Nauta Installation and Configuration

Administration Guide, Open Source Beta Version

Document Revision 1.0: January 2019

Document Revision History

Document Revision Number	Date	Comments
1.0	January 2019	Initial Release to Open Source, Beta Document Version

Contents

Figures

Tables

Nauta Overview

This Nauta Installation, Configuration, and Administration guide provides step-by-step instructions for installing and configuring Nauta. This guide also provides an overview of Nauta requirements, such as configuration management and operating system requirements. This guide is divided into three main sections: Preinstallation Information, Installation Tasks, and User and Account Management.

Note: For instructions on configuring the Nauta client, refer to the Nauta User Guide.

The Nauta software provides a multi-user, distributed computing environment for running deep learning model training experiments. Results of experiments, can be viewed and monitored using a command line interface (CLI), web UI and/or TensorBoard*. You can use existing data sets, use your own data, or downloaded data from online sources, and create public or private folders to make collaboration among teams easier.

Nauta runs using the industry leading Kubernetes* and Docker* platform for scalability and ease of management. Templates are available (and customizable) on the platform to take the complexities out of creating and running single and multi-node deep learning training experiments without all the systems overhead and scripting needed with standard container environments.

01

Preinstallation Information

Nauta System Software Components Requisite

As shown in Table 1, these are the packages included in the install package.

List of Included External Software Components

Table 1 shows the software package that includes the external software components for Nauta.

Table 1: Nauta External Software Components

Name	Version	Project link
addon-resizer	1.7	http://k8s.gcr.io/addon-resizer
dashboard	1.8.3	https://k8s.gcr.io/kubernetes-dashboard-amd64
defaultbackend	1.4	https://github.com/kubernetes/ingress-nginx
dnsmasq-nanny	1.14.8	https://github.com/kubernetes/dns
dns-sidecar	1.14.8	https://github.com/kubernetes/dns
elasticsearch	6.2.3	https://github.com/elastic/elasticsearch
etcd	3.3.9	https://github.com/coreos/etcd
flannel	0.9.1	https://github.com/coreos/flannel
fluentd	0.12	https://www.fluentd.org
heapster	1.4.2	https://github.com/kubernetes/heapster
ingress	0.14.0	http://quay.io/kubernetes-ingress-controller/nginx-ingress-controller
kubectl	1.10.11	https://github.com/kubernetes/kubernetes/tree/master/pkg/kubectl
kube-dns	1.14.8	https://github.com/kubernetes/dns
kube-proxy	1.10.6	http://gcr.io/google-containers/kube-proxy-amd64
mkl-dnn	0.14	https://github.com/intel/mkl-dnn
nginx	1.14.0	https://www.nginx.com
pause	3.1	http://gcr.io/google-containers/pause-amd64
redsocks	0.5	https://github.com/darkk/redsocks
registry	2.6.2	https://github.com/docker/distribution
tensorflow	1.9.0	https://github.com/tensorflow/tensorflow
tiller	2.9.1	https://github.com/helm/helm

Key Components

The Nauta complete bundle (including installation scripts) contains the following:

- Vanilla Kubernetes cluster on top of provisioned Hardware and Operating System-level Software.
- Nauta components (containerized components, their configuration, integration method, and so on.)

The Nauta software components are optimized for AI containers with Nauta-optimized libraries.

How to Build the Nauta Package

Dependencies and Requirements

There are dependencies and requirements (such as having Ubuntu installed) to build Nauta, if you wish to perform your own build. You *must* download the tarball from the Git repository before starting the build process.

Ubuntu* 16.04 LTS or 18.04 LTS

- python3-venv
- python3-dev
- binutils
- build-essential
- pip3
- tar
- pigz
- Docker*

Note: During platform build process, TensorFlow* is checked by Horovod* installation inside Docker* container.

Your Build Machine

You *must* meet all TensorFlow* requirements, and in particular your CPU must have support for SSE instructions. To make sure SSE is available on a CPU, call cat `/proc/cpuinfo.sse` `sse2` should be listed in `flags` field.

The build process hardware should have at least 12 GB of RAM and at least 100GB of space for temporary containers, registries, and so on. The size of final tar.gz file is ~5GB.

The build process requires access to Docker commands. Remember to add the user to Docker group by using: `sudo usermod -aG docker [user]` if he/she has not been added there yet.

For more information, refer to the Post-install Docker guide.

Proxy Settings

Utilize `http_proxy, https_proxy` and `no_proxy` environment variables, if you are behind a proxy. For `no_proxy` include 127.0.0.1 and localhost.

If proxy issues occur during the build process, it is recommended that you configure a transparent proxy (for example, a redsocks-based solution).

Build

In the main directory of Nauta repository invoke:

```
make k8s-installer-build
```

During the build process, Docker images related to Nauta are prepared. After a successful build, the `tar.gz` file can be found in the `tools/.workspace` directory.

Connecting to the Internet

To build Nauta package you need internet connection so that you can untar the tarball, configure your proxy settings, DNS settings, and so on.

During the build process, Docker images related to Nauta are prepared. After a successful build, the `tar.gz` file can be found in the `tools/.workspace` directory.

Tarball—Output of the Build

An artifact's associated name is: `nauta-{version}-{build-id}.tar.gz` (for example: `nauta-1.0.0-190110100005.tar.gz`). In addition, the package contains docs, images, config files, and ansible tasks. To install the platform, follow this installation guide with a prepared `tar.gz` artifact.

