Гайд для новичков по установке Kubernetes

© кадр из к/ф «Пираты Карибского моря»

С чего начинается практическое освоение любой системы? Правильно, с установки. Данный гайд является компиляцией из народной мудрости, официальной документации, а также собственного опыта и призван помочь новичкам разобраться с тем, как же все таки устанавливать Kubernetes.

Мы потренируемся ставить как вырожденный кластер «все-в-одном», состоящий только из одного узла, так и настоящий высокодоступный (high available) кластер с полным резервированием. В процессе работы мы рассмотрим применение различных контейнерных движков (Container Runtimes): cri-o, containerd, связки Docker + cri-dockerd plugin. Кроме этого, потренируемся настраивать отказоустойчивый балансировщик нагрузки на базе keepalived и haproxy.

Весь процесс установки будет детальным образом прокомментирован и разложен по шагам, а в реперных точках мы будем делать снимки состояния виртуальных машин (snapshots), что позволит рассмотреть различные варианты установки без необходимости делать одну и ту же работу по несколько раз.

Оглавление

1. Зачем все это надо?

Одним из золотых правил построения надежной и безопасной информационной инфраструктуры является максимальная изоляция ее компонентов друг от друга. Хорошим признаком достижения этой цели можно считать кейс, когда один сервер выполняет только одну основную функцию. Например, контроллер Active Directory – это только контроллер Active Directory, а не файловый или Интернет-прокси сервер в придачу.

Изначально инфраструктура разделялась с помощью выделенных аппаратных серверов, но это было очень дорогое удовольствие. Потом появилась технология виртуализации. Затраты на изоляцию существенно снизились, но все же оставались довольно высоки: виртуальные машины потребляли значительно количество вычислительных ресурсов, медленно запускались, им требовались отдельные лицензии на системное и прикладное ПО.

Следующим этапом развития стала контейнеризация. Если виртуальная машина — это почти отдельный компьютер со своим BIOS, операционной системой, драйверами и так далее, то контейнер — это изолированная часть операционной системы узла с минимумом прикладного ПО, обеспечивающего его запуск. В результате этого контейнеры занимают мало места, быстро стартуют и существенно минимизируют другие сопутствующие расходы.

Контейнеры идеологически очень похожи на портативный (portable) софт. Они содержат в себе лишь те файлы, что нужны для запуска конкретной программы. Однако, в отличии от обычных портативных программ, технологии контейнеризации отделяют контейнеры друг от друга и от хозяйской (host) операционной системы, позволяя каждому контейнеру считать себя отдельным компьютером, что очень похоже на работу виртуальных машин.

С точки зрения безопасности изоляция контейнеров хуже, нежели изоляция виртуальных машин. Тем не менее, достигаемого уровня безопасности достаточно для большинства сценариев применения.

Контейнеризация получила широкую известность вместе с Docker. Эта система сделала работу с контейнерами чрезвычайно простой и доступной. Она хорошо подходит для управления контейнерами в небольших проектах, но для серьезных задач, когда нужно оперировать большим числом контейнеров, организовывать отказоустойчивые конфигурации, гибко управлять вычислительными ресурсами, ее возможностей недостаточно, и здесь на сцену выходит герой нашей статьи – система управления/оркестрации (orchestration) контейнерами Kubernetes.

2. Как устроен Kubernetes

По факту Kubernetes — очень гибкое решение, которое может быть настроено бесчисленным количеством способов. Это, конечно, очень здорово для работы, но является сущим адом при изучении. Поэтому здесь мы не будем рассматривать все возможные варианты построения системы, а ограничимся лишь базовым, достаточным для её первичного освоения.

Kubernetes кластер состоит из двух типов узлов: управляющих и рабочих.

Управляющие узлы, как видно из названия, предназначены для управления кластером. Они отдают команды рабочим узлам на запуск и остановку рабочих нагрузок (workloads), отслеживают состояние кластера, перераспределяют задачи в случае отказов и совершают множество других управленческих действий. Рабочие узлы — это пчелки, выполняющие всю полезную работу, ради которой функционирует кластер.

На самом деле управляющие узлы тоже могут выполнять рабочие нагрузки, правда с точки зрения безопасности это считается нежелательным. Поскольку, если в одной из рабочих нагрузок будет вредоносный код, то, будучи запущенным на управляющем узле, он сможет натворить гораздо больше бед, нежели будучи запущенным на рабочем узле.

Типовой состав ПО рабочего узла включает в себя (*Рисунок 1*):

1. Служебные компоненты Kubernetes: агент управления узлом kubelet, узловой прокси kube-proxy
2. Сетевой плагин (Container Network Interface, CNI plugin).
3. Контейнерный движок: cri-o, containerd или Docker + cri-dockerd plugin.
4. Рабочие нагрузки (workloads), то есть сами контейнеры, из-за которых все и затевалось. Однако, здесь важно уточнить один существенный момент — минимальной единицей управления рабочей нагрузкой в Kuberbetes является "*под*" (pod), состоящий из одного (как правило) или нескольких контейнеров.

Рисунок 1

Состав ПО управляющих узлов (*Рисунок 2*) дополнительно включает в себя:

1. Управляющие компоненты Kubernetes: планировщик kube-scheduler, базовый демон управления kube-controller-manager, REST API сервер управления kube-apiserver.
2. Отказоустойчивое хранилище etcd.
3. Балансировщик нагрузки (для случаев использования нескольких управляющих узлов в кластере).

