of the upstream servers. The scripts and modules are categorized as follows:

- HTTP Metrics - tracks number and size of HTTP requests, connections, and latency
  - **init_by_lua_block –** initializes and registers the metrics
  - **log_by_lua_block –** logs the requests, bytes and latency
  - **prometheus.lua** – modified by Dell EMC DevOps team and defines the code for the metrics: counter, guage, and histogram implementation specific to HTTP requests, connections and latency.
- HTTP and Stream Monitoring via Web Page
  - **nginx-module-sts-master and nginx-module-stream-sts-master** – provides stream server traffic status information such as requests per second, responses, and total traffic received and sent. If no stream context is defined, no information will be shown.
  - **nginx-module-vts-master –** provides HTTP server traffic status information.

In this example, the *"init_by_lua_block"* and *"log_by_lua_block"* are responsible for keeping track of the HTTP connections, requests, and latency by host. It utilizes the Prometheus metric library to keep track of the metrics and viewable via a separate web page. As shown in Figure 47, this module is initialized by an "*.init" call and returns a Prometheus object to register different metrics. The code related to the object return is defined in the prometheus.lua file. The prometheus.lua file is attached to this whitepaper and comments within the Lua file are self-explanatory. The reader is encouraged to review the file for more details on the script. The metrics registered and implemented in the prometheus.lua file include gauge, counter and histogram to collect total number of HTTP requests by host, the HTTP request latency by host, total size of incoming requests in bytes and number of HTTP connections. The *init_by_lua_block* runs when the NGINX configuration file is loaded (initialization phase) and the *log_by_lua* at the log request processing phase. The following directives are defined prior to the Lua calls:

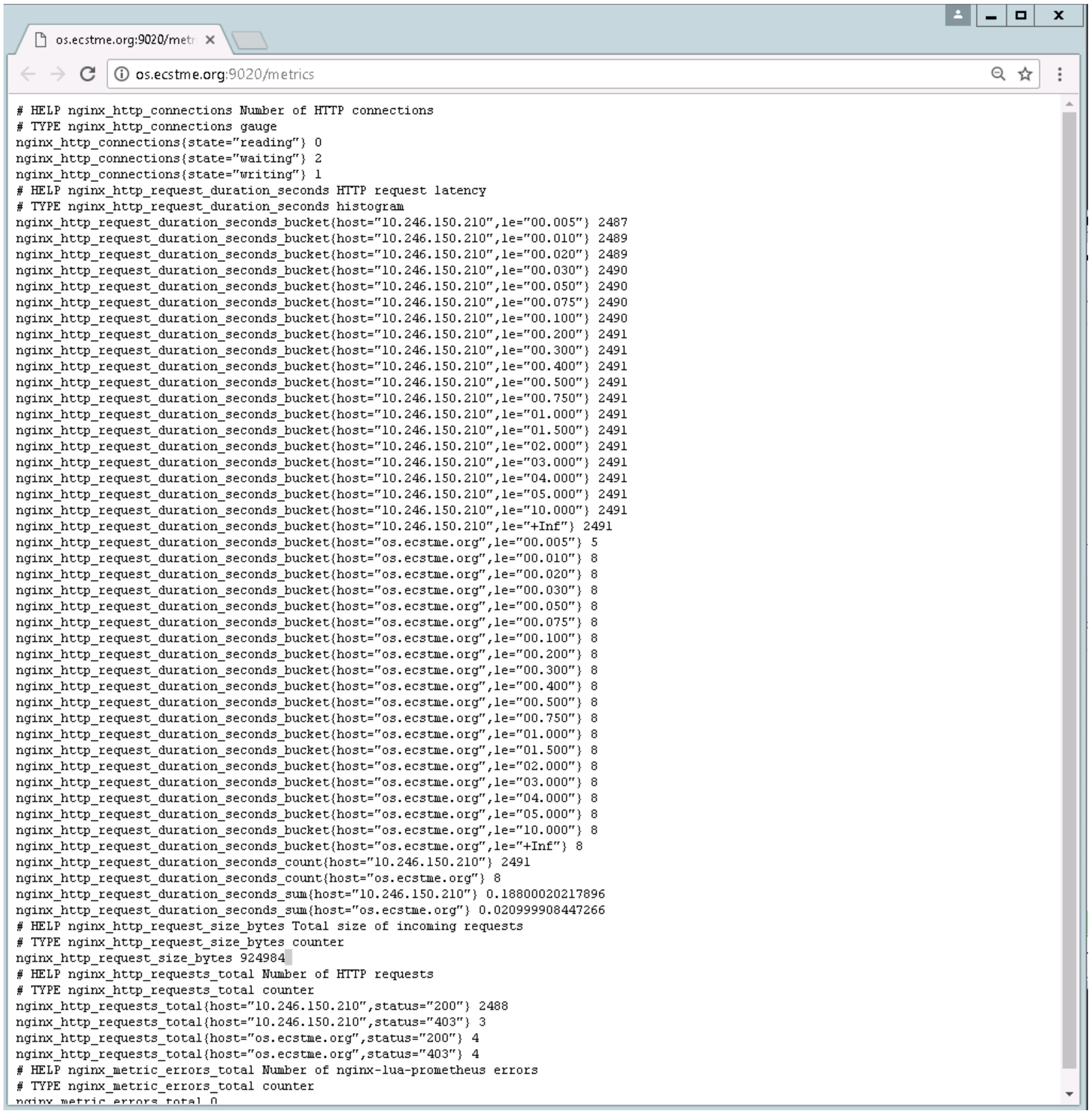
- **lua_package_path** – path to where Lua scripts are located. In this example, it is in a created directory within the OpenResty directory, */usr/local/openresty/nginx/lua.*
- **lua_shared_dict** – declares a named shared memory zone, *healthcheck* , of certain size, *10M*, and is shared by all NGINX worker processes.

These metrics are made viewable by adding a location context labeled *"/metrics"* within the server context. Since the server context is listening on port 9020, access to web page requires the port in the URL (i.e. http://os.ecstme.org:9020/metric) as shown in Figure 48.

Figure 47 – HTTP Metrics Directives

```
http {
    ...
    lua_package_path "${prefix}/lua/?.lua;;";
    lua_shared_dict prometheus_metrics 10M;

    init_by_lua_block {
        prometheus = require("prometheus").init("prometheus_metrics")
        metric_requests = prometheus:counter(
            "nginx_http_requests_total", "Number of HTTP requests", {"host", "status"})
        metric_latency = prometheus:histogram(
            "nginx_http_request_duration_seconds", "HTTP request latency", {"host"})
        metric_bytes = prometheus:counter(
            "nginx_http_request_size_bytes", "Total size of incoming requests")
        metric_connections = prometheus:gauge(
            "nginx_http_connections", "Number of HTTP connections", {"state"})
    }
      log_by_lua_block {
                   metric_requests:inc(1, {ngx.var.host, ngx.var.status})
                   metric_bytes:inc(tonumber(ngx.var.request_length))
                   metric_latency:observe(ngx.now() - ngx.req.start_time(), {ngx.var.host})
               }
               ...
    server {
           listen 9020;
           ...
           location /metrics {
                access_log off;
                content_by_lua_block {
                    metric_connections:set(ngx.var.connections_reading, {"reading"})
                    metric_connections:set(ngx.var.connections_waiting, {"waiting"})
                    metric_connections:set(ngx.var.connections_writing, {"writing"})
                    prometheus:collect()
                }
           }
           ...
   }
}
```

Figure 48 - Example View of HTTP Metric Output



```
# HELP nginx_http_connections Number of HTTP connections
# TYPE nginx_http_connections gauge
nginx_http_connections{state="reading"} 0
nginx_http_connections{state="waiting"} 2
nginx_http_connections{state="writing"} 1
# HELP nginx_http_request_duration_seconds HTTP request latency
# TYPE nginx_http_request_duration_seconds histogram
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="00.005"} 2487
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="00.010"} 2489
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="00.020"} 2489
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="00.030"} 2490
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="00.050"} 2490
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="00.075"} 2490
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="00.100"} 2490
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="00.200"} 2491
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="00.300"} 2491
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="00.400"} 2491
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="00.500"} 2491
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="00.750"} 2491
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="01.000"} 2491
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="01.500"} 2491
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="02.000"} 2491
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="03.000"} 2491
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="04.000"} 2491
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="05.000"} 2491
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="10.000"} 2491
nginx_http_request_duration_seconds_bucket{host="10.246.150.210",le="+Inf"} 2491
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.005"} 5
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.010"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.020"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.030"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.050"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.075"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.100"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.200"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.300"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.400"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.500"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.750"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="01.000"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="01.500"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="02.000"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="03.000"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="04.000"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="05.000"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="10.000"} 8
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="+Inf"} 8
nginx_http_request_duration_seconds_count{host="10.246.150.210"} 2491
nginx_http_request_duration_seconds_count{host="os.ecstme.org"} 8
nginx_http_request_duration_seconds_sum{host="10.246.150.210"} 0.18800020217896
nginx_http_request_duration_seconds_sum{host="os.ecstme.org"} 0.020999908447266
# HELP nginx_http_request_size_bytes Total size of incoming requests
# TYPE nginx_http_request_size_bytes counter
nginx_http_request_size_bytes 924984
# HELP nginx_http_requests_total Number of HTTP requests
# TYPE nginx_http_requests_total counter
nginx_http_requests_total{host="10.246.150.210",status="200"} 2488
nginx_http_requests_total{host="10.246.150.210",status="403"} 3
nginx_http_requests_total{host="os.ecstme.org",status="200"} 4
nginx_http_requests_total{host="os.ecstme.org",status="403"} 4
# HELP nginx_metric_errors_total Number of nginx-lua-prometheus errors
# TYPE nginx_metric_errors_total counter
nginx_metric_errors_total 0
```

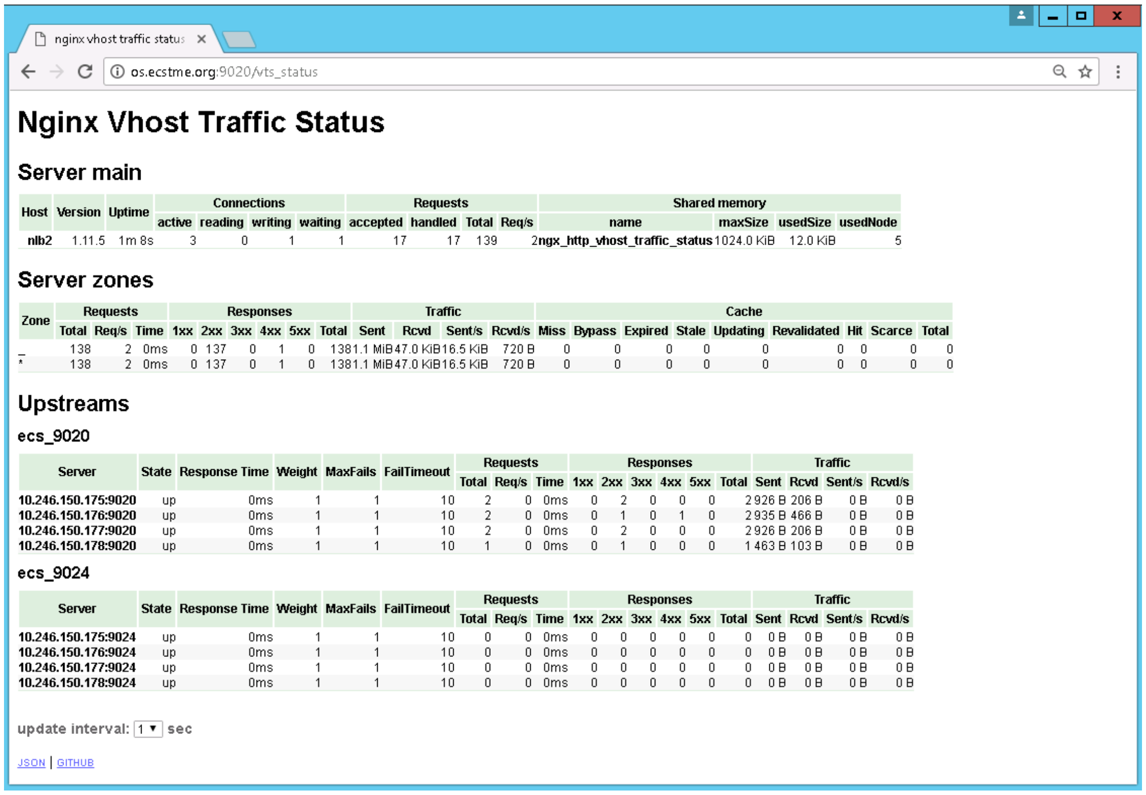
For HTTP traffic monitoring, the nginx-module-vts is used and vhost_traffic_status_zone is declared in the http context. This module is referenced within the location context within the server context as vts_status. Similarly for stream status, nginx-module-sts and nginx-module-stream-sts are utilized. The stream_server_traffic_status_zone is defined within the http context and server_traffic_status_zone is defined within stream context as illustrated in Figure 49.