To interact with an installed Nauta platform, the nctl client is required. For more information refer to the chapter, *Client Installation and Configuration* in section: *How to Build Nauta CLI* in the Nauta User Guide.

Nauta Installer System Requirements

When installing Nauta it should be installed on a separate machine (your installer machine), as Nauta requires a separate machine to run installer.

Nauta Supported Operating Systems

Nauta supports the following Operating Systems:

- Red Hat* Enterprise Linux* 7.5 or CentOS* 7.5
- Ubuntu* 16.04

Red Hat Enterprise Linux 7.5

Required on system, software requirements:

- sshpass (when password authentication is used)

CentOS 7.5

Required on system, software requirements:

- sshpass (when password authentication is used)

Ubuntu 16.04

Required on system, software requirements:

- python* 3.5
- apt required packages:
    - python3-pip
    - build-essential
    - libffi-dev
    - libssl-dev
    - sshpass

Target Host Requirements

For the target host, install Nauta on bare metal install only with Red Hat Enterprise Linux 7.5 (can be preinstalled). In addition, the following is also required on the target host.

- Configured access to master host over SSH.
    - This is configured access from your *Installer Machine* to your *Target Host* (master).
- Full network connectivity between target hosts is required. In addition, Installer connectivity is only required to the master node.

Red Hat Enterprise Linux 7.5

Required packages include:

- byacc
- cifs-utils
- ebtables
- ethtool
- gcc
- gcc-c++
- git
- iproute
- iptables >= 1.4.21
- libcgroup
- libcgroup-devel
- libcgroup-tools
- libffi-devel
- libseccomp-devel
- libtool-ltdl-devel
- make
- nfs-utils
- openssh
- openssh-clients
- openssl
- openssl-devel
- policycoreutils-python
- python
- python-backports
- python-backports-ssl_match_hostname
- python-devel
- python-ipaddress
- python-setuptools
- rsync
- selinux-policy >= 3.13.1-23
- selinux-policy-base >= 3.13.1-102
- selinux-policy-targeted >= 3.13.1-102
- socat
- systemd-libs
- util-linux
- vim
- wget

Valid Repositories

If the operating system is installed and configured with a valid repository that contains the required packages, an Administrator *does not* need to install the packages manually. However, if the repository is not valid the Installer will attempt to install the package automatically. If this fails an error message is displayed.

Inventory File Configuration Tasks

Configuration files are *YAML* files* used to define configuration settings so that will be used with Nauta. As part of the Inventory configuration tasks you need to create a YAML file and modify the file for the system inventory.

Inventory File Example

Below is an example of Inventory File and shows one Master Node and five Worker nodes. Your configuration may differ from the example shown.

Note: Wrap around formatting is shown in the example. However, in the YAML file it appears as a single line string.

```
[master]
master-0 ansible_ssh_host=192.168.100 ansible_ssh_user=root
ansible_ssh_pass=YourPassword internal_interface=em2 data_interface=em2
external_interface=em3 local_data_device=/dev/sdb1


[worker]
worker-0 ansible_ssh_host= 192.168.100.61 ansible_ssh_user=root
ansible_ssh_pass=YourPassword internal_interface=p3p1 data_interface=p3p1
external_interface=em1


worker-1 ansible_ssh_host= 192.168.100.55 ansible_ssh_user=root
ansible_ssh_pass=YourPassword internal_interface=p3p1 data_interface=p3p1
external_interface=em1


worker-3 ansible_ssh_host= 192.168.100.106 ansible_ssh_user=root ansible_ssh_
pass=YourPassword internal_interface=p3p1 data_interface=p3p1
external_interface=em1


worker-4 ansible_ssh_host= 192.168.100.107 ansible_ssh_user=root ansible_ssh_
pass=YourPassword internal_interface=p3p1 data_interface=p3p1
external_interface=em1
```

Inventory File Structure

The file contains two sections:

- **Master:** Contains a description of a master node. This section *must* contain *exactly* one row.
- **Worker:** Contains descriptions of workers. Each worker is described in one row. In this section, it can have one or many rows depending on a structure of a cluster.

The format of rows with node's descriptions is as follows (all parts of these rows are separated with spaces):

```
[NAME] [VARIABLES]
```

NAME: This is a name of a node. It should be a hostname of a real node. However, this *is not* a requirement.

VARIABLES: Variables describing the node in the following format:

```
[VARIABLE_NAME]=[VARIABLE_VALUE]
```

Installation Inventory Variable

Table 2 shows the Basic SSH parameters for RedHat* Ansible* that must be used to determine how to gain root access on remote hosts.

Table 2: Installation Inventory Variables

Name	Description
ansible_ssh_user	The user name *must be* the same for master and worker nodes
ansible_ssh_pass	The SSH password to use
ansible_ssh_host	The name (DNS name) or IP Address of the host to connect to. This is the first entry on the row and this is the alias information. It assigns the first full substring and uses an alias for that node. **Note:** This *should not* be the alias that was defined in the Inventory file example and *do not* enter the alias name assigned in the previous row.
ansible_ssh_port	The SSH port number, if not defined 22 is used. However, if this variable is present it must have an assigned value. If you do not enter a value in this row for the file, the default value is 22.
ansible_ssh_private_key_file	This is a *Private Key* file used by SSH.