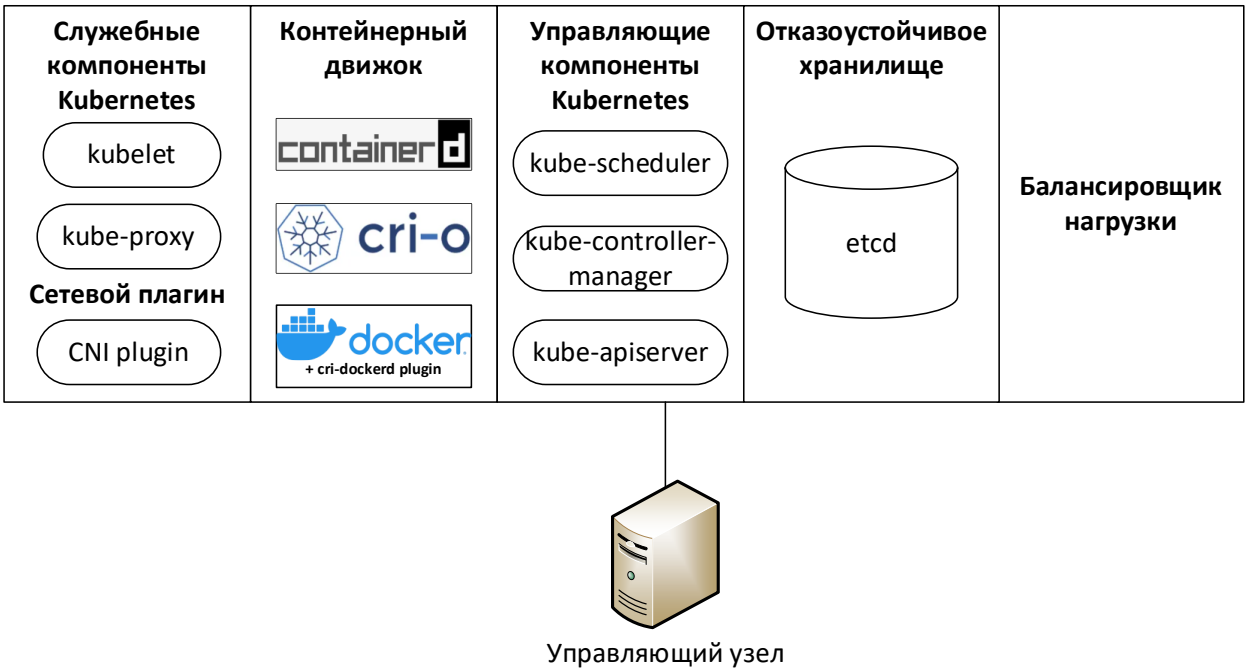

Рисунок 2

Важно отметить, что практически все компоненты, кроме kubelet и kube-proxy, могут функционировать в Kubernetes в качестве рабочих нагрузок. Другими словами, Kubernetes может управлять сам собой.

Читая о Kubernetes, часто можно услышать фразу: «Kubernetes – это просто: всего 5 бинарников». Под пятью бинарниками обычно понимают: kubelet, kubeproxy, kube-scheduler, kube-controller-manager, kube-apiserver. При этом почему-то всегда умалчивается о других обязательных компонентах кластера, хотя бы о том же etcd, так что Kubernetes — это далеко не просто и далеко не пять бинарников.

3. Способы установки Kubernetes

Kubernetes в учебных целях может быть реализован с помощью различных утилит и готовых дистрибутивов:

- kind (Kubernetes in Docker). Кластер, функционирующий на локальном компьютере «внутри» Docker.
- minikube. Кластер в одной утилите для запуска на локальном компьютере.
- Docker desktop – дистрибутив Docker для запуска на локальном компьютере с возможностью включить Kubernetes одной галкой – наверное, самый дружественный вариант для людей, которые не представляют, что такое Kubernetes, и хотят просто на него глянуть.

Для промышленного использования Kubernetes должен быть развернут «по-честному». Для этого существуют следующие варианты:

- Развертывание с помощью kubespay – набора скриптов (playbook’s) для системы управления инфраструктурой Ansible.

- Полностью ручное развёртывание «hard way». Гайд на английском можно почитать тут, на русском тут.
- Развертывание с помощью утилиты *kubeadm*. Это то, чем мы будем заниматься далее.

4. Схема виртуального стенда

Согласно официальной документации, к машинам, на которых разворачивается Kubernetes, выдвигаются следующие требования:

- 2+ GB ОЗУ;
- 2+ процессорных ядра;
- Linux хост с отключенным файлом подкачки (swap);

При разворачивании кластера на нескольких узлах, согласно тем же документам, накладываются дополнительные требования:

- полная сетевая связанность узлов;
- на каждом узле должны быть уникальные:
    - имена узлов (проверка с помощью команды "*hostname*"),
    - MAC-адреса (проверка с помощью команды "*ip link*"),
    - параметр *product_uuid*, являющийся уникальным идентификатором виртуальной машины (проверка с помощью команды "*cat /sys/class/dmi/id/product_uui*").

В ходе экспериментов выяснились, что дополнительно к этому узлы должны иметь статические IP-адреса и зарегистрированные DNS имена, что требуется для автоматического выпуска сертификатов во время работы *kubeadm*.

Минимальное количество рабочих узлов для схемы с резервированием – 2. Логичное требование: один сломался другой на замену. Минимальное количество управляющих узлов для схемы с резервирование – 3. Данное странное требование продиктовано официальной документацией: в Kubernetes должно быть нечетное количество управляющих улов. Минимальное число нечетное число для обеспечения избыточности — 3.

Таким образом, наш виртуальный стенд (*Рисунок 3*) будет состоять из пяти виртуальных машин, находящихся в одноранговой сети, имеющей выход в Интернет, с помощью виртуального маршрутизатора, реализующего NAT.

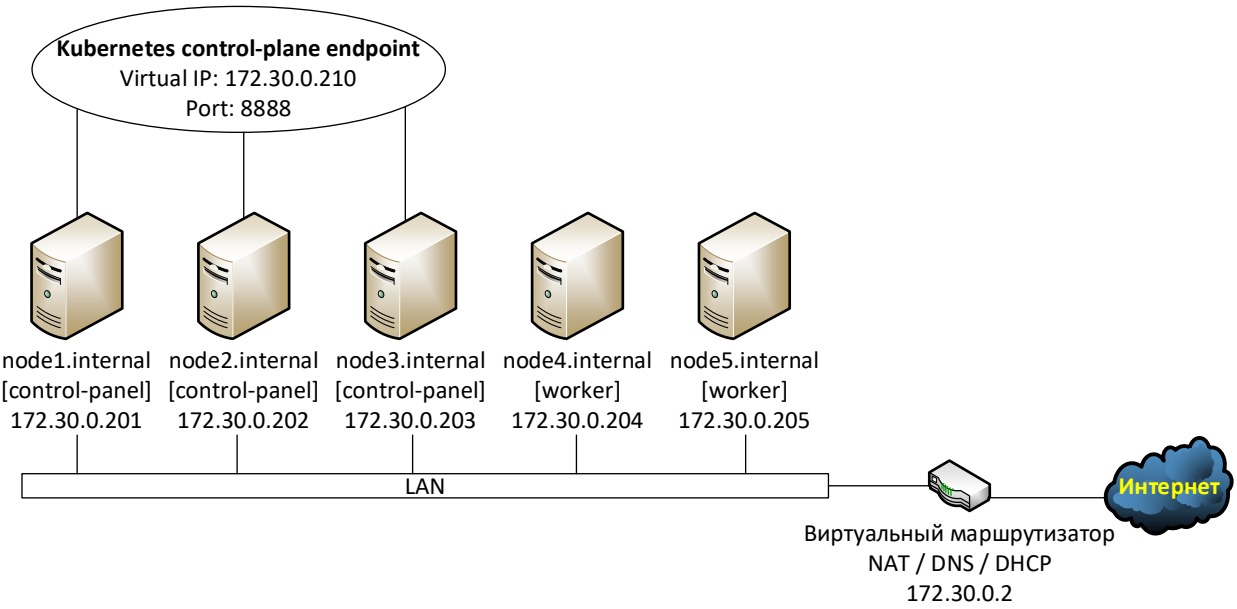

Рисунок 3

Виртуальные машины будут работать под управлением ОС Debian 11 x64, установленной с минимальным количеством пакетов. Все необходимое будем явно доставлять.

5. Постановка задач

Задача 1. Организовать на базе узла **node1** вырожденный Kubernetes кластер («все-в-одном»).
Задача 2. Организовать высокодоступный кластер Kubernetes, в котором узлы **node1**, **node2**, **node3** будут управляющими (control-panel), а **node3** и **node4** – рабочими (workers).

6. Решение

Договоримся, что все действия в данном гайде будем выполнять от пользователя *root*. Сначала обе задачи будем решать параллельно, а затем в нужных местах сделаем развилки. Для минимизации переделки, в случае выявления ошибок, будем периодически проверять настройки и делать снимки состояния виртуальных машин.