Figure 49 - HTTP and Stream Monitoring Directives

```
http {

        ...
        ...

        stream_server_traffic_status_zone;
        vhost_traffic_status_zone;


        server {
                listen 9020
                    ...
                    ...

                 location /sts_status {
                     access_log off;
                     stream_server_traffic_status_display;
                     stream_server_traffic_status_display_format html;
                 }

                 location /vts_status {
                     access_log off;
                     vhost_traffic_status_display;
                     vhost_traffic_status_display_format html;
                 }

            ...
            ...

            }

}

stream {
        server_traffic_status_zone;
        ...
        ...
        ...
}
```

Figure 50 provides a sample view of the web page output for HTTP. As can be seen from the webpage, the number of connections and requests coming into the OpenResty web platform and the amount of shared memory being used. It also highlights the state, response time, number and sizes of HTTP requests, and response being sent to each of the proxied servers (ECS Nodes). Figure 51 shows a sample view of stream (TCP) traffic, going to backend servers (ECS Nodes). Similar traffic information is provided for the TCP traffic. In this example, the stream traffic is NFS traffic.

Figure 50 - Example View of HTTP Traffic Web Page



Nginx Vhost Traffic Status

Server main

Host	Version	Uptime	Connections				Requests				Shared memory			
			active	reading	writing	waiting	accepted	handled	Total	Req/s	name	maxSize	usedSize	usedNode
nlb2	1.11.5	1m 8s	3	0	1	1	17	17	139	2	ngx_http_vhost_traffic_status	1024.0 KiB	12.0 KiB	5

Server zones

Zone	Requests			Responses						Traffic				Cache								
	Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s	Miss	Bypass	Expired	Stale	Updating	Revalidated	Hit	Scarce	Total
_	138	2	0ms	0	137	0	1	0	138	1.1 MiB	47.0 KiB	16.5 KiB	720 B	0	0	0	0	0	0	0	0	0
*	138	2	0ms	0	137	0	1	0	138	1.1 MiB	47.0 KiB	16.5 KiB	720 B	0	0	0	0	0	0	0	0	0

Upstreams

ecs_9020

Server	State	Response Time	Weight	MaxFails	FailTimeout	Requests			Responses						Traffic			
						Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s
10.246.150.175:9020	up	0ms	1	1	10	2	0	0ms	0	2	0	0	0	2	926 B	206 B	0 B	0 B
10.246.150.176:9020	up	0ms	1	1	10	2	0	0ms	0	1	0	1	0	2	935 B	466 B	0 B	0 B
10.246.150.177:9020	up	0ms	1	1	10	2	0	0ms	0	2	0	0	0	2	926 B	206 B	0 B	0 B
10.246.150.178:9020	up	0ms	1	1	10	1	0	0ms	0	1	0	0	0	1	463 B	103 B	0 B	0 B

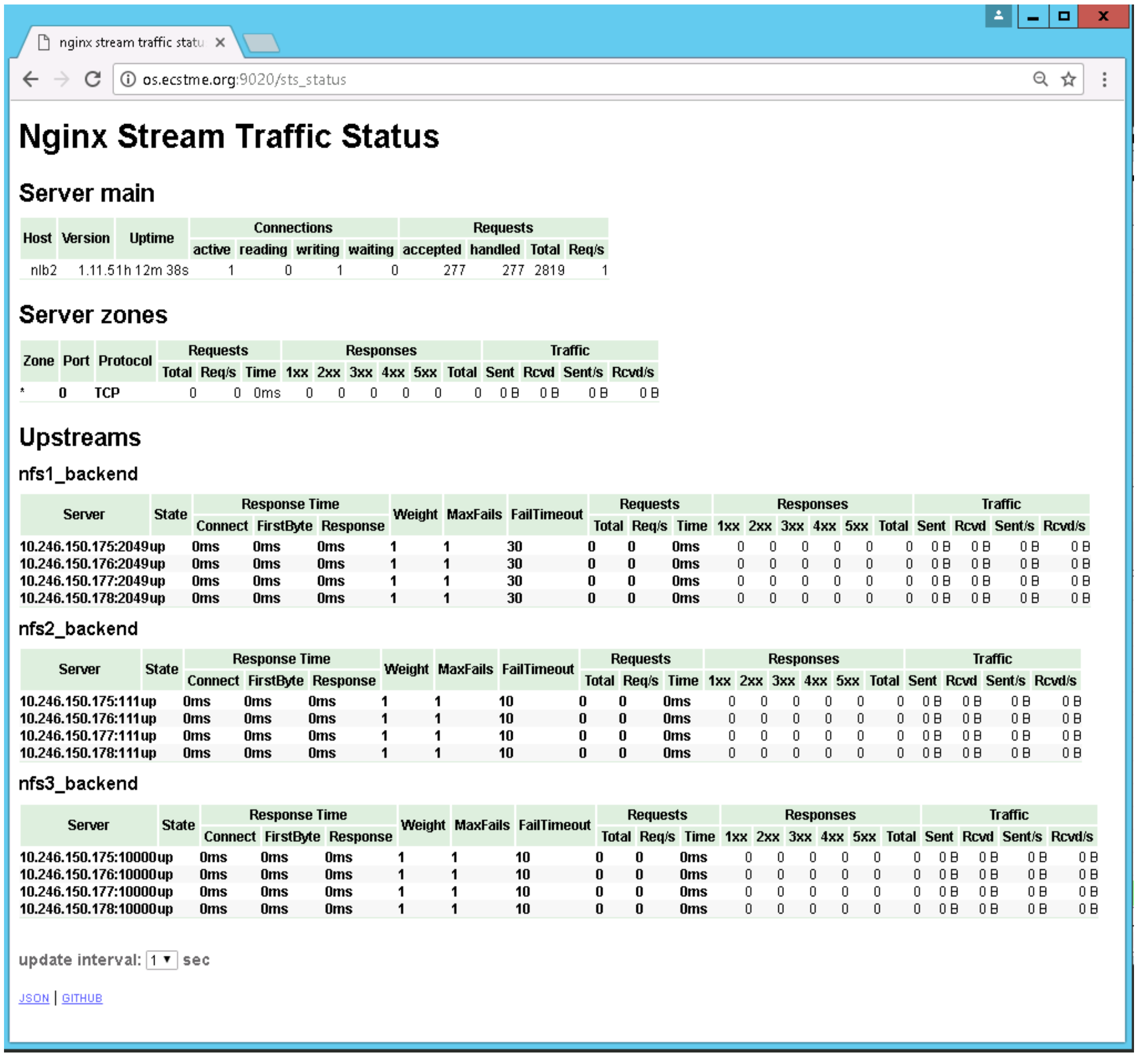
ecs_9024

Server	State	Response Time	Weight	MaxFails	FailTimeout	Requests			Responses						Traffic			
						Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s
10.246.150.175:9024	up	0ms	1	1	10	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B
10.246.150.176:9024	up	0ms	1	1	10	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B
10.246.150.177:9024	up	0ms	1	1	10	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B
10.246.150.178:9024	up	0ms	1	1	10	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B

update interval: 1 sec

JSON | GITHUB

Figure 51 - Example View of Stream Traffic Web Page



Nginx Stream Traffic Status

Server main

Host	Version	Uptime	Connections				Requests			
			active	reading	writing	waiting	accepted	handled	Total	Req/s
nlb2	1.11.5	1h 12m 38s	1	0	1	0	277	277	2819	1

Server zones

Zone	Port	Protocol	Requests			Responses						Traffic			
			Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s
*	0	TCP	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B

Upstreams

nfs1_backend

Server	State	Response Time			Weight	MaxFails	FailTimeout	Requests			Responses						Traffic			
		Connect	FirstByte	Response				Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s
10.246.150.175:2049	up	0ms	0ms	0ms	1	1	30	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B
10.246.150.176:2049	up	0ms	0ms	0ms	1	1	30	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B
10.246.150.177:2049	up	0ms	0ms	0ms	1	1	30	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B
10.246.150.178:2049	up	0ms	0ms	0ms	1	1	30	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B

nfs2_backend

Server	State	Response Time			Weight	MaxFails	FailTimeout	Requests			Responses						Traffic			
		Connect	FirstByte	Response				Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s
10.246.150.175:111	up	0ms	0ms	0ms	1	1	10	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B
10.246.150.176:111	up	0ms	0ms	0ms	1	1	10	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B
10.246.150.177:111	up	0ms	0ms	0ms	1	1	10	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B
10.246.150.178:111	up	0ms	0ms	0ms	1	1	10	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B

nfs3_backend

Server	State	Response Time			Weight	MaxFails	FailTimeout	Requests			Responses						Traffic			
		Connect	FirstByte	Response				Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s
10.246.150.175:10000	up	0ms	0ms	0ms	1	1	10	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B
10.246.150.176:10000	up	0ms	0ms	0ms	1	1	10	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B
10.246.150.177:10000	up	0ms	0ms	0ms	1	1	10	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B
10.246.150.178:10000	up	0ms	0ms	0ms	1	1	10	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B

update interval: 1 ▾ sec

JSON | GITHUB

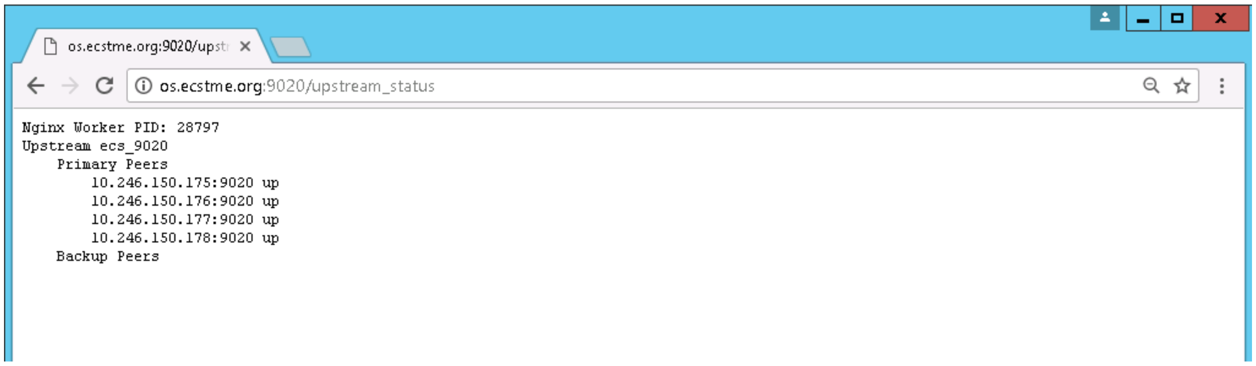
Also available are directives for exhibiting the status of the upstream servers (i.e. ECS Nodes), stream (TCP) health check status page, and NGINX. Figure 52 shows these directives which are also defined within the server context listening on port 9020.