Note: If an Administrator decides to choose something other than root for Ansible SSH user, the user *must be* configured in sudoers file with NOPASSWD option. Refer to Ansible Inventory documentation for more information.

Inventory File Structure Variables

Table 3 shows the list of Inventory file variables used in your Inventory file.

Table 3: Inventory File Structure Variables

Variable Name	Description	Type	Default	Required	Used When	Acceptable Values
internal_interface	This is used for internal communication between Kubernetes processes and pods.	string	none	true	Always for both for master and worker nodes	This is the interface name
local_data_device	This device is used for Nauta internal data and NFS data in case of local NFS*	string	none	true	Used with master nodes	This is the path to block device
local_data_path	This is used as the mountpoint for local_data_device	string	local_data_path is optional	false	This is used with master nodes	This is the absolute path where data is located in file system
data_interface	This is used for data transfer. This is the same type of variable as internal interface	string	none	true	Always for both for master and worker nodes	This is the interface name
external_interface	This is used for external network communication and is an *Ethernet Connection*.	string	none	true	Always for both for master and worker nodes	This is the interface name

Configuration File Tasks

Nauta configuration tasks include configuring the proxies, and DNS servers and other configuration tasks.

Note: In the examples shown, **Bold Green** indicates parameter name and ***Bold Italic Blue*** indicates exemplary parameter value.

Example Configuration File and Options

Proxy Configuration

Description: This is the Proxy setting for internal applications and, this proxy configuration is a dictionary.

Default value: {}

- **Note:** Instances of curly brackets { } indicates that this is an empty dictionary.

Example Configuration File

```
proxy:
  http_proxy: http://<your proxy address and port>
  ftp_proxy: http://<your proxy address and port>
  https_proxy: http://<your proxy address and port>
  no_proxy: <localhost, any other addresses>, 10.0.0.1,localhost,.nauta
  HTTP_PROXY: http://proxy-chain.intel.com:911
  FTP_PROXY: http://<your proxy address and port>
  HTTPS_PROXY: http://<your proxy address and port>
  NO_PROXY: .<localhost, any other addresses>, 10.0.0.1,localhost,.nauta
```

Note: Proxy addresses should be replaced by a specific value by a client.

dns_servers

Description: This is a list of DNS servers used for resolution: a max of three entries.

Default value: []

- **Note:** Instances of squares brackets [ ] indicates that this is an empty list.

```
dns_servers:
  - 10.102.60.20
  - 10.102.60.30
```

dns_search_domains

Description: This is a domain used as part of a domain search list.

Default value: []

```
dns_search_domains:
  - example.com
```

domain

Description: This is the *internal* domain name.

Default value: nauta

```
domain: nauta
```

nodes_domain

Description: Internal subdomain for infrastructure. Full domain for an infrastructure is:

```
<nodes_domain>.<domain>
```

Default value: infra

Note: The *domain* setting is for top domain. Together with *nodes_domain* it creates a full zone domain. For example:

```
domain: "nauta"
nodes_domain: "infra"
```

This results in full domain `"infra.nauta"`.

```
nodes_domain: lab007
```

k8s_domain

Description: This is the internal subdomain for Kubernetes resources. Full domain for infrastructure is:

```
k8s_domain: kubernetes
```

Default value: kubernetes

```
k8s_domain: kubernetes-001
```

kubernetes_pod_subnet

Description: This is the Network Range for Kubernetes pods.

Default Value: 10.3.0.0/16

```
kubernetes_pod_subnet: 10.3.0.0/16
```

kubernetes_svc_subnet

Description: This is the Network Range for Kubernetes services.

Default Value: 10.4.0.0/16

```
kubernetes_svc_subnet: 10.4.0.0/16
```

insecure_registries

Description: This is a list of insecure Docker registries added to configuration.

Default value: []

Note: This refers to Docker registries only.

```
insecure_registries: - my.company.registry:9876
```

enabled_plugins

Description: This is a list of enabled Yum* plugins.

Default value: []

```
enabled_plugins: - presto
```

disabled_plugins

Description: This is a list of disabled Yum plugins.

Default value: []

```
disabled_plugins: - presto
```

use_system_enabled_plugins

Description: This defines if Yum should use system-enabled plugins.

Default value: False

```
use_system_enabled_plugins: False
```

enabled_repos

Description: This is a list of enabled repositories, and is used for external dependencies installation.

Default value: []

```
enabled_repos: - rhel-local
```

disabled_repos

Description: This is a list of disabled repositories, and is used for external dependencies installation.

Default value: []

```
disabled_repos: - rhel
```

use_system_enabled_repos

Description: This defines if the default system repositories should be enabled in external dependencies installation.

Default value: True

```
use_system_enabled_repos: True
```

input_nfs

Description: Definition of *input* NFS mounts for Samba. By default, internal NFS provisioner is used.

Default value: {}

Fields

- **path:** NFS path to mount
- **server:** NFS server

output_nfs

Description: This is the definition of *output* NFS mounts for Samba. By default, internal NFS provisioner is used.

Default value: {}

Fields

- **path:** NFS path to mount
- **server:** NFS server

```
nauta_configuration:
  input_nfs:
    path: "{{ nfs_base_path }}/input"
    server: "{{ nfs_server }}"
  output_nfs:
    path: "{{ nfs_base_path }}/output"
    server: "{{ nfs_server }}"
```

Optional Features: Redsocks and NFS

Either Redsocks or NFS is installed and configured during the installation process. By default, Redsocks is *not* installed; however, NFS is installed by default.