В начальной точке у нас должно быть 5 свежеустановленных виртуальных машин, работающих под ОС Debian 11 x64. Машины должны быть в виртуальной сети с настройками 172.30.0.0/24. Шлюз по умолчанию – 172.30.0.2, он же DHCP, DNS и NAT сервер. Все машины должны иметь доступ в Интернет. Если все так, то делаем снимок состояния виртуальных машин и назовем его «START».

6.1. Предварительная настройка узлов кластера

6.1.1. Настройка статических IP адресов узлов кластера

На узле node1 содержимое файла */etc/network/interfaces* заменим следующим:

```
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

source /etc/network/interfaces.d/*

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto ens33
iface ens33 inet static
address 172.30.0.201
netmask 255.255.255.0
gateway 172.30.0.2
```

На узле node2 содержимое файла */etc/network/interfaces* заменим следующим:

```
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

source /etc/network/interfaces.d/*

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto ens33
iface ens33 inet static
address 172.30.0.202
netmask 255.255.255.0
gateway 172.30.0.2
```

На узле node3 содержимое файла */etc/network/interfaces* заменим следующим:

```
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

source /etc/network/interfaces.d/*

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto ens33
iface ens33 inet static
address 172.30.0.203
netmask 255.255.255.0
gateway 172.30.0.2
```

На узле node4 содержимое файла */etc/network/interfaces* заменим следующим:

```
# This file describes the network interfaces available on your system
```

```
# and how to activate them. For more information, see interfaces(5).

source /etc/network/interfaces.d/*

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto ens33
iface ens33 inet static
address 172.30.0.204
netmask 255.255.255.0
gateway 172.30.0.2
```

На узле node5 содержимое файла */etc/network/interfaces* заменим следующим:

```
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

source /etc/network/interfaces.d/*

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto ens33
iface ens33 inet static
address 172.30.0.205
netmask 255.255.255.0
gateway 172.30.0.2
```

6.1.2. Настройка имен узлов кластера

На узле node1 выполним команду:

```
hostnamectl set-hostname node1.internal
```

На узле node2 выполним команду:

```
hostnamectl set-hostname node2.internal
```

На узле node3 выполним команду:

```
hostnamectl set-hostname node3.internal
```

На узле node4 выполним команду:

```
hostnamectl set-hostname node4.internal
```

На **узле node5** выполним команду:

```
hostnamectl set-hostname node5.internal
```

6.1.3. Настройка DNS

На всех узлах содержимое файла */etc/resolv.conf* заменим следующим:

```
nameserver 172.30.0.2
```

6.1.4. Настройка файла hosts

Поскольку мы не используем DNS-сервер, то для разрешения важных для нас DNS-имен настроим файлы hosts на всех узлах кластера.

На всех узлах выполним следующую команду:

```
cat > /etc/hosts <<EOF
127.0.0.1       localhost

# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

# Cluster nodes
172.30.0.201 node1.internal
172.30.0.202 node2.internal
172.30.0.203 node3.internal
172.30.0.204 node4.internal
172.30.0.205 node5.internal
EOF
```

6.1.5. Проверка сетевых настроек

Убедимся, что все сетевые настройки сделаны правильно.
Для этого на каждом узле по очереди выполним следующие тестовые команды:

```
# Проверка доступности шлюза по умолчанию
ping 172.30.0.2 -c 1

# Проверка доступности node1
ping 172.30.0.201 -c 1
ping node1.internal -c 1

# Проверка доступности node2
ping 172.30.0.202 -c 1
ping node2.internal -c 1

# Проверка доступности node3
ping 172.30.0.203 -c 1
ping node3.internal -c 1

# Проверка доступности node4
ping 172.30.0.204 -c 1
ping node4.internal -c 1

# Проверка доступности node5
ping 172.30.0.205 -c 1
ping node5.internal -c 1

# Проверка «видимости» Интернета
ping 8.8.8.8 -c 1
```

Все тесты должны проходить без ошибок. Если все так, то делаем снимок состояния виртуальных машин и называем его «NETWORK».

6.1.6. Установка вспомогательных пакетов

Вариант A. Самый простой и быстрый вариант — это установить все и везде, не зависимо от того, нужно оно там или нет.

На всех узлах выполним команду:

```
apt install -y curl wget gnupg sudo iptables tmux keepalived haproxy
```

Вариант B. Более трудоемкий вариант — это на каждом узле поставить только то, что нужно.

На всех узлах выполним команду:

```
apt install -y curl wget gnupg sudo iptables
```

На узле node1 выполним команду:

```
apt install -y tmux
```

На узлах node1, node2, node3 выполним команду:

```
apt install -y keepalived haproxy
```

6.1.7. Предварительная подготовка Linux для использования Kubernetes

Согласно официальной документации, для работы Kubernetes необходимо разрешить маршрутизацию IPv4 трафика, настроить возможность iptables видеть трафик, передаваемый в режиме моста, а также отключить файлы подкачки.

На всех узлах выполним команды:

```
# Настройка автозагрузки и запуск модуля ядра br_netfilter и overlay
cat <<EOF | tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF

modprobe overlay
modprobe br_netfilter

# Разрешение маршрутизации IP-трафика
echo -e "net.bridge.bridge-nf-call-ip6tables = 1\nnet.bridge.bridge-nf-call-
iptables = 1\nnet.ipv4.ip_forward = 1" > /etc/sysctl.d/10-k8s.conf
sysctl -f /etc/sysctl.d/10-k8s.conf

# Отключение файла подкачки
swapoff -a
sed -i '/ swap / s/^/#/' /etc/fstab
```

Проверка корректности настройки
Чтобы убедиться, что все требуемые параметры настроены правильно, рекомендуется перезагрузить виртуальную машину.

Для проверки автоматической загрузки модулей *br_netfilter* и *overlay* выполним команды:

```
lsmod | grep br_netfilter
lsmod | grep overlay

## Ожидаемый результат должен быть следующим (цифры могут отличаться):
# br_netfilter           32768  0
# bridge                258048  1 br_netfilter
# overlay               147456  0
```

Для проверки успешности изменения настроек в параметрах сетевого стека выполним команду:

```
sysctl net.bridge.bridge-nf-call-iptables net.bridge.bridge-nf-call-ip6tables
net.ipv4.ip_forward

## Ожидаемый результат:
# net.bridge.bridge-nf-call-iptables = 1
# net.bridge.bridge-nf-call-ip6tables = 1
# net.ipv4.ip_forward = 1
```

Для проверки отключения файла подкачки выполним команду:

```
swapon -s

## Ожидаемый вывод команды – пустой. Она ничего не должна отобразить.
```

6.1.8. [ОПЦИОНАЛЬНО] Разрешение авторизации в SSH от пользователя root

Внимание! Данный шаг снижает безопасность узлов. Выполнять его можно только в учебной среде, когда весь виртуальный стенд работает на вашем локальном компьютере, и виртуальные машины не доступны из сети.