Figure 52 - Health Status Directives for ECS Nodes, Stream and NGINX process.

```
http {

        ...
        ...

    stream_server_traffic_status_zone;
    vhost_traffic_status_zone;


server {
        listen 9020
            ...
            ...

        # status page for upstream servers
        location = /upstream_status {
            access_log off;
            default_type text/plain;
            content_by_lua_block {
                local hc = require "resty.upstream.healthcheck"
                ngx.say("Nginx Worker PID: ", ngx.worker.pid())
                ngx.print(hc.status_page())
            }
        }

        # stream health check status page
        location = /stream_upstream_status {
            access_log off;
            default_type text/plain;
            stream_check_status json;
        }

        # status page for nginx
        location = /nginx_status {
            access_log off;
            default_type text/plain;
            stub_status on;
        }

       ...
            ...

    }

}
```

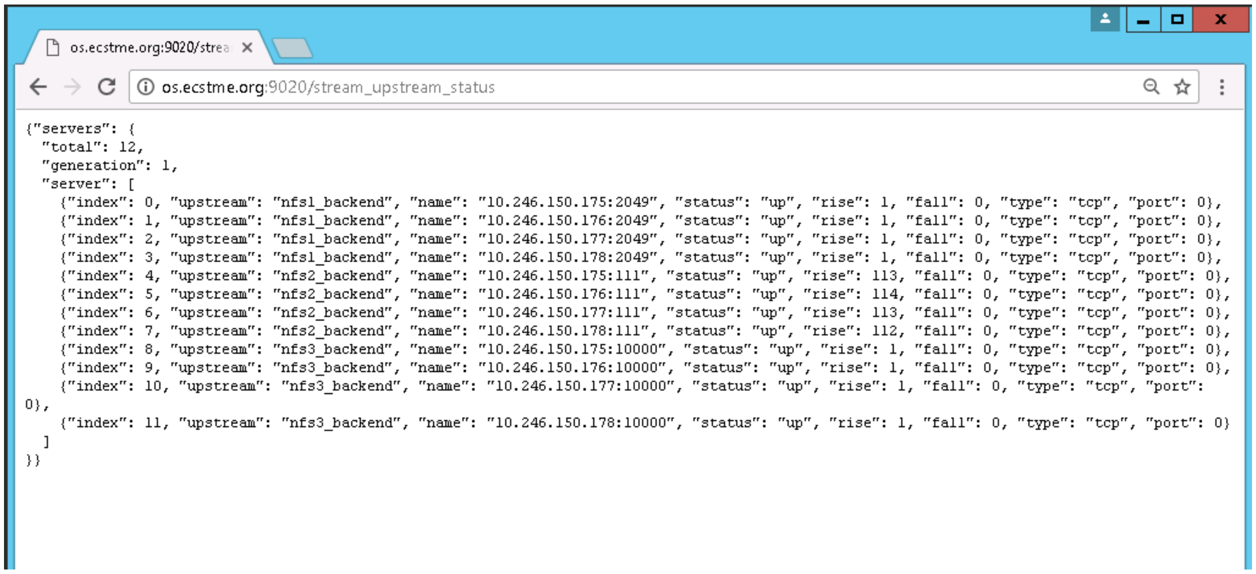
Figures 53, 54, and 55 provide example views of each of the web page output showing the ECS Nodes, stream, and NGINX status. Again, the status information is accessible via port 9020 and the “match” criteria specified in the location context, is specified on the URL for instance, http://os.ecstme.org:9020/upstream_status.

Figure 53 - Upstream Servers (ECS Nodes) Status



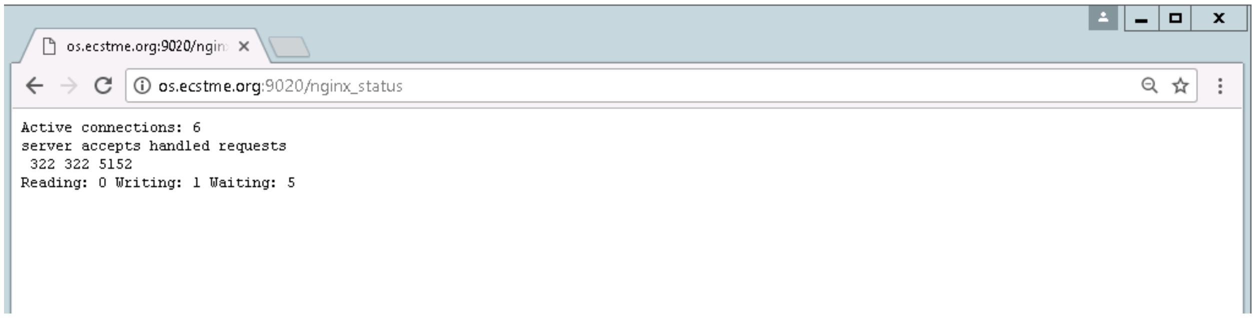
```
Nginx Worker PID: 28797
Upstream ecs_9020
    Primary Peers
        10.246.150.175:9020 up
        10.246.150.176:9020 up
        10.246.150.177:9020 up
        10.246.150.178:9020 up
    Backup Peers
```

Figure 54 - Stream Status (In this Example, NFS)



```
{"servers": {
  "total": 12,
  "generation": 1,
  "server": [
    {"index": 0, "upstream": "nfs1_backend", "name": "10.246.150.175:2049", "status": "up", "rise": 1, "fall": 0, "type": "tcp", "port": 0},
    {"index": 1, "upstream": "nfs1_backend", "name": "10.246.150.176:2049", "status": "up", "rise": 1, "fall": 0, "type": "tcp", "port": 0},
    {"index": 2, "upstream": "nfs1_backend", "name": "10.246.150.177:2049", "status": "up", "rise": 1, "fall": 0, "type": "tcp", "port": 0},
    {"index": 3, "upstream": "nfs1_backend", "name": "10.246.150.178:2049", "status": "up", "rise": 1, "fall": 0, "type": "tcp", "port": 0},
    {"index": 4, "upstream": "nfs2_backend", "name": "10.246.150.175:111", "status": "up", "rise": 113, "fall": 0, "type": "tcp", "port": 0},
    {"index": 5, "upstream": "nfs2_backend", "name": "10.246.150.176:111", "status": "up", "rise": 114, "fall": 0, "type": "tcp", "port": 0},
    {"index": 6, "upstream": "nfs2_backend", "name": "10.246.150.177:111", "status": "up", "rise": 113, "fall": 0, "type": "tcp", "port": 0},
    {"index": 7, "upstream": "nfs2_backend", "name": "10.246.150.178:111", "status": "up", "rise": 112, "fall": 0, "type": "tcp", "port": 0},
    {"index": 8, "upstream": "nfs3_backend", "name": "10.246.150.175:10000", "status": "up", "rise": 1, "fall": 0, "type": "tcp", "port": 0},
    {"index": 9, "upstream": "nfs3_backend", "name": "10.246.150.176:10000", "status": "up", "rise": 1, "fall": 0, "type": "tcp", "port": 0},
    {"index": 10, "upstream": "nfs3_backend", "name": "10.246.150.177:10000", "status": "up", "rise": 1, "fall": 0, "type": "tcp", "port":
0},
    {"index": 11, "upstream": "nfs3_backend", "name": "10.246.150.178:10000", "status": "up", "rise": 1, "fall": 0, "type": "tcp", "port": 0}
  ]
}}
```

Figure 55 - NGINX status

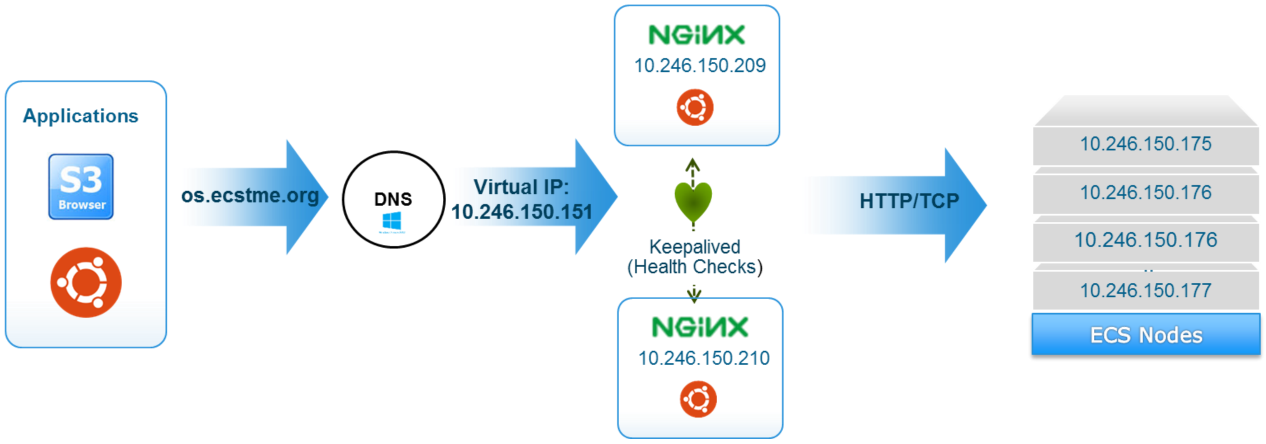


```
Active connections: 6
server accepts handled requests
 322 322 5152
Reading: 0 Writing: 1 Waiting: 5
```

6 Highly Available Example

The ECS with single OpenResty example in the previous section can be extended to add a second OpenResty to create a redundant OpenResty setup. In this example, another virtual machine (10.246.150.209) with Ubuntu 16.04 LTS operating system and OpenResty was configured the same way as previous example. The "keepalived" utility was installed on both OpenResty servers to do health checks between the two. The redundant OpenResty servers were configured in an active/passive mode. A virtual IP initially maps to the primary OpenResty server and if the primary server fails, then the virtual IP redirects to the IP of secondary OpenResty server until the primary comes up again. Figure 56 illustrates the setup for the redundant OpenResty load balancers environment described in this example.

Figure 56 - ECS with Redundant NGINX Setup Example



6.1 Virtual IP

On each of the load balancer servers, modify the *"net.ipv4.ip_nonlocal_bind"* to 1 in the kernel file /etc/sysctl.conf. This allows OpenResty to bind to a shared IP address which is 10.246.150.151 in this example. With an editor, such as "vi", add the line shown in Figure 57.