This is a map of enabled features. Redsocks *is not* enabled during the installation, as the default is set to false (shown in the example below). Therefore, if you want to install Redsocks you need to set the feature switch to True.

Caution: After the installation should you decide you want to install Redsocks, you will need to redo the entire installation to include Redsocks and set the feature switch to True. It *cannot* be changed to False in your config file after the initial install. Redsocks and NFS are independent of each other, so use judgment when initially setting these feature switches.

```
features:
  nfs: True
  redsocks: False
```

Features: Network File System (NFS) and Redsocks

Nauta features include NFS and Redsocks*. To configure either NFS or Redsocks, you *must* enable either feature and configure feature options.

Network File System Overview

The Network File System, or NFS allows a system to share directories and files with others over a network. The advantage of using NFS is that end-users as well as programs can access files on remote systems in the same way as local files. In addition, NFS uses less disk space, as it can store data on a single machine while remaining accessible to others over a network.

Redsocks Configuration

Redsocks configuration is an optional part of the installer; therefore, it might apply only to limited number of installations.

Features List

There are two possible values: true and false.

NFS: default: True

Redsocks: default: disabled

How to Enable Features

Additional features can be enabled using features object in configuration, as shown below.

```
features:
  redsocks: False
```

Feature Plugin Configuration

Configuration for features should be placed under `features_config`.

```
features:
  redsocks: True
features_config:
  redsocks:
    IP: 10.217.247.236
    Port: 1080
```

Redsocks Configuration Parameters

IP

Description: This is the IP address of Socks5 proxy.

```
Required: True
```

Port

Description: This is the port address of Socks5 proxy.

```
Required: True
```

Interfaces

Description: Comma-separated list of interfaces from which traffic should be managed by RedSocks.

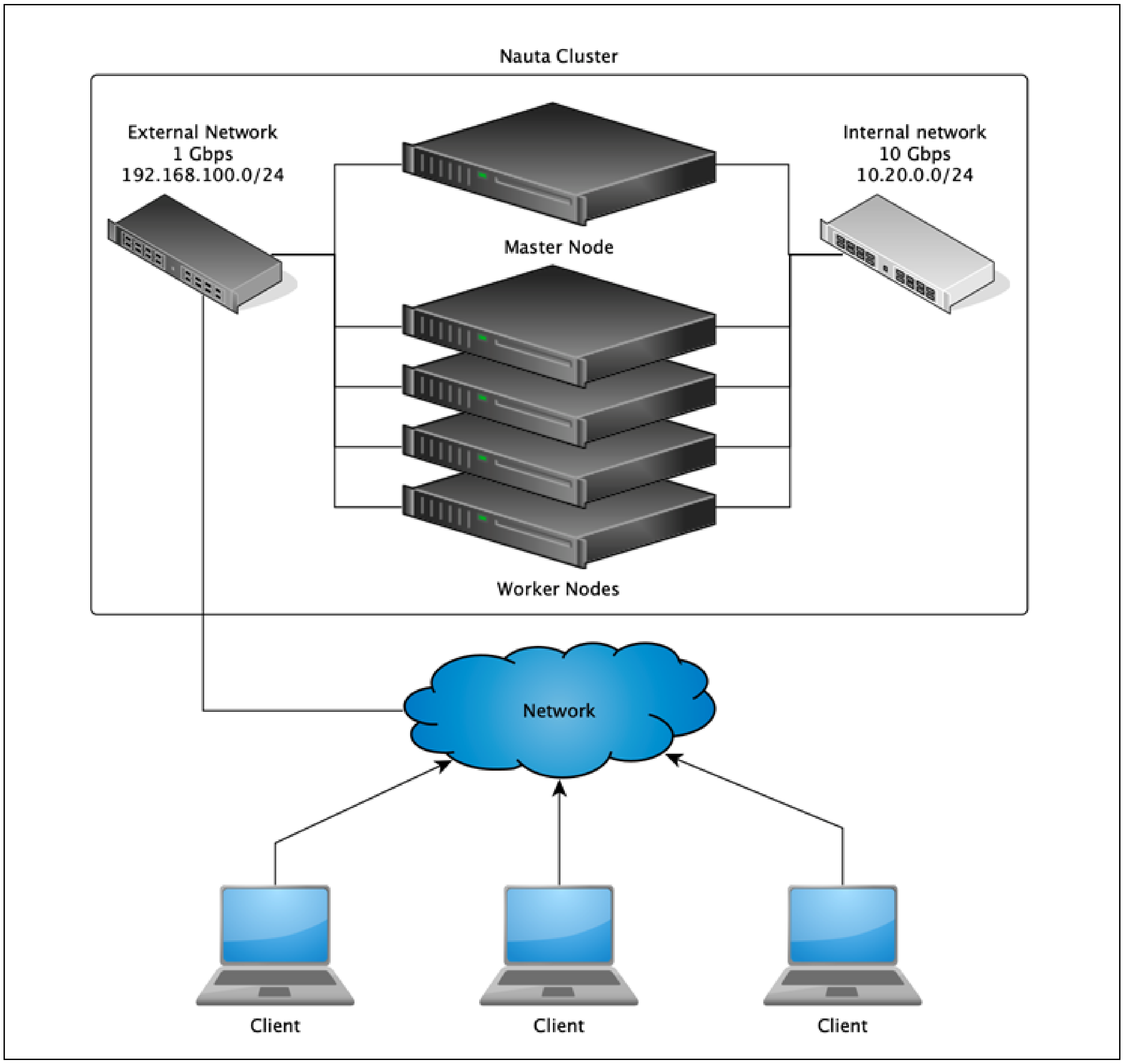
Description: This is the IP address of Socks5 proxy.

```
Required: False
Default: cni0
```