При выполнении этой работы нам придется часто подключаться по SSH к машинам нашего стенда. По умолчанию настройки безопасности Debian запрещают нам напрямую подключаться под *root*, и приходится выполнять двойную авторизацию: сначала заходить под пользователем, а потом переключаться на root. Упростим себе жизнь и разрешим SSH сразу пускать нас под *root*/

На всех узлах выполним следующие команды:

```
echo "PermitRootLogin yes" > /etc/ssh/sshd_config.d/01-permitroot.conf
service sshd restart
```

Восстановление запрета авторизации под root выглядит следующим образом.

На всех узлах нужно будет выполнить команды:

```
rm /etc/ssh/sshd_config.d/01-permitroot.conf
service sshd restart
```

Лайфхак. Использование TMUX для одновременного конфигурирования нескольких узлов

Еще одним способом упростить себе жизнь будет использование программы tmux для совершения одинаковых действий (например, установки программ) на нескольких узлах. Магия работает за счет того, что *tmux* позволяет одновременно открыть несколько окон, а затем включить синхронизацию, и консольные команды, введенные в одном окне, будут

автоматически транслироваться во все другие окна.

В частности, для проведения одинаковых работ на всех узлах стенда (например, как в следующем шаге) необходимо сделать следующее:

1. На узле **node1** запустить *tmux*. Напомню, что именно на этот узел мы ранее поставили *tmux*.
2. С помощью интерфейса *tmux* сделать 4 дополнительных окна.
3. В полученных окнах поочередно совершить подключение по *ssh* к оставшимся узлам: **node2**, **node3**, **node4**, **node5**.
4. Перейти в окно, соответствующее узлу **node1**.
5. Активировать режим синхронизации команд между окнами
6. Провести необходимые работы.

По окончании работ:

1. Отключить режим синхронизации команд.
2. Закрыть все созданные окна.
3. Выйти из *tmux*.

Программа *tmux* может управляться как горячими клавишами, так и текстовыми командами. Принцип управления горячими клавишами заключается в нажатии префикса Ctrl-B (одновременно Ctrl и кнопку «B»), а затем кнопки соответствующей команды. Например, <Ctrl-B %> – разделит окно по вертикали. Аналогичная ей текстовая команда «split-window -h» сделает тоже самое. Для перехода в командный режим необходимо нажать <Ctrl-B :>. Перечень всех горячих клавиш можно узнать, нажав <Ctrl-B ?>.

С теорией разобрались, рассмотрим пример использования.

Пример проверки доступности Интернет одновременно на всех узлах кластера

1. На узле **node1** запускаем *tmux* (Рисунок 4).

Рисунок 4

2. Нажимая <Ctrl-B “> 3 раза, разделяем окно на 3 горизонтальных окна (Рисунок 5).

Рисунок 5

3. Нажимая <Ctrl-B %> разделяем нижнее окно на 3 вертикальных окна (Рисунок 6).

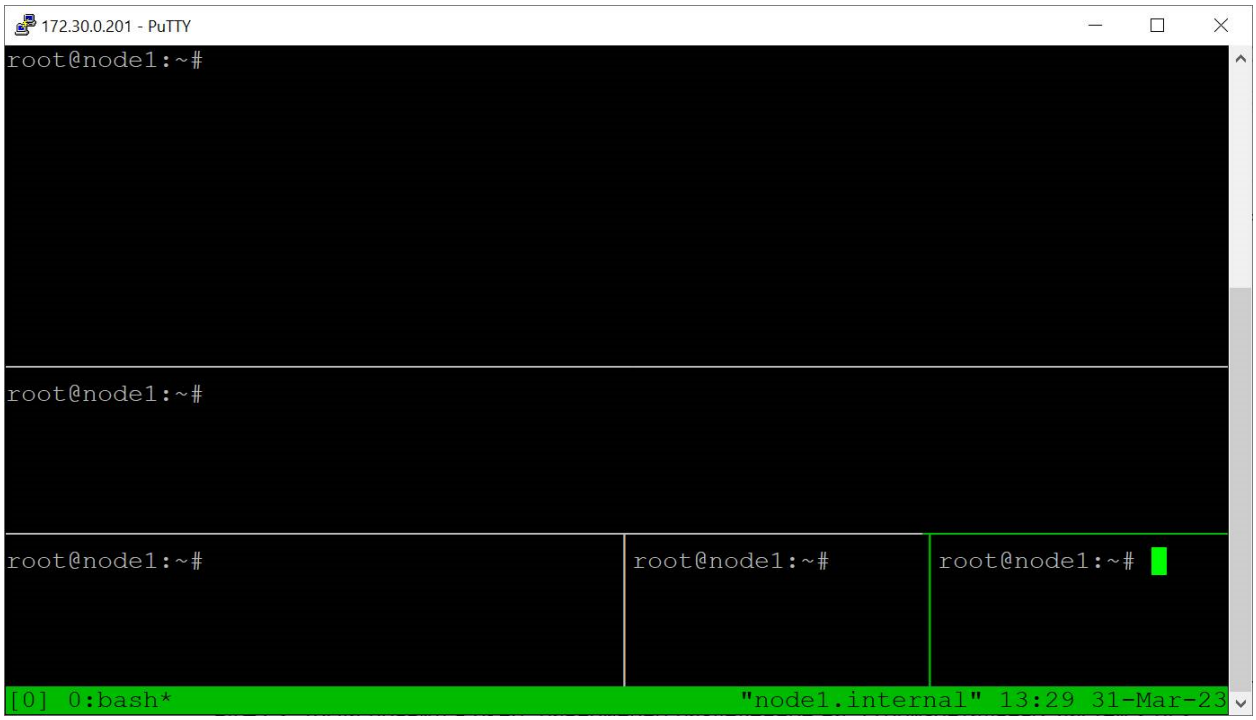

Рисунок 6

4. Используя навигацию между окнами с помощью <Ctrl-B стрелки>, а также возможности изменения размера окон <Ctrl-B Alt-стрелки>, можно добиться более красивого размера окон (Рисунок 7).