Figure 57 - Add to "/etc/sysctl.conf"

```
net.ipv4.ip_nonlocal_bind=1
```

Run "sysctl –p" command to have this setting take effect without the need to reboot as shown in Figure 58.

Figure 58 - Command to Enable the Setting Modified in /etc/sysctl.conf file.

```
# sysctl -p
```

6.2 Keepalived

The keepalived utility is a routing software package available on Linux. It is written in C and its main purpose is to provide health checks between systems. Install the keepalived utility on both load balancers. If Linux curl package is not already installed on the system, install this as well. Figure 59 provides the commands to install keepalived and curl.

Figure 59 - Example Command to Install keepalived

```
# apt-get update
# apt-get install keepalived
# apt-get install curl
```

In both load balancers, edit or create a keepalived.conf file in "/etc/keepalived" directory with entries as shown in Figure 60 and Figure 61. The difference between the two files is the priority value and state, where 101 and MASTER represent the primary and 100 and BACKUP is secondary. Also note that ens160 is set to the virtual IP. For more information on keepalived, refer to the keepalived website: http://www.keepalived.org/

Figure 60 - Example keepalived.conf for Primary OpenResty

```
vrrp_script check_slb {
        script "/usr/local/sbin/check_etm.sh" # sends HTTP request to check process and ECS
        interval 2                      # Does checks every two seconds
        weight 2                        # Add two points of prio if OK
        fall 1
        rise 1
        timeout 30
}

vrrp_instance vi1 {
        state MASTER
        interface ens160
        virtual_router_id 50
        priority 101                    # 101 is primary and 100 is secondary
        advert_int 1


        authentication {                # Protects VRRP against attacks
           auth_type PASS
           auth_pass password           # secret password shared between servers
        }

        virtual_ipaddress {
            10.246.150.151              # Virtual IP
        }

        track_script {
            check_slb
        }
}
```

Figure 61 - Example keepalived.conf for Secondary OpenResty

```
vrrp_script check_slb {
        script "/usr/local/sbin/check_etm.sh" # sends HTTP request to check process and ECS
        interval 2                      # Does checks every two seconds
        weight 2                        # Add two points of prio if OK
        fall 1
        rise 1
        timeout 30
}

vrrp_instance vi1 {
        state BACKUP
        interface ens160
        virtual_router_id 50
        priority 100                    # 101 is primary and 100 is secondary
        advert_int 1

        authentication {                     # Protects VRRP against attacks
            auth_type PASS
            auth_pass password               # secret password shared between servers
        }

        virtual_ipaddress {
            10.246.150.151              # Virtual IP
        }

        track_script {
            check_slb
        }
}
```

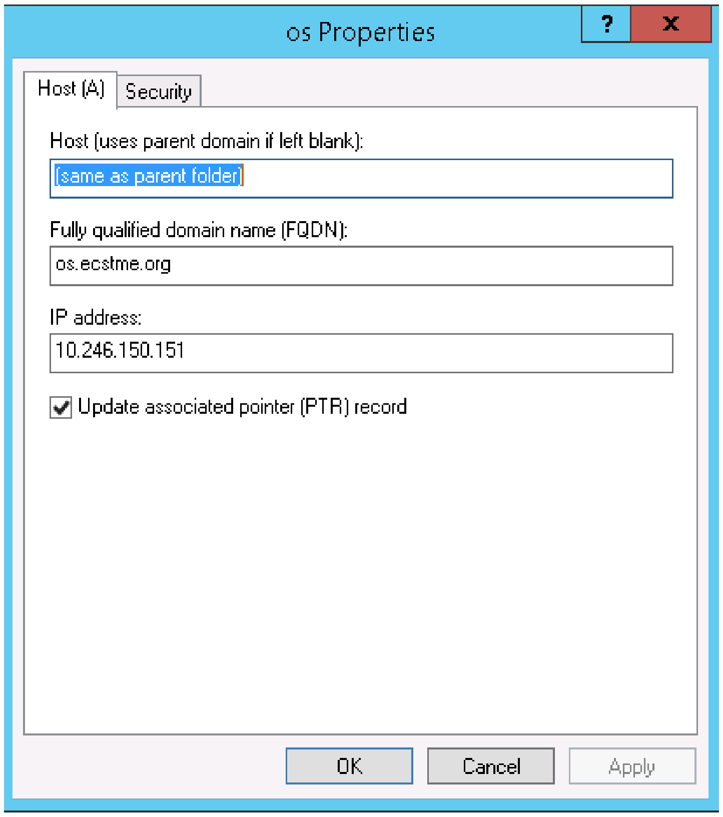
The *check_etm.sh* called in the *keepalived.conf* is illustrated in Figure 62. This script checks the health of the OpenResty servers and ECS nodes by sending an HTTP ping to port 9020. Place this file in */usr/local/sbin* directory and do a "chmod 755" to make this script executable.

Figure 62 - Script to Check Health of OpenResty Servers and ECS Nodes

```
#!/bin/bash
alive=`curl http://localhost:9020/?ping 2>/dev/null | grep "Data Node is Available" | wc -l`
if [ $alive -ge 1 ]; then
   exit 0
else
   exit 1
fi
```

On the redundant OpenResty server, copy the nginx.conf file, Lua scripts and the SSL certificate created on the other OpenResty server described in the previous example and place them in the install directory. For the SSL certificates, use DNS names in the SANs as opposed to IP addresses so the same certificate file can be used on both systems. Then modify the DNS as pictured in Figure 63 to point the os.ecstme.org A-record to the virtual IP defined in the keepalived.conf file, 10.246.150.151 in this example.

Figure 63 - DNS Record with Virtual IP



Host (uses parent domain if left blank):
(same as parent folder)

Fully qualified domain name (FQDN):
os.ecstme.org

IP address:
10.246.150.151

☑ Update associated pointer (PTR) record

Afterwards, start the keepalived service and restart OpenResty service on each server as illustrated in Figure 64.

Figure 64 - Command to Start keepalived Example

```
# service keepalived start

# /usr/local/openresty/bin/openresty -s stop
# /usr/local/openresty/bin/openresty
```

Then, check that the virtual IP is on the primary OpenResty server as highlighted in Figure 65.

Figure 65 - Virtual IP on OpenResty Server

```
# ip addr show ens160
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:0c:29:19:cb:d6 brd ff:ff:ff:ff:ff:ff
    inet 10.246.150.209/24 brd 10.246.150.255 scope global dynamic ens160
       valid_lft 438405sec preferred_lft 438405sec
    inet 10.246.150.151/32 scope global ens160
       valid_lft forever preferred_lft forever
    inet6 fe80::5279:f4e8:a3ef:cc44/64 scope link
       valid_lft forever preferred_lft forever
```

6.3 Validation

Start the S3 Browser to validate the setup. Since the S3 browser is utilizing the DNS entry name, “os.ecstme.org”, no additional modifications are needed. Then, validate the redundant setup by shutting down the primary OpenResty service by issuing a *“/usr/local/openresty/bin/openersty –s stop”.* Once the secondary server recognizes that the primary is down, the secondary picks up the Virtual IP as exemplified in Figure 66. Access to the objects on ECS should still be available via S3 Browser since the secondary load balancer is handling the requests. Similarly with the NFS client, the mount point should not be affected when the primary load balancer goes down.

Figure 66 - Virtual IP on Secondary Load Balancer Server

```
# ip addr show ens160
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:0c:29:e0:0d:a0 brd ff:ff:ff:ff:ff:ff
    inet 10.246.150.210/24 brd 10.246.150.255 scope global dynamic ens160
       valid_lft 467902sec preferred_lft 467902sec
    inet 10.246.150.151/32 scope global ens160
       valid_lft forever preferred_lft forever
    inet6 fe80::56a3:6d72:957b:b10f/64 scope link
       valid_lft forever preferred_lft forever
```

Syslog (/var/log/syslog) can also be viewed on each server. Figure 67 shows a sample log file listing the different states on the primary OpenResty server during a failure and after it came back online. When keepalived started, state was initially MASTER and then when *check_slb* failed, state transitioned to BACKUP and once it started up again, it re-claimed MASTER state. Similar state transitions can be observed on the secondary syslog file. The syslog file can also provide clues if an error occurred in startup of keepalived.

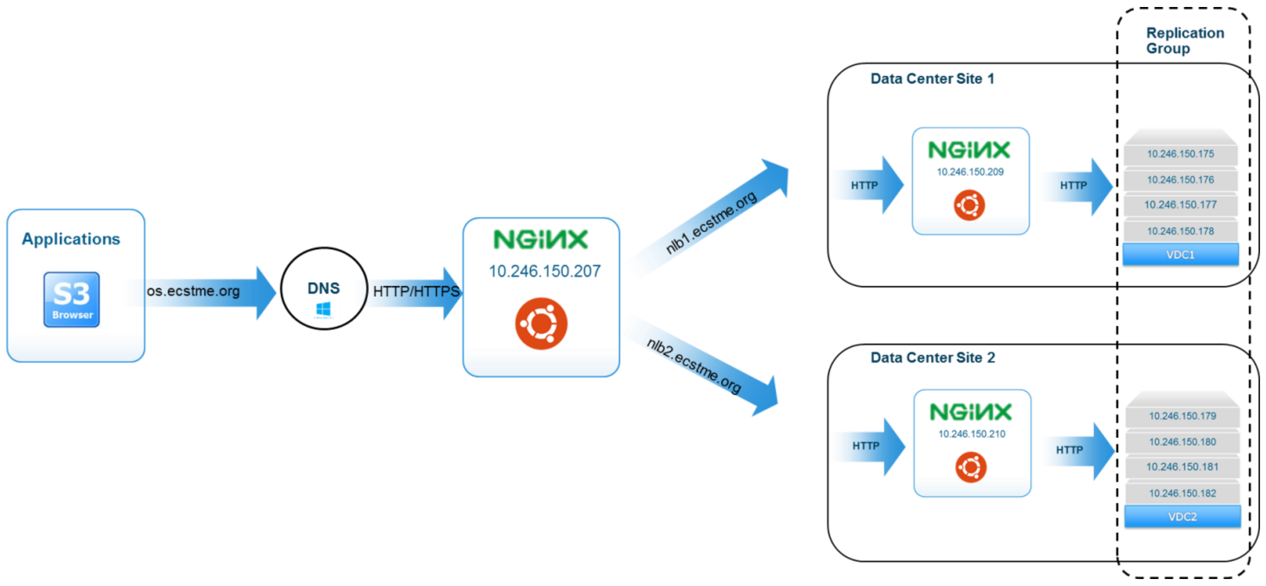
Figure 67 - Sample Output of Syslog - Keepalived Transition States

```
# tail /var/log/syslog

Jun  8 12:56:33 nlb1 Keepalived_vrrp[535]: VRRP_Instance(vi1) Transition to MASTER STATE
Jun  8 12:56:34 nlb1 Keepalived_vrrp[535]: VRRP_Instance(vi1) Entering MASTER STATE
Jun  8 12:56:34 nlb1 avahi-daemon[962]: Registering new address record for 10.246.150.151 on
ens160.IPv4.
Jun  8 12:56:42 nlb1 Keepalived_vrrp[535]: VRRP_Script(check_slb) succeeded
Jun  8 12:57:10 nlb1 Keepalived_vrrp[535]: VRRP_Script(check_slb) failed
Jun  8 12:57:12 nlb1 Keepalived_vrrp[535]: VRRP_Instance(vi1) Received higher prio advert
Jun  8 12:57:12 nlb1 Keepalived_vrrp[535]: VRRP_Instance(vi1) Entering BACKUP STATE
Jun  8 12:57:12 nlb1 avahi-daemon[962]: Withdrawing address record for 10.246.150.151 on ens160.
Jun  8 12:58:24 nlb1 Keepalived_vrrp[535]: VRRP_Script(check_slb) succeeded
Jun  8 12:58:24 nlb1 Keepalived_vrrp[535]: VRRP_Instance(vi1) forcing a new MASTER election
Jun  8 12:58:24 nlb1 Keepalived_vrrp[535]: VRRP_Instance(vi1) forcing a new MASTER election
Jun  8 12:58:25 nlb1 Keepalived_vrrp[535]: VRRP_Instance(vi1) Transition to MASTER STATE
Jun  8 12:58:26 nlb1 Keepalived_vrrp[535]: VRRP_Instance(vi1) Entering MASTER STATE
Jun  8 12:58:26 nlb1 avahi-daemon[962]: Registering new address record for 10.246.150.151 on
ens160.IPv4.
```

7 Global Load Balancing Example

When deploying ECS in a multi-site environment, global load balancing is recommended. This example describes a sample utilization of OpenResty as a global load balancier. As a best practice, a highly redundant environment is recommended for the global load balancer and on each site to prevent single point of failures. However, in this example (Figure 68), for simplicity, a single OpenResty is deployed as the global load balancer which points to two site load balancers. The site load balancers, forward requests to two different ECS systems which are geo-federated and configured as a single replication group. This example only provides details for handling S3 requests via the S3 Browser.