Networking Configuration Example

Figure 1, page 24 shows an example Nauta Networking Diagram. While you can build a cluster with 1 machine to run all the examples it is suggested to utilize at least 4 worker nodes (as shown in the example). The worker nodes should run Red Hat Enterprise Linux 7.5. All interfaces (both external and internal) are Ethernet interfaces.

Figure 1: Nauta Networking Diagram Example



02 Installation Tasks

Installation Requirements

Nauta Package: Extraction from Local Package

Copy the package to the installer machine, then untar it using the following command.

```
tar -zxf nauta-1.0.0-beta.tar.gz -C <destination>
```

Nauta Structure

In the extracted archive, the following appears:

Files:

- **installer.sh:** sh script
- **ansible.cfg:** configuration file for ansible

Folders:

- **bin:** binary directory
- **docs:** documentation
- **libs:** contains various scripts that are used by the installer
- **nauta:** Nauta Enterprise applications deployer
- **platform:** Kubernetes platform deployer

Components Version

To see the list of installed components and their versions, refer to: List of Included External Software Components, page 8.

Installation Process

Before proceeding with this step, you *must* create an Inventory and Configuration file (see Inventory File Example, page 15 and Example Configuration File and Option, page 18).

Installation Procedure

To install Nauta, follow these steps:

1. Set variables with configuration file, and: set environment variables to point to the configuration and inventory file on Installer Machine.

   ENV_INVENTORY (mandatory): Inventory file location, for example:

   ```
   export ENV_INVENTORY=$(<absolute path inventory file>)
   ```

   ENV_CONFIG (mandatory): Configuration file location, for example:

   ```
   export ENV_CONFIG=$(<absolute path config file>)
   ```

2. Run the installation:

   ```
   ./installer.sh install
   ```

Installation Script Options

Invoke `./installer.sh` with one of the following options:

- **install:** Kubernetes and Nauta Installation
- **platform-install:** Kubernetes installation only
- **nauta-install:** Nauta installation only
  - **Note:** It is assumed that Kubernetes is already installed, as well as a previous version of Nauta.
- **nauta-upgrade:** Nauta installation upgrade

Installation Output

Nauta will be installed on cluster nodes in the following folders: `/opt/nauta, /usr/bin, /etc/nauta-cluster`.

Access Files

On installer machine, the following files will be created in the Installation folder. These files are access files used to connect to the cluster using kubectl client.

As an output of Kubernetes installation, a file is created in the main installation directory:

```
platform-admin.config - cluster admin config file
```

As an output of the Nauta installation a file is created in main installation directory:

```
nauta-admin.config - NAUTA admin config file
```

Upgrading Nauta

To upgrade Nauta, do the following:

```
export NAUTA_KUBECONFIG=/<PATH>/nauta-admin.config
```

Call the installer with nauta-upgrade option:

```
./installer.sh nauta-upgrade
```

Note: It is recommended that you *do not use* the cluster during an upgrade.

03

User and Account Management

User and Account Management

Create a User Account

Creating a new user account creates a user account configuration file with *kubectl configuration* files.

Setting up an Administrator Environment

Before creating user accounts, you need to complete the following steps:

1. Install nctl using the description in the *Nauta User Guide*.
2. Copy the `nauta-admin.config` file to the folder where you have nctl installed. Place this configuration file in the `<location>` on the machine where nctl is running.
   - An Administrator will need the `nauta-admin.config` file on their machine (the machine where nctl resides) and it needs to be set in the `kube.config` file.
3. Set up the `KUBECONFIG` variable to point to the full path of `nauta-admin.config`, to where you copied the file in step 2. Follow the instructions below (*Creating a User*) to create users.
   - The `KUBECONFIG` variable must reside on the same machine where nctl is installed.

```
$ export KUBECONFIG=<PATH>/nauta-admin.config
```

Note: A default Admin account is created during the install process. Another Admin account cannot be created. Therefore, if additional Admins are needed a new install is required.

Creating a User Account

The following is used to create a Nauta user, *not an* Administrator. Only the person that performed the original install can complete these steps. SSH access will be required by the installer. Administrators do not have access to complete these steps.