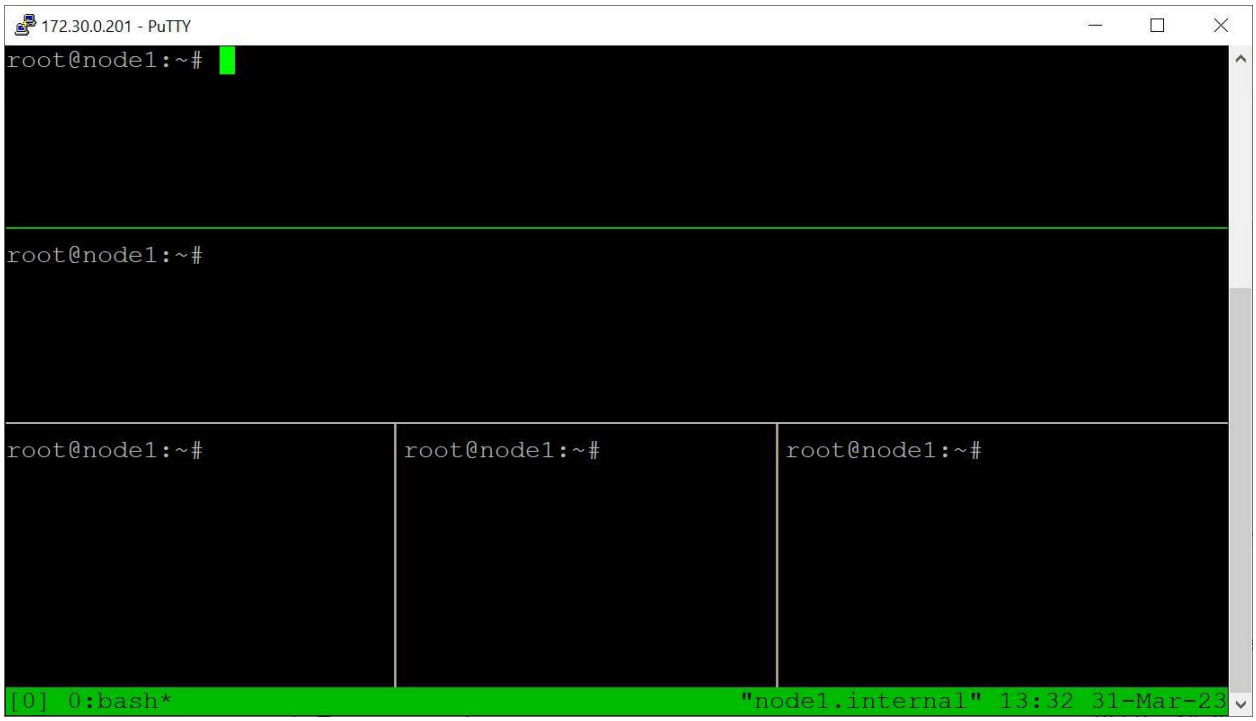

Рисунок 7

Если не хочется мудрить с размерами окон, можно просто нажать <Ctrl-B Alt-2>, тогда появится окно, равномерно разделенное на 5 горизонтальных частей (Рисунок 8).

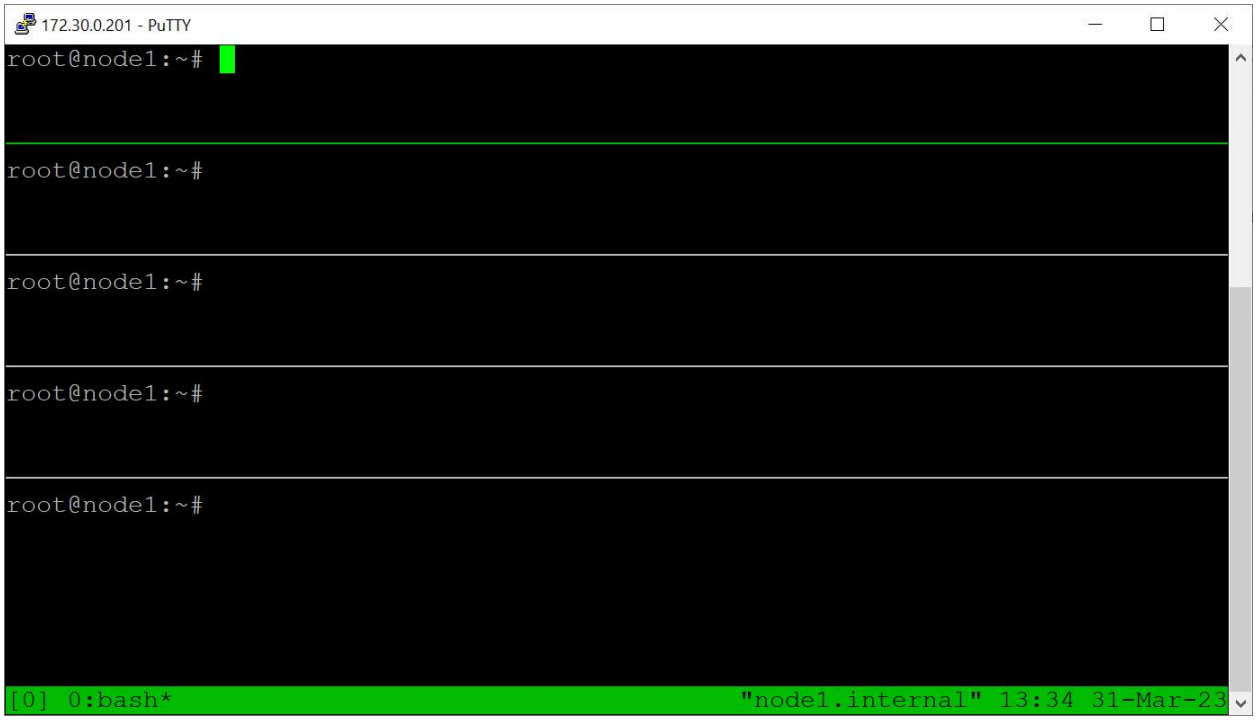

Рисунок 8

5. Поочередно в каждом окне, кроме самого верхнего, с помощью команды «*ssh root@nodeX.internal*», где X – номер узла (например, node4.internal), подключимся ко всем узлам кластера (Рисунок 9).

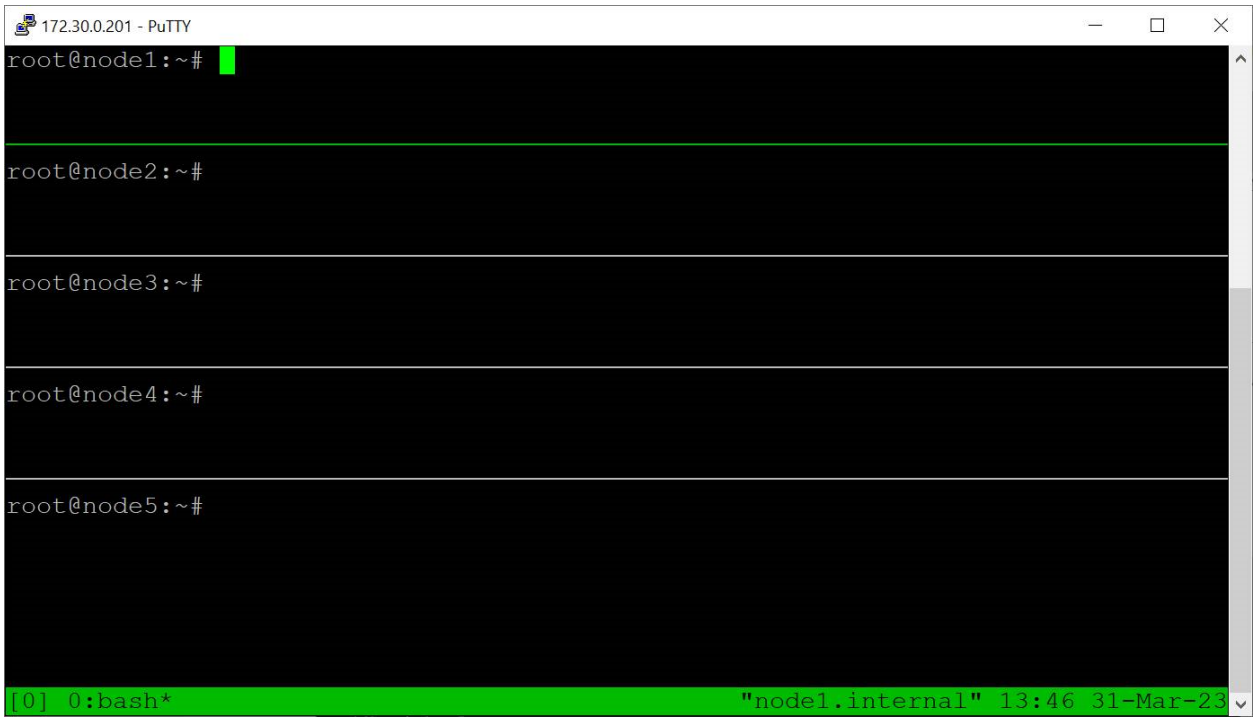

Рисунок 9

Примечание. Для отчистки экрана от лишнего вывода можно воспользоваться командой «*clear*».