Figure 68 - Global Load Balancing Example



To optimize reads and writes for ECS in a geo-replicated environment, Dell EMC DevOps teams has developed a geo-pinning Lua script. They also have developed some health check scripts to validate the upstream servers. This section describes the scripts and configuration files to deploy OpenResty with two ECS in a global environment.

7.1 Setup

The same two virtual machines (10.246.150.209 and 10.246.150.210) with Ubuntu 16.04 LTS and OpenResty installed in previous examples are utilized in this example. Another 4-node ECS appliance was configured as VDC 2. A replication group was created containing VDC1 and VDC2. Also, an additional virtual machine was configured with IP 10.246.150.207 to act as a global load balancer. The virtual machine was installed with Ubuntu 16.04 LTS, OpenResty, libraries, add-on modules, and dependencies similarly to the other virtual machines. After setup of the global load balancer, install additional files (*http.lua and http_headrs.lua*) needed for the health checks for the upstream servers as shown in Figure 69. These Lua scripts, provides extra functions for protection against infinite loops as well as formatting headers, parsing, and other functions.

Figure 69 - Additional Lua Files

```
# cd /usr/local/openresty/lualib/resty
# wget https://raw.githubusercontent.com/pintsized/lua-resty-
http/master/lib/resty/http.lua
# wget https://raw.githubusercontent.com/pintsized/lua-resty-
http/master/lib/resty/http_headers.lua
# chmod 644 http.lua http_headers.lua

# mkdir /usr/local/openresty/nginx/lua
# cp /tmp/prometheus.lua /usr/local/openresty/nginx/lua
```

Then, copy the SSL certificates from the other virtual machines and the prometheus.lua script into the install directories. Commands in Figure 70 assumes that ssl scripts were remotely copied to the global OpenResty server and placed in /tmp. Also download the attached nginx-sample.zip files and place into /tmp and copy them to the install directory.

Figure 70 - Copy SSL Certificates and Lua Scripts to Global OpenResty Server

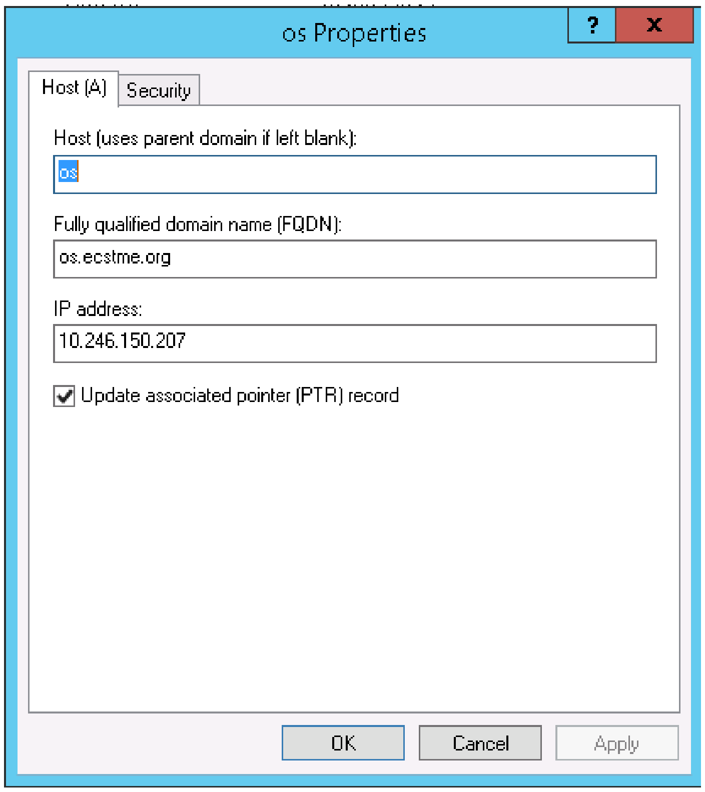
```
# mkdir /usr/local/openresty/nginx/ssl
# cp /tmp/ssl/combined.pem /usr/local/openresty/nginx/ssl
# cp /tmp/ssl/server.key /usr/local/openresty/nginx/ssl

# mkdir /usr/local/openresty/nginx/lua
# cp /tmp/prometheus.lua /usr/local/openresty/nginx/lua

# cd /tmp
# unzip nginx-sample.zip
# cp /tmp/nginx-sample/gslb/gslb.lua /usr/local/openresty/nginx
# cp /tmp/nginx-sample/gslb/lua/* /usr/local/openresty/nginx/lua
```

The files within the gslb directory of the zip file include the following:

- nginx.conf – example configuration file for a global environment which calls the Lua scripts for geo-pinning and healthchecking.
- gslb.lua – geo-pinning script to optimize reads and writes for ECS and distributes load between sites.
- healthcheck.lua – conducts health checks on ECS nodes
- prometheus.lua – metric collection
- strutils.lua – string utilities
- s3/urllib.lua – support URL parsing and style functions for the gslb.lua and healthcheck.lua scripts.

Next modify the DNS entry for *os.ecstme.org* and assign the IP of the global OpenResty server as shown in Figure71.

Figure 71 - DNS Assignment for Global Load Balancing.



Host (uses parent domain if left blank):
os

Fully qualified domain name (FQDN):
os.ecstme.org

IP address:
10.246.150.207

☑ Update associated pointer (PTR) record

7.2 OpenResty Configuration Files

The nginx.conf file at each site has an upstream context defined with the IPs of ECS nodes at each site. For this example, the upstream context named *ecs_9020* in the nginx.conf for OpenResty servers at each site has been modified as follows:

- nlb1.ecstme.org - IPs of ECS nodes for VDC 1 (10.246.150.175-10.246.150.178)
- nlb2.ecstme.org – IPs of ECS nodes for VDC2 (10.246.150.179-10.246.150.182)

As mentioned in previous the OpenResty single deployment example, a separate file can be created to contain the upstream servers and an “include” directive in the nginx.conf file can be added within the http context. Thus, for the global OpenResty server, an *upstreams.conf* file was created with the OpenResty DNS names of each site and an *“include upstreams.conf*” directive is defined in the nginx.conf before it is referenced in the server context as shown in Figure 72.

Figure 72 - Example of upstreams.conf File.

```
upstream vdc1 {
    server nlb1.ecstme.org:9020;
    keepalive 32;
}

upstream vdc2 {
    server nlb2.ecstme.org:9020;
    keepalive 32;
}
```

The nginx.conf file for the global OpenResty server has similar directives found in the previous example deployments such as events and http contexts, directives for optimizations, logging, and nested server and location contexts for proxying request to backend servers, and Lua directives. Since most of these directives were explained previously, only the directives that differ are highlighted in this section. The nginx.conf used in this example is attached to this whitepaper and a snippet is shown in Figure 73.

For health checks of upstream servers, an *init_by_lua_block* utilizes the **healthcheck.lua** script to run health checks on the upstream servers and ECS nodes. This init block is started when the nginx.conf file is loaded. The healthcheck.lua in the attached nginx-sample.zip has comments that describe each of its functions and variables. Sample functions inside healthcheck.lua script include:

- is_maintenance - parses the response body to determine whether the upstream ECS node is in MAINTENANCE_MODE
- get_load_factor – parses the response body to enumerate the LOAD_FACTOR (effectively the connection count) of the upstream ECS node.
- check_server - makes a request against the'?ping' resource on the upstream data node to retrieve the object service health by requesting its status including MAINTENANCE_MODE Status.
- check_pool - service loop which check status of individual data nodes within an upstream pool, using spawned threads
- do_check – does a call to check_pool to check each of the upstream servers.
- gen_peers_status_info – provides peer status for the pool of ECS nodes.

Reader is encouraged to read the healthcheck.lua script for additional details which contains self-explanatory comments. Check the error.log file in /usr/local/openresty/nginx/logs for errors in the spawning of this health check.

The server context listens on the 9020 (HTTP) and 9021 (HTTPS) ports for S3 requests. For HTTPS, SSL is terminated at the global load balancing server and forwarded to the upstream server using HTTP. Since the name of the global load balancer has been modified to be os.ecstme.org, certificates generated in previous examples are copied to the global load balancer. A Lua script, **gslb.lua**, is embedded within the load balancer configuration file to implement the geo-pinning algorithm. The geo-pinning script is explained in the next section. SSL termination can also occur at the site level load balancer instead of at the global level, if desired.

Figure 73 - Example Snippet of nginx.conf for Global Deployment Example

```
worker_processes  1;
pid /var/run/nginx.pid;
events {
    worker_connections 4096;
    use epoll;
    multi_accept on;
}
http {
    lua_package_path "${prefix}/lua/?.lua;;";
    lua_code_cache on;
    lua_socket_log_errors off;
    #lua_need_request_body off;
    lua_shared_dict stats 32k;
    include upstreams.conf;
    init_worker_by_lua_block {
        local hc = require "healthcheck"

        local ok, err = hc.run_checker{
            interval = 10,
            timeout = 60,
            max_fails = 1
        }

        if not ok then
            ngx.log(ngx.ERR, "failed to spawn health checker: ", err)
            return
        end
    }
    server {
        listen 9020;
        location / {
            set $root_host "ecstme.org"; # set the actual domain name here
            rewrite_by_lua_file 'gslb.lua';
            proxy_pass http://$proxyto;
        }
      }
    server {
        listen 9021 ssl;
        ssl on;
        ssl_certificate /usr/local/openresty/nginx/ssl/combined.pem;
        ssl_certificate_key /usr/local/openresty/nginx/ssl/server.key;
        ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers HIGH:!aNULL:!MD5;
        ssl_prefer_server_ciphers on;
        ssl_session_cache shared:SSL:32m;
        ssl_session_timeout 1d;

        location / {
            set $root_host "ecstme.org"; # set the actual domain name here
            set $proxyto '';
            rewrite_by_lua_file 'gslb.lua';
            proxy_pass http://$proxyto;
        }
    }
}
```

7.2.1 Geo-Pinning (gslb.lua)

The **gslb.lua** implements a hash algorithm (modulo to “n” sites) referred to as “geo-pinning” to distribute the creation of objects across sites for storage efficiency and direct reads and updates to site owning object for performance. In ECS architecture, the site that originated the creation of object is the site owner of object. In order to maintain strong consistency a read request first checks with site owning the object for consistency of data at requesting site. If a request comes to site that does not own the object, response times are prolonged in order to check with owning site first and fetch data if data is not current in cache of remote site. With geo-pinning, the read goes directly to site owner of object to process requests and thus improving read performance. The geo-pinning logic is as follows:

1. Derive hash ID from the URI and headers of a request. If it is a bucket request, the bucket is the ID. If it is an object request, the key is the ID.
2. Calculate the hash string from the hash ID using SHA1 algorithm.
3. Get first six HEX number of hash string and conduct a modulo operation based on the number of VDCs

It has several dependent Lua scripts and modules that include:

- ngx.upstream – NGINX upstream module included as part of OpenResty.
- s3.urllib – URL libraries, part of nginx-samples.zip file attached.
- resty.string – OpenResty string libaries
- config – local configuration directives, included as part of the nginx-samples.zip file attached.