Administrator Tasks

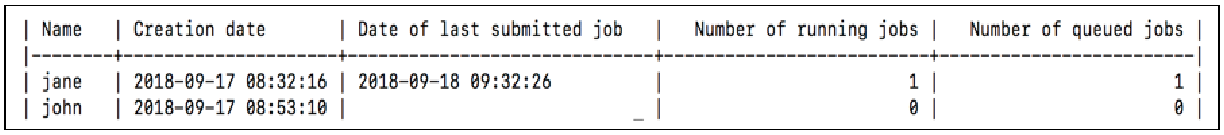
1. The Nauta `user create` command sets up a namespace and associated roles for the named user on the cluster. It sets up *home* directories, named after the username, on the *input* and *output* network shares with file-system level access privileges. To create a user:

```
$ nctl user create <username>
```

2. This command also creates a configuration file named: `<username>.config` and places this file in the user's home directory. To verify that your new user has been created:

```
$ nctl user list
```

3. This lists all users, including the new user just added, but *does not* show Administrators. An example is shown below.



```
| Name   | Creation date       | Date of last submitted job  |  Number of running jobs |  Number of queued jobs |
|--------+---------------------+-----------------------------+-------------------------+------------------------|
| jane   | 2018-09-17 08:32:16 | 2018-09-18 09:32:26         |                       1 |                      1 |
| john   | 2018-09-17 08:53:10 |                           _ |                       0 |                      0 |
```

User Tasks

1. As Administrator, you need to provide `<username>.config` file to the Nauta user. The user must save this file to an appropriate location of their choosing on the machine that is running Nauta; for example, their home directory using the following command:

   ```
   $ cp <username>.config ~/
   ```

2. Use the export command to set this variable for the user:

   ```
   $ export KUBECONFIG=~/<username>.config
   ```

Limitations

Users with the same name *cannot* be created directly after being removed. In addition, user names are limited to a 32 character maximum and there are no special characters except for hyphens. However, all names *must* start with a letter *not* a number. You can use a hyphen to join user names, for example: john-doe (see Troubleshooting for more details).

Delete a User Account

Only an Administrator can delete user accounts and deleting a user removes that user's account from the Nauta software; therefore, that user *will not* be able to log in to the system. This will halt and remove all experiments and pods; however, all artifacts related to that user's account, such as the users input and output folders and all data related to past experiments they have submitted remains.

Remove a User

To remove a user:

```
$ nctl user delete <user_name>
```

This command asks for confirmation.

Do you want to continue? [y/N]: press y to confirm deletion.

The command may take up to 30 seconds to delete the user and you may receive the message: *User is still being deleted*. Check the status of the user in a while. Recheck, as desired.

Purging

To permanently remove (purge) all artifacts associated with the user and all data related to past experiments submitted by that user:

```
$ nctl user delete <user_name> -p/--purge
```

Note: No data from the Nauta storage is deleted during this operation.

Limitations

The `nauta user delete` command may take up to 30 seconds to delete the user. A new user with the same user name *cannot* be created until after the delete command confirms that the first user with the same name has been deleted (see Troubleshooting for more details).

Launching the Web UI

To review the Resources Dashboard, launch the Web UI from the Command Line Interface (CLI).

Do the following to launch the Web UI:

1. Use the following command:

```
$ nctl launch webui
```

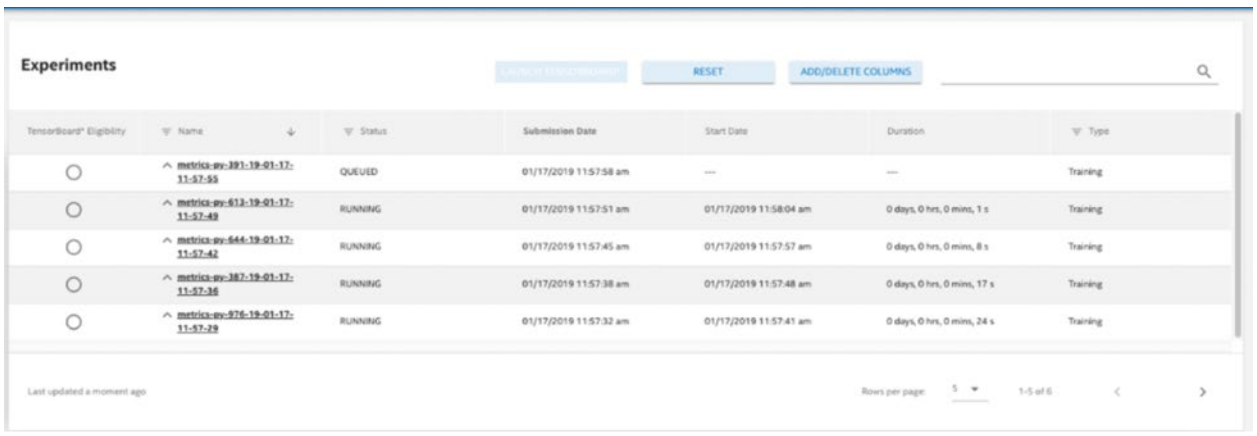
2. Your default web browser opens and displays the Web UI. For Administrators, the Web UI displays empty a list experiments and shows the following message:

```
No data for currently signed user. Click here to load all users data.
```

   The data is loaded and displays.

Figure 2 shows the Nauta Web UI. This UI contains experiment information that can be sorted by name, status, and various dates.