6. Теперь перейдем в самое верхнее окно, в этом примере оно будет у нас командным. Затем перейдем в режим ввода команд, нажав <Ctrl-B :> и в командной строке введем команду «*set synchronize-panes on*» (Рисунок 10)

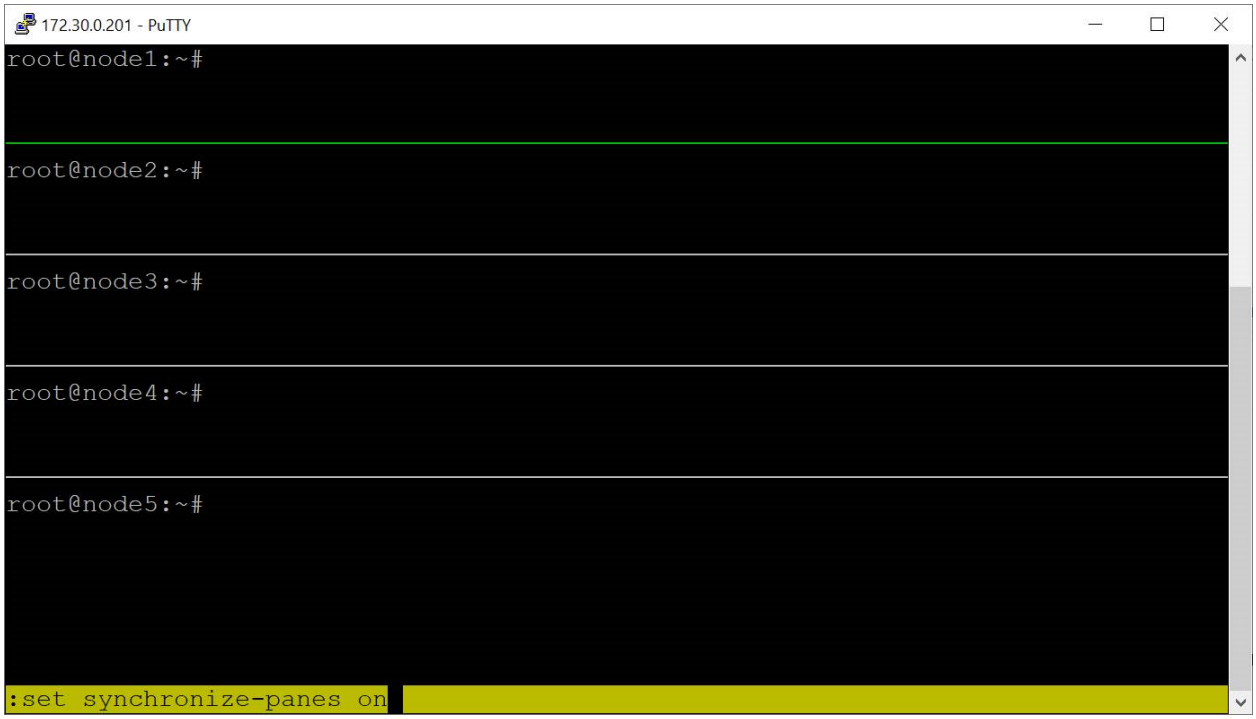

Рисунок 10

7. Теперь введем команду «*ping 8.8.8.8 -c 1*», и она одновременно выполнится на всех панелях (Рисунок 11)

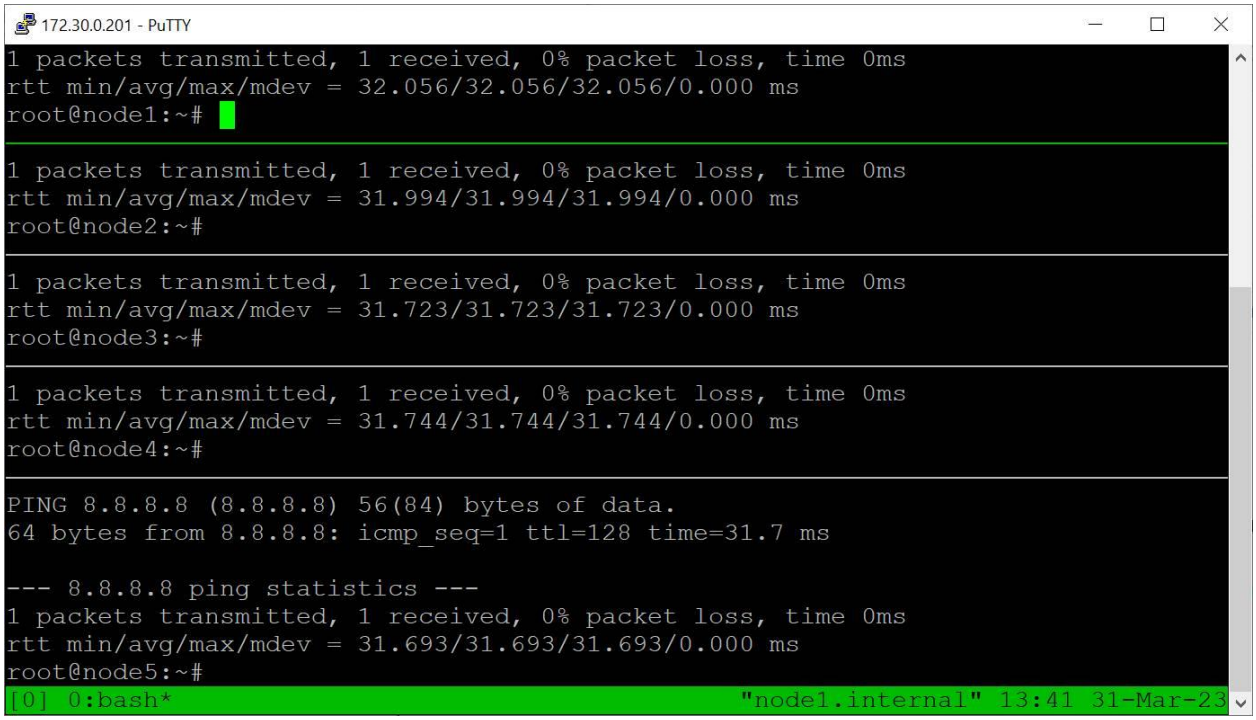

Рисунок 11

8. Для завершения режима синхронизации необходимо в командном режиме *tmux* ввести команду «*set synchronize-panes off*». Закрытие панелей производится клавишами <Ctrl-B x>. Другой вариант в режиме активной синхронизации просто набрать команду «*exit*».