Reader is encouraged to read through the gslb.lua script and dependent scripts such as s3.urllib and config attached to this whitepaper. The scripts contain self-explanatory comments.

7.2.2 Monitoring

For metric collection, as shown in Figure 74, the *init_by_lua_block* and *log_by_lua* are defined to register and log the metrics collected using the Prometheus library as was previously explained in the previous section.

Figure 74 - Metric Collection Directives.

```
server {
       listen 9020;
      lua_shared_dict prometheus_metrics 10M;
      init_by_lua_block {
          prometheus = require("prometheus").init("prometheus_metrics")
          metric_requests = prometheus:counter(
            "nginx_http_requests_total", "Number of HTTP requests", {"host", "status"})
          metric_latency = prometheus:histogram(
            "nginx_http_request_duration_seconds", "HTTP request latency", {"host"})
          metric_bytes = prometheus:counter(
            "nginx_http_request_size_bytes", "Total size of incoming requests")
          metric_connections = prometheus:gauge(
            "nginx_http_connections", "Number of HTTP connections", {"state"})
       }
       log_by_lua_block {
           metric_requests:inc(1, {host, ngx.var.status})
            metric_bytes:inc(tonumber(ngx.var.request_length))
            metric_latency:observe(ngx.now() - ngx.req.start_time(), {ngx.var.host})
        }
}
```

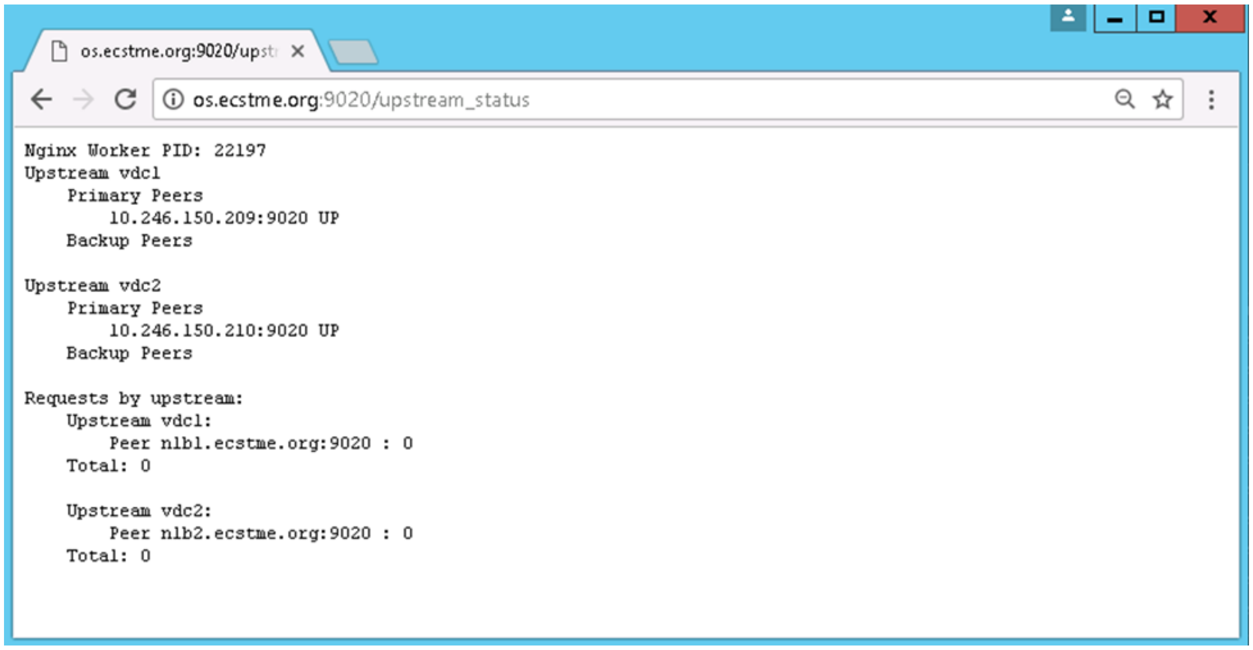
For exposing via a webpage, the location context contains the monitoring directives as illustrated in Figure 75. The heatlhcheck.lua is also used for providing the status page for upstream servers.

Figure 75 - Monitoring Directives within Server Context

```
server {
      listen 9020;
       # status page for upstream servers
       location = /upstream_status {
           access_log off;
           default_type text/plain;
           content_by_lua_block {
               local hc = require "healthcheck"
               ngx.say("Nginx Worker PID: ", ngx.worker.pid())
               ngx.print(hc.status_page())
               local upstream = require 'ngx.upstream'
               local us = upstream.get_upstreams()
               local stats = ngx.shared.stats
               ngx.say()
               ngx.say("Requests by upstream:")
               for _, u in ipairs(us) do
                   ngx.say("    Upstream ", u, ":")
                   local sum = 0
                   local srvs = upstream.get_servers(u)
                   for _, srv in ipairs(srvs) do
                       local cnt = stats:get(srv.name)
                       if not cnt then
                           cnt = 0
                       end
                       ngx.say("        Peer ", srv.name, " : ", cnt)
                       sum = sum + cnt
                   end
                   ngx.say("    Total: ", sum)
                   ngx.say()
               end
           }
       }
       # status page for nginx
       location = /nginx_status {
           access_log off;
          default_type text/plain;
           stub_status on;
       }
       location /metrics {
           access_log off;

           content_by_lua_block {
               metric_connections:set(ngx.var.connections_reading, {"reading"})
               metric_connections:set(ngx.var.connections_waiting, {"waiting"})
               metric_connections:set(ngx.var.connections_writing, {"writing"})
               prometheus:collect()
           }
       }
   }
```

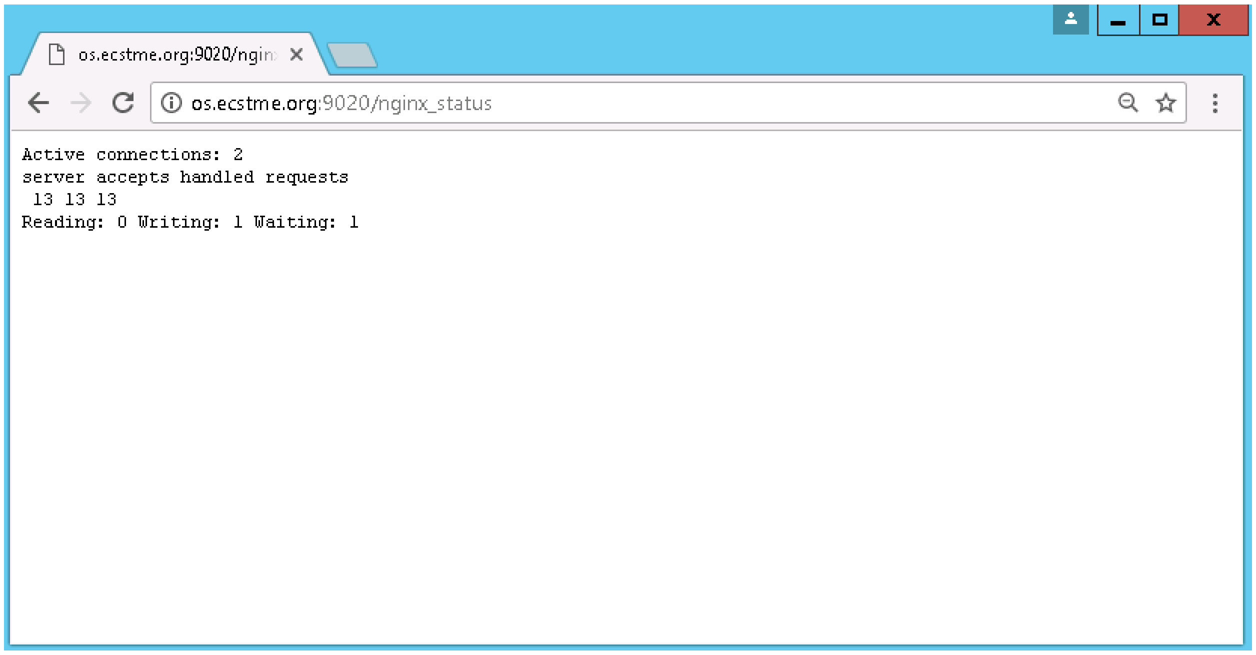
Sample views of the kind of information provided by the monitoring directives are shown in Figures 76, 77, and 78.