Figure 2: Nauta Web UI



Experiments

TensorBoard* Eligibility	Name	Status	Submission Date	Start Date	Duration	Type
○	metrics-py-391-19-01-17-11-57-55	QUEUED	01/17/2019 11:57:58 am	---	---	Training
○	metrics-py-613-19-01-17-11-57-49	RUNNING	01/17/2019 11:57:51 am	01/17/2019 11:58:04 am	0 days, 0 hrs, 0 mins, 1 s	Training
○	metrics-py-644-19-01-17-11-57-42	RUNNING	01/17/2019 11:57:45 am	01/17/2019 11:57:57 am	0 days, 0 hrs, 0 mins, 8 s	Training
○	metrics-py-387-19-01-17-11-57-36	RUNNING	01/17/2019 11:57:38 am	01/17/2019 11:57:48 am	0 days, 0 hrs, 0 mins, 17 s	Training
○	metrics-py-976-19-01-17-11-57-29	RUNNING	01/17/2019 11:57:32 am	01/17/2019 11:57:41 am	0 days, 0 hrs, 0 mins, 24 s	Training

Last updated a moment ago

Rows per page: 5 1-5 of 6

Troubleshooting

Jupyter Error 1

Saving a file causes the following error:

```
Creating file failed. An error occurred while creating a new file.
Unexpected error while saving file: input/home/filename [Errno 2]
No such file or directory: '/mnt/input/home/filename'
```

The error appears when a user tries to save file in `/input/home` folder which is a read only folder. In Jupyter, select the `/output/home` folder to correct this issue.

Jupyter Error 2

Closing the Jupyter notebook window in a Web browser causes experiments to stop executing. Attaching to the same Jupyter session still shows the stopped experiment.

Note: This is a known issue in Jupyter (refer to: https://github.com/jupyter/notebook/issues/1647 for more information). Currently, there *is no* workaround.

User Management Error (Users with the Same Name Cannot be Created)

After deleting a user name and verifying that the user name *is not* on the list of user names, it *is not* possible to create a new user with the same name within short period of time. This is due to a user's-related Kubernetes objects, which are deleted asynchronously by Kubernetes and due to these deletions can take time.

Note: To resolve, wait a few minutes before creating a user with the same name.

Docker Error

The Docker* installation on the client machine takes up significant space and contains a large amount of container images. In the Docker documentation it states: https://docs.docker.com

Docker takes a conservative approach to cleaning up unused objects (often referred to as "garbage collection"), such as images, containers, volumes, and networks: these objects are generally not removed unless you explicitly ask Docker to do so.

Note: Refer to the following information for detailed instructions on how to prune unused Docker images: https://docs.docker.com/config/pruning

Nauta Connection Error

Launching TensorBoard* instance and launching Web UI *does not* work.

After running Nauta to launch the Web UI (`nctl launch webui`) or the `nauta launch tb <experiment_name>` commands, a connection error message may be visible. During the usage of these commands, a proxy tunnel to the cluster is created.

As a result, a connection error can be caused by an incorrect `user-config` generated by Administrator or by incorrect proxy settings in a local machine. To prevent this, ensure that a valid `user-config` is used and check the proxy settings. In addition, ensure that the current proxy settings *do not* block any connection to a local machine.

Insufficient Resources Causes Experiment Failures

An experiment fails just after submission, even if the script itself is correct.

If a Kubernetes cluster *does not* have enough resources, the pods used in experiments are evicted. This results in failure of the whole experiment, even if there are no other reasons for this failure, such as those caused by users (like lack of access to data, errors in scripts and so on).

It is recommended, therefore that the Administrator investigate the failures to determine a course of action. For example, why have all the resources been consumed; then try to free them.

Removal of Docker Images

Due to known errors in Docker Garbage Collector making automatic removal of Docker images is laborious and error-prone.

Before running the Docker Garbage Collector, the Administrator should remove images that are no longer needed, using the following procedure:

1. Expose the internal Docker registry's API by exposing locally port 5000, exposed by *`nauta-docker-registry service`* located in the *nauta* namespace. This can be done, for example by issuing the following command on a machine that has access to Nauta: `kubectl port-forward svc/nauta-docker-registry 5000 -n nauta`
2. Get a list of images stored in the internal registry by issuing the following command (it is assumed that port 5000 is exposed locally): curl http://localhost:5000/v2/_catalog
   - For more information on Docker Images, refer to: https://docs.docker.com/engine/reference/commandline/image_ls/
3. From the list of images received in the previous step (step 2), choose those that should be removed. For each chosen image, execute the following steps:
   a. Get the list of tags belonging to the chosen image by issuing the following command: curl http://localhost:5000/v2/<image_name>/tags/list
   b. For each tag, get a digest related to this tag: curl --header "Accept: application/vnd.docker.distribution.manifest.v2+json" http://localhost:5000/v2/<image_name>/manifests/

      Digest is returned in a header of a response under the Docker-Content-Digest key
   c. Remove the image from the registry by issuing the following command: curl -X "DELETE" --header "Accept: application/vnd.docker.distribution.manifest.v2+json" http://localhost:5000/v2/<image_name>/manifests/
4. Run Docker Garbage Collector by issuing the following command: kube`ctl exec -it $(kubectl get --no-headers=true pods -l app=docker-registry -n nauta -o custom-`

```
columns=:metadata.name) -n nauta registry garbage-collect
/etc/docker/registry/config.yml
```

5. Restart system's Docker registry. This can be done by deleting the pod with label: `nauta_app_name=docker-registry`

Running Experiments are Evicted After Running Memory-consuming Horovod's Training

When running memory-consuming multi-node, Horovod-based experiments evicts other experiments that work on the same nodes where Horovod-related tasks were scheduled. This is caused by Horovod-related tasks that consume all memory on one master node and worker nodes. This results in eviction of other tasks running on these nodes. However, this is the standard behavior of Kubernetes.