ВНИМАНИЕ! Использование *tmux* для установок ПО может привести к психологической зависимости от чувства азарта, вызванного переживанием за скорость выполнения процесса в том или ином окне. Не надо расстраиваться и удивляться, когда в каждом конкретном случае окно победитель будет отличаться от ваших ожиданий. Помните, азартные игры до добра не доводят.

6.1.9. Установка kubeadm и kubectl

kubectl – основная утилита командной строки для управления кластером Kubernetes, kubeadm – утилита для развертывания кластера Kubernetes. Установка данных утилит осуществляется в соответствии с официальным руководством.

На всех узлах выполним следующие команды:

```
# Настройка deb-репозитория Kubernetes
curl -fsSLo /etc/apt/trusted.gpg.d/kubernetes-archive-keyring.gpg
https://packages.cloud.google.com/apt/doc/apt-key.gpg

echo "deb [signed-by=/etc/apt/trusted.gpg.d/kubernetes-archive-keyring.gpg]
https://apt.kubernetes.io/ kubernetes-xenial main" | tee
/etc/apt/sources.list.d/kubernetes.list

# Обновление перечня доступных пакетов
apt update

# Установка пакетов kubeadm и kubectl
apt install -y kubeadm kubectl
```

На этом первый этап завершен. Наши узлы готовы к дальнейшим экспериментам. Делаем снимок состояния виртуальных машин и называем его «HOST IS PREPARED». К этому снимку мы будем неоднократно возвращаться.

6.2. Установка контейнерного движка

Kubernetes модульная система и может работать с различными контейнерными движками. При проведении экспериментов будем устанавливать по одному движку за раз. Хотя, как вы наверное догадались, на хосте одновременно может быть несколько движков и на разных узлах используемые движки могут отличаться, но это уже совсем другая история.

По окончанию установки движка будем делать снимок состояния виртуальных машин, затем откатываться на предыдущий снимок («HOST IS PREPARED») и ставить следующий движок.

6.2.A. Вариант A. Установка cri-o

Установка по осуществляется по официальной документации.

На всех узлах выполним следующие команды:

```
export VERSION=1.26
export OS=Debian_11

echo "deb
https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable/
$OS/ /" > /etc/apt/sources.list.d/devel:kubic:libcontainers:stable.list
echo "deb
http://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable:/
cri-o:/$VERSION/$OS/ /" >
/etc/apt/sources.list.d/devel:kubic:libcontainers:stable:cri-o:$VERSION.list

curl -L
https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable:
/cri-o:/$VERSION/$OS/Release.key | apt-key add -
curl -L
https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable/
$OS/Release.key | apt-key add -

apt-get update
apt-get install -y cri-o cri-o-runc

mkdir /var/lib/crio

systemctl daemon-reload
systemctl enable --now crio
```

6.2.A.1. Проверка доступности сокета cri-o

Для тестов мы будем использовать утилиту crictl, которая автоматически установилась вместе с *kubeadm*.

На всех узлах запустим команду:

```
crictl --runtime-endpoint unix:///var/run/crio/crio.sock version

## Ожидаемый результат:
# Version:  0.1.0
# RuntimeName:  cri-o
# RuntimeVersion:  1.25.2
# RuntimeApiVersion:  v1
```

6.2.A.2. Проверка запуска контейнеров с помощью cri-o

Кроме доступности сокета мы можем проверить фактическую возможность запуска контейнеров.

На всех узлах запустим команды:

```
export CONTAINER_RUNTIME_ENDPOINT=unix:///run/crio/crio.sock

cat > pod.config << _EOF_
{
    "metadata": {
        "name": "test-pod",
        "namespace": "default",
        "attempt": 1,
```

```
        "uid": "18fbfef14ae3a43"
    },
    "log_directory": "/tmp",
    "linux": {
    }
}
_EOF_

cat > container.config << _EOF_
{
  "metadata": {
    "name": "hello-world-container"
  },
  "image":{
    "image": "hello-world"
  },
  "log_path":"hello-world.log",
  "linux": {
  }
}
_EOF_

crictl pull hello-world
#crictl run container.config pod.config

POD_ID=$(crictl runp pod.config)
CONTAINER_ID=$(crictl create $POD_ID container.config pod.config)
crictl start $CONTAINER_ID

cat /tmp/hello-world.log

## Ожидаемый ответ
# …
# 2023-03-28T20:06:48.436017504+03:00 stdout F
# 2023-03-28T20:06:48.436017504+03:00 stdout F Hello from Docker!
# …
```

Если все тесты прошли успешно, то делаем снимок состояния виртуальных машин и называем его «CRI-O».

6.2.B. Вариант B. Установка containerd

Откатываемся к состоянию виртуальных машин «HOST IS PREPARED». Установка осуществляется в соответствии с официальным руководством.

На всех узлах выполним команды:

```
# Установка containerd
wget
https://github.com/containerd/containerd/releases/download/v1.7.0/containerd-
1.7.0-linux-amd64.tar.gz
tar Cxzvf /usr/local containerd-1.7.0-linux-amd64.tar.gz
rm containerd-1.7.0-linux-amd64.tar.gz

# Создание конфигурации по умолчанию для containerd
mkdir /etc/containerd/
containerd config default > /etc/containerd/config.toml
```

```
# Настройка cgroup драйвера
sed -i 's/SystemdCgroup \= false/SystemdCgroup \= true/g'
/etc/containerd/config.toml

# Установка systemd сервиса для containerd
wget
https://raw.githubusercontent.com/containerd/containerd/main/containerd.service
mv containerd.service /etc/systemd/system/

# Установка компонента runc
wget https://github.com/opencontainers/runc/releases/download/v1.1.4/runc.amd64
install -m 755 runc.amd64 /usr/local/sbin/runc
rm runc.amd64

# Установка сетевых плагинов:
wget
https://github.com/containernetworking/plugins/releases/download/v1.2.0/cni-
plugins-linux-amd64-v1.2.0.tgz
mkdir -p /opt/cni/bin
tar Cxzvf /opt/cni/bin cni-plugins-linux-amd64-v1.2.0.tgz
rm cni-plugins-linux-amd64-v1.2.0.tgz

# Запуск сервиса containerd
systemctl daemon-reload
systemctl enable --now containerd
```

6.2.B.1. Проверка доступности сокета containerd

На всех узлах выполним команду:

```
crictl --runtime-endpoint unix:///var/run/containerd/containerd.sock version

## Ожидаемый результат:
# Version:  0.1.0
# RuntimeName:  containerd
# RuntimeVersion:  v1.7.0
# RuntimeApiVersion:  v1
```

6.2.B.2. Проверка возможности запуска контейнеров с помощью containerd

На всех узлах выполним команды:

```
ctr images pull docker.io/library/hello-world:latest
ctr run docker.io/library/hello-world:latest hello-world

## Ожидаемый результат:
# …
# Hello from Docker!
# This message shows that your installation appears to be working correctly.
# …
```

Если все тесты прошли успешно, то делаем снимок состояния виртуальных машин и называем его «CONTAINERD».

6.2.C. Вариант C. Установка Docker + cri-dockerd

Откатываемся к состоянию виртуальных машин «HOST IS PREPARED». В начале следует установить Docker. Для этого воспользуется официальным руководством.