Figure 76 - Sample View of Upstream Servers Status



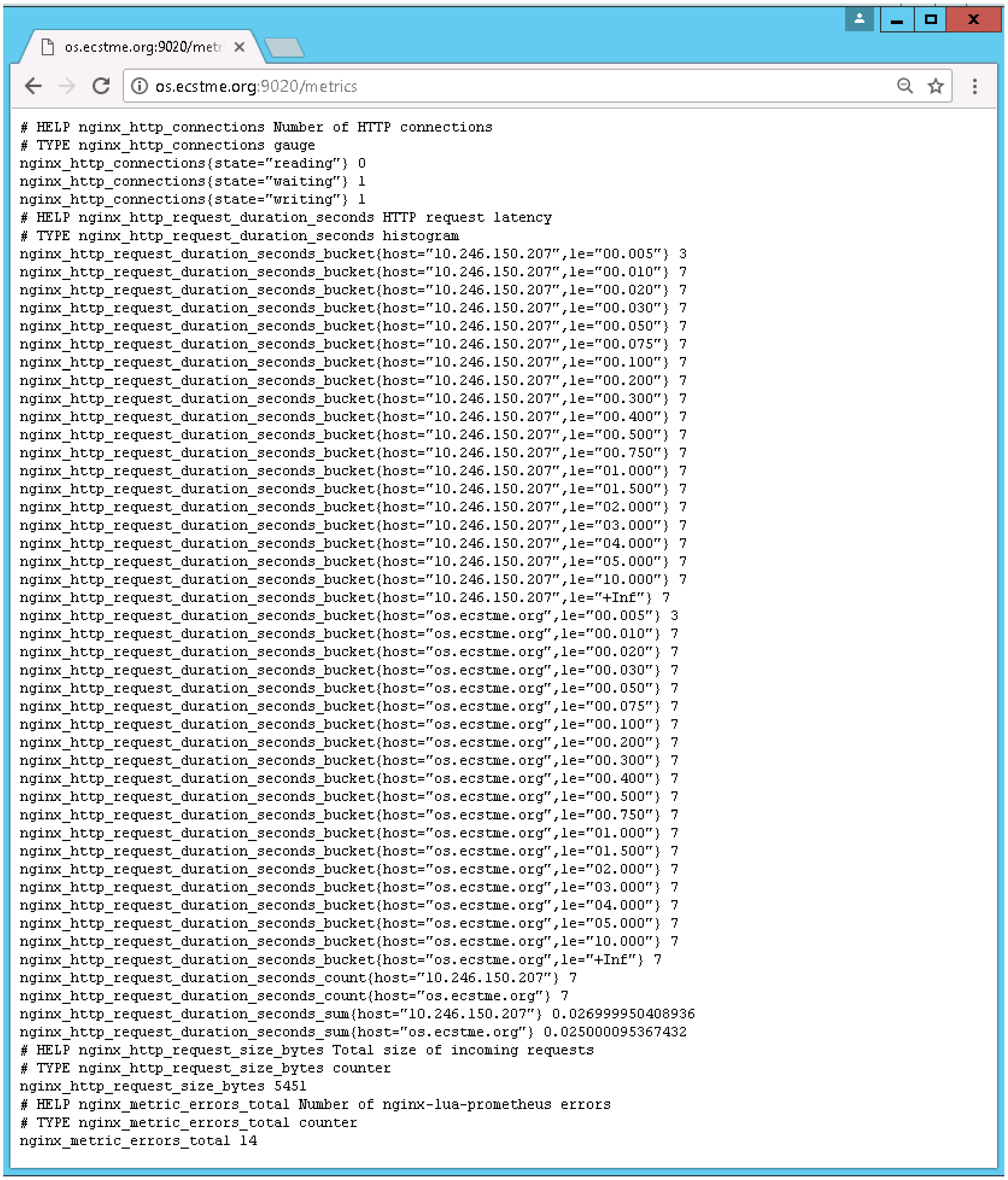
```
Nginx Worker PID: 22197
Upstream vdc1
    Primary Peers
        10.246.150.209:9020 UP
    Backup Peers

Upstream vdc2
    Primary Peers
        10.246.150.210:9020 UP
    Backup Peers

Requests by upstream:
    Upstream vdc1:
        Peer nlb1.ecstme.org:9020 : 0
    Total: 0

    Upstream vdc2:
        Peer nlb2.ecstme.org:9020 : 0
    Total: 0
```

Figure 77 – Sample View of NGINX status



```
Active connections: 2
server accepts handled requests
 13 13 13
Reading: 0 Writing: 1 Waiting: 1
```

Figure 78 - Sample View of Metrics Collected

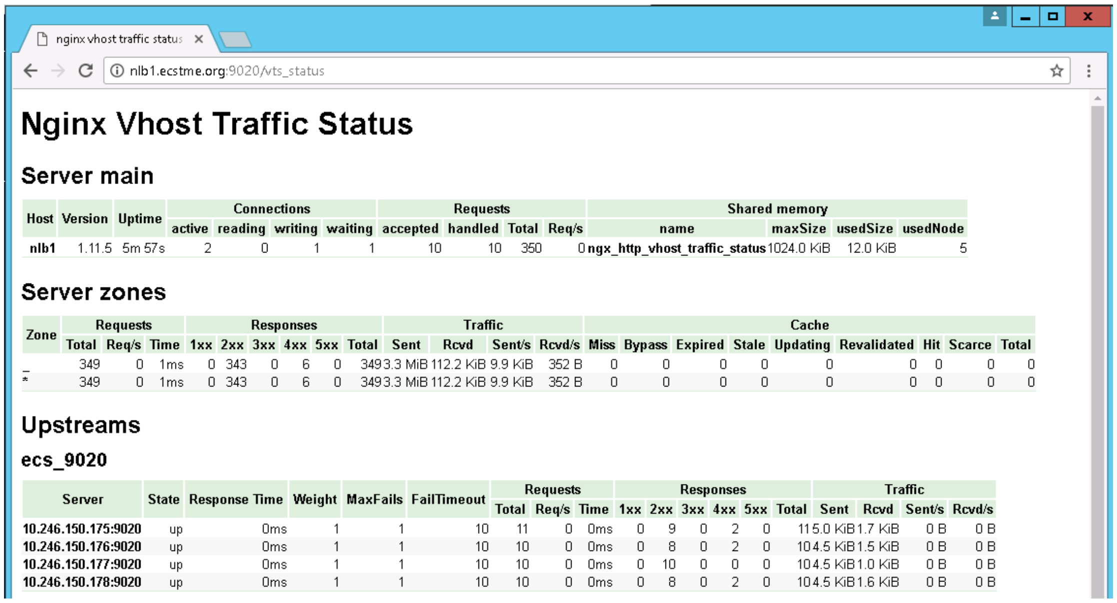


```
# HELP nginx_http_connections Number of HTTP connections
# TYPE nginx_http_connections gauge
nginx_http_connections{state="reading"} 0
nginx_http_connections{state="waiting"} 1
nginx_http_connections{state="writing"} 1
# HELP nginx_http_request_duration_seconds HTTP request latency
# TYPE nginx_http_request_duration_seconds histogram
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="00.005"} 3
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="00.010"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="00.020"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="00.030"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="00.050"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="00.075"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="00.100"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="00.200"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="00.300"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="00.400"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="00.500"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="00.750"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="01.000"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="01.500"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="02.000"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="03.000"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="04.000"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="05.000"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="10.000"} 7
nginx_http_request_duration_seconds_bucket{host="10.246.150.207",le="+Inf"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.005"} 3
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.010"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.020"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.030"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.050"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.075"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.100"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.200"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.300"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.400"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.500"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="00.750"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="01.000"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="01.500"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="02.000"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="03.000"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="04.000"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="05.000"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="10.000"} 7
nginx_http_request_duration_seconds_bucket{host="os.ecstme.org",le="+Inf"} 7
nginx_http_request_duration_seconds_count{host="10.246.150.207"} 7
nginx_http_request_duration_seconds_count{host="os.ecstme.org"} 7
nginx_http_request_duration_seconds_sum{host="10.246.150.207"} 0.026999950408936
nginx_http_request_duration_seconds_sum{host="os.ecstme.org"} 0.025000095367432
# HELP nginx_http_request_size_bytes Total size of incoming requests
# TYPE nginx_http_request_size_bytes counter
nginx_http_request_size_bytes 5451
# HELP nginx_metric_errors_total Number of nginx-lua-prometheus errors
# TYPE nginx_metric_errors_total counter
nginx_metric_errors_total 14
```

7.3 Validation

As in previous examples, the S3 browser can also be used to validate this global load balancing example. The settings in the previous examples should still apply since the DNS name of the global load balancer used in this example has been modified to be os.ecstme.org.   Also by viewing the status pages per the monitoring directives on each of the site load balancers (nlb1 and nlb2), it can provide information that traffic is in fact being distributed among sites. Figures 79 and 80 provide examples of the *vts_status* page for nlb1.ecstme.org and nlb2.ecstme.org to validate distribution.

Figure 79 - HTTP S3 traffic for NLB1



Nginx Vhost Traffic Status

Server main

Host	Version	Uptime	Connections				Requests				Shared memory			
			active	reading	writing	waiting	accepted	handled	Total	Req/s	name	maxSize	usedSize	usedNode
nlb1	1.11.5	5m 57s	2	0	1	1	10	10	350	0	ngx_http_vhost_traffic_status	1024.0 KiB	12.0 KiB	5

Server zones

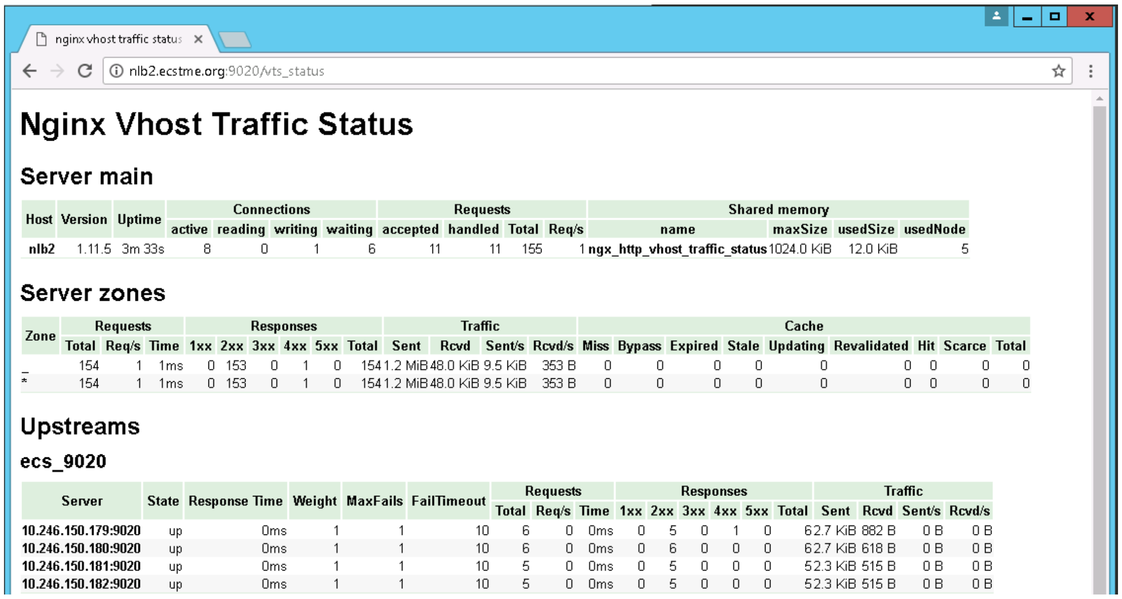
Zone	Requests			Responses						Traffic				Cache								
	Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s	Miss	Bypass	Expired	Stale	Updating	Revalidated	Hit	Scarce	Total
_	349	0	1ms	0	343	0	6	0	349	3.3 MiB	112.2 KiB	9.9 KiB	352 B	0	0	0	0	0	0	0	0	0
*	349	0	1ms	0	343	0	6	0	349	3.3 MiB	112.2 KiB	9.9 KiB	352 B	0	0	0	0	0	0	0	0	0

Upstreams

ecs_9020

Server	State	Response Time	Weight	MaxFails	FailTimeout	Requests			Responses						Traffic			
						Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s
10.246.150.175:9020	up	0ms	1	1	10	11	0	0ms	0	9	0	2	0	11	5.0 KiB	1.7 KiB	0 B	0 B
10.246.150.176:9020	up	0ms	1	1	10	10	0	0ms	0	8	0	2	0	10	4.5 KiB	1.5 KiB	0 B	0 B
10.246.150.177:9020	up	0ms	1	1	10	10	0	0ms	0	10	0	0	0	10	4.5 KiB	1.0 KiB	0 B	0 B
10.246.150.178:9020	up	0ms	1	1	10	10	0	0ms	0	8	0	2	0	10	4.5 KiB	1.6 KiB	0 B	0 B