To prevent this, schedule Horovod tasks to be only one on their nodes. To do this, set the requested number of CPUs when submitting an experiment near to maximum allocatable number of CPUs for cluster's nodes.

If the cluster has nodes with different numbers, it should be set to the greatest available number of CPUs on nodes. In this case, review the number of nodes involved in this experiment.

- Number of allocatable CPUs can be gathered by executing the `kubectl` describe node: `<node_name>` command.
- Number of allocatable CPUs is displayed in the `Allocatable->cpu` section.
- Number of CPUs gathered this way should be passed as a: `-p resources.requests.cpu <cpu_number>` parameter while issuing experiment submit.

For more information, see:
https://gitlab.cncf.ci/kubernetes/kubernetes/blob/9e25d9f6141fd0ff9bb38eaecdfbc523c9e41553/docs/design/resource-qos.md

Command Connection Error

After running `nauta launch webui` or `nctl launch tb <experiment_name>` command connection error message may display. During this command, a proxy tunnel to the cluster is created. Connection errors can be caused by incorrect user config generated by an Administrator or incorrect proxy settings for local machine.

To prevent this, ensure that a valid user config is used and check proxy settings. In addition, ensure that current proxy settings *do not* block any connection to a local machine.

Inspecting Draft Logs

While viewing the logs and output from some commands, a user may see the following message:

```
Inspect the logs with: `draft logs 01CVMD5D3B42E72CB5YQX1ANS5`
```

However, when running the command, the result is that the command *is not* found (or the logs *are not* found if there is a separate draft installation). This is because draft is located in NAUTA configuration directory: `'config'` and needs a `--home` parameter. To view the draft logs, use the following command:

```
$ ~/config/draft --home ~/config/.draft logs 01CVMD5D3B42E72CB5YQX1ANS5
```

Experiment's Pods Stuck in Terminating State on Node Failure

There may be cases where a node suddenly becomes unavailable, for example due to a power failure. Experiments using TFJob templates, which were running on such a node, will stay *Running in Nauta* and pods will terminate indefinitely. Furthermore, an experiment *is not* rescheduled automatically.

An example of such occurrence is visible using the kubectl tool:

```
                           user@ubuntu:~$ kubectl get nodes
NAME                                STATUS     ROLES    AGE   VERSION
worker0.node.infra.nauta             NotReady   Worker   7d    v1.10.6
worker1.node.infra.nauta             Ready      Worker   7d    v1.10.6
master.node.infra.nauta              Ready      Master   7d    v1.10.6

                           user@ubuntu:~$ kubectl get pods
NAME                                   READY   STATUS        RESTARTS   AGE
experiment-master-0                    1/1     Terminating   0          45m
tensorboard-service-5f48964d54-tr7xf   2/2     Terminating   0          49m
tensorboard-service-5f48964d54-wsr6q   2/2     Running       0          43m
tiller-deploy-5c7f4fcb69-vz6gp         1/1     Running       0          49m
```

This is related to unresolved issues found in Kubernetes and design of TF-operator. For more information, see links below.

- https://github.com/kubeflow/tf-operator/issues/720
- https://github.com/kubernetes/kubernetes/issues/55713

To solve this issue, manually resubmit the experiment using `nauta`.

A Multinode, Horovod-based Experiment Receives a FAILED Status after a Pod's Failure

If during execution of a multinode, Horovod-based experiment one of pods fails, then the whole experiment gets the FAILED status. Such behavior is caused by a construction of Horovod framework. This framework uses an Open MPI framework to handle multiple tasks instead of using Kubernetes features. Therefore, it *cannot* rely also on other Kubernetes features such as restarting pods in case of failures.

This results in training jobs using Horovod *does not* resurrect failed pods. The Nauta system also *does not* restart these training jobs. If a user encounters this, they should rerun the experiment manually. The construction of a multinode TFJob-based experiment is different, as it uses Kubernetes features. Thus, training jobs based on TFJobs restart failing pods so an experiment can be continued after their failure without a need to be restarted manually by a user.

Nodes Hang When Running Heavy Training Jobs

Running heavy training jobs on workers with the operating system kernel older than 4.* might lead to hanging the worker node. See https://bugzilla.redhat.com/show_bug.cgi?id=1507149 for more information.

This may occur when a memory limit for 0 job is set to a value close to the maximum amount of memory installed on this node. These problems are caused by errors in handling memory limits in older versions of the kernel. To avoid this problem, it is recommended to install on all nodes of a cluster with a newer version of a system's kernel.

The following kernel was verified as a viable fix for this issue (see link below).

https://elrepo.org/linux/kernel/el7/x86_64/RPMS/kernel-ml-4.19.2-1.el7.elrepo.x86_64.rpm

To install the new kernel, refer to: *Chapter 5, Manually Upgrading the Kernel* in RedHat's Kernel Administration Guide (see link below).

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/kernel_administration_guide/ch-manually_upgrading_the_kernel

Note: The above kernel *does not* include RedHat's optimizations and hardware drivers.

Platform Client for Microsoft Windows

Using *standard* Microsoft* Windows* terminals (`cmd.exe`, `power shell`) is enough to interact with platform, but there is a sporadic issue with the output. Some lines can be surrounded with incomprehensible characters, for example:

```
[K[?25hCannot connect to K8S cluster: Unable to connect to the server: d...
```

Note: The recommended shell environment for Windows operating system is bash. For bash-based terminals, this issue *does not* occur.