Figure 80 - HTTP S3 traffic for NLB2



Nginx Vhost Traffic Status

Server main

Host	Version	Uptime	Connections				Requests				Shared memory			
			active	reading	writing	waiting	accepted	handled	Total	Req/s	name	maxSize	usedSize	usedNode
nlb2	1.11.5	3m 33s	8	0	1	6	11	11	155	1	ngx_http_vhost_traffic_status	1024.0 KiB	12.0 KiB	5

Server zones

Zone	Requests			Responses						Traffic				Cache								
	Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s	Miss	Bypass	Expired	Stale	Updating	Revalidated	Hit	Scarce	Total
_	154	1	1ms	0	153	0	1	0	154	1.2 MiB	48.0 KiB	9.5 KiB	353 B	0	0	0	0	0	0	0	0	0
*	154	1	1ms	0	153	0	1	0	154	1.2 MiB	48.0 KiB	9.5 KiB	353 B	0	0	0	0	0	0	0	0	0

Upstreams

ecs_9020

Server	State	Response Time	Weight	MaxFails	FailTimeout	Requests			Responses						Traffic			
						Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s
10.246.150.179:9020	up	0ms	1	1	10	6	0	0ms	0	5	0	1	0	6	2.7 KiB	882 B	0 B	0 B
10.246.150.180:9020	up	0ms	1	1	10	6	0	0ms	0	6	0	0	0	6	2.7 KiB	618 B	0 B	0 B
10.246.150.181:9020	up	0ms	1	1	10	5	0	0ms	0	5	0	0	0	5	2.3 KiB	515 B	0 B	0 B
10.246.150.182:9020	up	0ms	1	1	10	5	0	0ms	0	5	0	0	0	5	2.3 KiB	515 B	0 B	0 B

8 Best Practices

Utilizing a load balancer with ECS is highly recommended. Highlights of some the best practices when deploying with ECS include:

- Do not use a load balancer for CAS traffic since the Centera SDK has a built-in load balancer in software and cannot function without direct access to all nodes.
- Use the load balancer to terminate SSL connections to reduce the load on the ECS Nodes
- If SSL termination is required on ECS nodes itself, then use Layer 4 (tcp) to pass through the SSL traffic to ECS nodes for handling. The certificates would need to be installed on the ECS nodes and not on the load balancer.
- Use redundant load balancers to prevent single point of failure.
- For NFS, use only the high available functionality of the load balancer.
- Enable web monitoring for OpenResty to monitor traffic as described in this whitepaper.
- When deploying three or more ECS sites, employ a global load balancing mechanism such as the gslb.lua described in the Global Deployment example to distribute load across sites and to take advantage of the storage efficiency by ECS XOR. The gslb.lua script also optimizes the local object read hit rate in a global deployment.

9 Conclusions

OpenResty provides a low-cost option for customers desiring to utilize a load balancer with ECS. By using the OpenResty version, the functionality of standard NGINX is enhanced and expanded. It integrates libraries and add-on modules that allow for enhanced monitoring and customization of NGINX with ECS. Sample Lua scripts were provided and the reader can modify and extend these samples based on specific requirements. Examples and best practices were described in this whitepaper to provide guidance and a reference to architects interested in deploying NGINX (OpenResty) with ECS.

A Attachment – NGINX-SAMPLE.ZIP

The nginx-samples.zip contains all the sample configuration files and Lua scripts used in the examples described in this whitepaper. In order to download the attachment, your PDF reader must have permissions to be able to open a zip file. For instance, if using Adobe Acrobat Reader, please refer to this website on how to modify reader permissions to allow opening of zip files attached to PDF files:

- https://www.adobe.com/devnet-docs/acrobatetk/tools/AppSec/attachments.html

The nginx-samples.zip (Figure 81) file contains the following:

Figure 81 - Contents of nginx-samples.zip file

```
# unzip -l nginx-sample.zip
Archive:  nginx-sample.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2017-06-12 10:30   nginx-sample/
        0  2017-06-12 08:21   nginx-sample/gslb/
        0  2017-06-12 08:21   nginx-sample/gslb/lua/
      129  2017-06-12 08:21   nginx-sample/gslb/lua/config.lua
     7172  2017-06-12 08:21   nginx-sample/gslb/lua/resty_healthchecker.diff
      881  2017-06-12 08:21   nginx-sample/gslb/lua/strutils.lua
     4372  2017-06-12 08:21   nginx-sample/gslb/lua/gslb.lua
        0  2017-06-12 08:21   nginx-sample/gslb/lua/s3/
    16151  2017-06-12 08:21   nginx-sample/gslb/lua/s3/urllib.lua
     9601  2017-06-12 08:21   nginx-sample/gslb/lua/healthcheck.lua
    16063  2017-06-12 08:21   nginx-sample/gslb/lua/prometheus.lua
     4372  2017-06-12 08:21   nginx-sample/gslb/gslb.lua
        0  2017-06-12 08:21   nginx-sample/gslb/conf/
      140  2017-06-12 08:21   nginx-sample/gslb/conf/upstreams.conf
     5756  2017-06-12 08:21   nginx-sample/gslb/conf/nginx.conf
        0  2017-06-12 08:21   nginx-sample/slb/
        0  2017-06-12 08:21   nginx-sample/slb/lua/
    16063  2017-06-12 07:09   nginx-sample/slb/lua/prometheus.lua
        0  2017-06-12 08:39   nginx-sample/slb/conf/
     8224  2017-06-12 08:35   nginx-sample/slb/conf/nginx.conf
        0  2017-06-12 10:30   nginx-sample/install/
     3127  2017-06-12 09:43   nginx-sample/install/OpenRestyInstall.txt
        0  2017-06-12 10:29   nginx-sample/install/patches/
    27119  2017-06-12 10:29   nginx-sample/install/patches/stream_upstream_module.patch
     4861  2017-06-12 10:29   nginx-sample/install/patches/patch-1.11.x.patch
        0  2017-06-12 07:17   nginx-sample/keepalived/
      792  2017-06-12 07:14   nginx-sample/keepalived/keepalived.conf.Master
      159  2017-06-12 07:15   nginx-sample/keepalived/check_etm.sh
      789  2017-06-12 07:17   nginx-sample/keepalived/keepalived.conf.Backup
---------                     -------
   125771                     29 files
```

A.1 Install

All the commands to install OpenResty are described in ***install/install.txt*** file. It is best to perform these commands one by one so that if there are errors, it can be corrected before moving to the next step. The patches directory contains the two files needed to patch NGINX and ngx_stream_upstream_check_module-master described in the examples.

A.2 SLB

For **nginx-samples/slb** directory, steps to setup:

1. Place the **nginx.conf** in the OpenResty install directory for nginx (i.e. /usr/local/openresty/nginx/conf).
2. Modify the upstream ecs_9020 and ecs_9024 with the IPs of the ECS nodes for S3 and Swift respectively.
3. For NFS, modify the IPs in the stream context with the IPs of the ECS nodes.
4. Create a "lua" directory in the OpenResty install directory for nginx (i.e. /usr/local/openresty/nginx/lua)
5. Place the **prometheus.lua** script in the lua directory created in step 4.

A.3 Keepalived

If deploying a highly available OpenResty, there are two keepalived configuration files and check_etm.sh script in nginx-samples/keepalived directory. The steps to install include:

1. After install of keepalived on both servers hosting OpenResty, place the keepalived.conf.Master in /etc/keepalived directory on server that acts as master and rename it keepalived.conf. Place the keepalived.conf.Backup in /etc/keepalived directory on server that acts as backup.
2. Copy check_etm.sh script in /usr/local/sbin directory on both servers.

A.4 GSLB

If deploying a global environment setup, the nginx-samples/gslb directory contains the files needed. Setup includes:

1. Place the nginx.conf and upstreams.conf files in the OpenResty install directory for nginx configuration (i.e. /usr/local/openresty/nginx/conf).
2. Modify the upstreams.conf to include the IPs of the backend OpenResty servers.
3. Copy gslb.lua to the Openresty install directory for nginx (i.e. /usr/local/openresty/nginx).
4. Copy the lua directory or contents to the OpenResty install directory for nginx (i.e. /usr/local/openresty/nginx/lua).

B Troubleshooting

This section provides some common issues that can be encountered during build, install and deployment and provide some tips and logs that can be referenced.

B.1 Build Issues

Using “cut and paste” to imitate the examples in this whitepaper can cause some extra configure options to not be included. So, although the build may succeed, some modules or options may not be included such that compilation may fail with an error. Please re-run the configure command with correct options. Another reason is if the libraries needed were not installed properly. Check that all the libraries such as build_essentials and libncurses are installed with the latest versions. In most cases, the error message produced during the build can provide clues on the cause.

B.2 Logs

There are several logs available that provide additional clues on issues encountered. There are NGINX log files available to review located in the install directory (i.e. /usr/local/openresty/nginx/logs). There are two logs files available:

- access.log – logs information relating to client requests
- error.log – logs errors based on severity level. In the examples, a “warn” severity level is specified which indicates messages from error to warn are logged.

The syslog file located /var/log also can provide error messages relating to what is occurring on the host running Openresty.

B.3 Firewall Issues

A firewall is one of the main issues that can cause error in connecting to ECS. Certain ports need to be open in order to communicate to ECS. For instance, for Swift, ports 9024 and 9025 need to be opened. Check firewall settings on the host running OpenResty to ensure the ports for ECS are open.

C ECS Dedicated Cloud

The Dell EMC ECS DevOps team has developed the Lua scripts described in this document. It is meant mostly for reference and up to reader to modify based on their requirements. If deploying these scripts in an ECS Dedicated Cloud environment, there may be other directives defined to handle replication traffic in order to also provide the capability to shape or re-route replication traffic at the load balancer level. In this scenario, the replication endpoints on the ECS nodes point to the server load balancers. Refer to the ECS Dedicated Cloud Technical Overview whitepaper and your ECS Dedicated Cloud account team for more information on the ECS Dedicated Cloud.

D Technical Support and Resources

Dell EMC support is focused on meeting customer needs with proven services and support.

Dell EMC Community is an online technical community where IT professionals have access to numerous resources for Dell EMC software, hardware and services.

Storage Solutions Technical Documents on Dell TechCenter provide expertise that helps to ensure customer success on Dell EMC Storage platforms.

D.1 Related resources

OpenResty, NGINX, SSL

- OpenResty documentation
  - https://openresty.org/en/
- NGINX
  - https://nginx.org/
- OpenSSL documentation
  - https://www.openssl.org/docs/apps/openssl.html

ECS APIs and SDKs

- ECS Rest API
  - http://www.emc.com/techpubs/api/ecs/v3-0-0-0/index.htm
- Dell EMC Data Services – Atmos and S3 SDK
  - https://github.com/emcvipr/dataservices-sdk-java/releases
- Data Access Guide
  - https://support.emc.com/docu79368_ECS_3.0_Data_Access_Guide.pdf?language=en_US&language=en_US

ECS Product Documentation

- ECS product documentation at support site or the community links:
  - https://support.emc.com/products/37254_ECS-Appliance-/Documentation/
  - https://community.emc.com/docs/DOC-53956
- ECS Architecture and Overview
  - http://www.emc.com/collateral/white-papers/h14071-ecs-architectural-guide-wp.pdf
- ECS Networking and Best Practices
  - http://www.emc.com/collateral/white-paper/h15718-ecs-networking-bp-wp.pdf

ECS Community

- Getting Started with ECS SDKs
  - https://community.emc.com/docs/DOC-27910

.