На всех узлах выполним следующие команды:

```
curl -fsSL https://download.docker.com/linux/debian/gpg | gpg --dearmor -o
/etc/apt/trusted.gpg.d/docker.gpg

echo \
  "deb [arch=$(dpkg --print-architecture) signed-
by=/etc/apt/trusted.gpg.d/docker.gpg] https://download.docker.com/linux/debian
\
  $(lsb_release -cs) stable" | tee /etc/apt/sources.list.d/docker.list >
/dev/null

apt update

apt install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

После установки Docker установим плагин cri-dockerd. Делать это будем по размещённому в сети гайду.

На всех узлах выполним следующие команды:

```
wget https://github.com/Mirantis/cri-dockerd/releases/download/v0.3.1/cri-
dockerd-0.3.1.amd64.tgz
tar xvf cri-dockerd-0.3.1.amd64.tgz
mv cri-dockerd/cri-dockerd /usr/local/bin/

wget https://raw.githubusercontent.com/Mirantis/cri-
dockerd/master/packaging/systemd/cri-docker.service
wget https://raw.githubusercontent.com/Mirantis/cri-
dockerd/master/packaging/systemd/cri-docker.socket

mv cri-docker.socket cri-docker.service /etc/systemd/system/
sed -i -e 's,/usr/bin/cri-dockerd,/usr/local/bin/cri-dockerd,'
/etc/systemd/system/cri-docker.service

systemctl daemon-reload
systemctl enable cri-docker.service
systemctl enable --now cri-docker.socket
```

6.2.C.1. Проверка доступности сокета cri-dockerd

На всех узлах выполним следующую команду:

```
crictl --runtime-endpoint unix:///var/run/cri-dockerd.sock version

## Ожидаемый результат:
# Version:  0.1.0
# RuntimeName:  docker
# RuntimeVersion:  23.0.1
# RuntimeApiVersion:  v1
```

6.2.C.2. Проверка возможности Docker запускать контейнеры

На всех узлах выполним следующую команду:

```
docker run hello-world

## Ожидаемый результат:
# …
# Hello from Docker!
# This message shows that your installation appears to be working correctly.
# …
```

Если все тесты прошли успешно, то делаем снимок состояния виртуальных машин и называем его «CRI-DOCKERD».

На этом этап завершен. В зависимости от выбранного варианта на всех узлах должен быть установлен один из контейнерных движков, причем везде он должен быть одинаковым.

6.3. Развёртывание Kubernetes

6.3.A. Вариант A. Установка вырожденного Kubernete кластера

Кластер «все-в-одном» будем разворачивать на узле **node1**, соответственно все приведенные команды будут выполняться на нем же. Процесс разворачивания кластера состоит из следующих этапов:

1. Инициализация кластера Kubernetes.
2. Конфигурирование утилиты управления kubectl.
3. Установка сетевого плагина.
4. Настройка управляющего узла для выполнения рабочих нагрузок.

6.3.A.1. Инициализация кластера Kubernetes

Сначала рассмотрим инициализацию кластера на базе cri-o или containerd. Как это делать для Docker + cri-dockerd, расскажем чуть дальше. Инициализация кластера проводится одной командой:

```
kubeadm init --pod-network-cidr=10.244.0.0/16
```

Здесь в параметре --pod-network-cidr мы явно указываем IP-подсеть, которая будет использоваться "*подами*". Конкретный диапазон 10.244.0.0/16 выбран таким образом, чтобы совпадать с диапазоном по умолчанию для сетевого плагина flannel, который мы будем устанавливать чуть позже.

Для варианта с Docker + cri-dockerd рассмотренную ранее команду нужно чуть-чуть изменить:

```
kubeadm init \
```

```
                --pod-network-cidr=10.244.0.0/16 \
                --cri-socket unix:///var/run/cri-dockerd.sock
```

Здесь мы добавили параметр --cri-socket. Он нужен нам, чтобы указать, через какой Unix сокет Kubernetes должен общаться с контейнерным движком. Для cri-o или containerd мы этого не делали, так как доступный сокет там был всего один, и *kubeadm* об этом знал. Связка Docker + cri-dockerd создает на узле два сокета: один для Docker, а второй для cri-dockerd, поэтому Kubernetes требуется явно указать, какой из них использовать.

6.3.A.2. Конфигурирование утилиты управления kubectl

kubectl – основной рабочий инструмент по управлению кластером Kubernetes. По окончанию инициализации кластера необходимо настроить ее конфигурацию. Поскольку в начале статьи мы договорились, что будем работать от root, то для конфигурирования kubectl выполним следующие команды:

```
echo "export KUBECONFIG=/etc/kubernetes/admin.conf" > /etc/environment
export KUBECONFIG=/etc/kubernetes/admin.conf
```

Выполнять подобную операцию нужно на всех узлах, с которых мы хотим управлять Kubernetes. Для этого, конечно, требуется убедиться, что существует конфигурационный файл */etc/kubernetes/admin.conf*. В текущем случае данный файл был автоматически создан утилитой *kubeadm*, в других случаях его нужно явно поместить на требуемый узел.

6.3.A.3. Установка сетевого плагина

Как уже упоминалось выше, в качестве сетевого плагина мы будем использовать flannel. Тема сетевого взаимодействия в Kubernetes довольно обширна, и ей посвящено большое количество различных статей и документации. Лезть в эти дебри на данном этапе противопоказано, поэтому просто поставим один из самых распространённых сетевых плагинов.

На node1 выполним команду:

```
kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-
flannel.yml
```

6.3.A.4. Настройка управляющего узла для выполнения рабочих нагрузок

Для управления тем, какая нагрузка (рабочая / управляющая) может выполняться на узле, используется механизм «заражения и толерантности» (taint and toleration). Если говорить простыми словами, то это механизм меток. С помощью «заражений» узлу присваиваются определенные метки. Механизм толерантности показывает, с какими метками "*под*" готов

мириться, а с какими он работать не будет. По умолчанию на управляющих узлах Kubernetes устанавливается метка *«node-role.kubernetes.io/control-plane»*. Рабочие "*поды*" без дополнительный настроек на узлах с подобной меткой запускаться не будут.

Для того чтобы разрешить выполнение рабочих "*подов*" на управляющем узле, с последнего нужно снять метку *node-role.kubernetes.io/control-plane*, что может быть сделано командой:

```
kubectl taint nodes --all node-role.kubernetes.io/control-plane-
```

Примечание. Для простоты данная команда снимает метку со всех узлов кластера. Для случая кластера из одной машины это нормально, но в многомашинном варианте рекомендуется все же явно указывать узлы, с которых будут сниматься метки.

Вырожденный кластер «все-в-одном» готов. На узле **node1** можно сделать снимок состояния виртуальной машины и назвать «ALL IN ONE».

6.3.B. Вариант B. Организация отказоустойчивого кластера Kubernetes

Настройку данного варианта следует начать с отката состояния виртуальных машин на момент завершения установки любого из контейнерных движков.

6.3.B.1. Объяснение, за счет чего достигается отказоустойчивость

Организация отказоустойчивости в кластере Kubernetes базируется на решении двух стратегических задач:

1. Отказоустойчивое выполнение рабочих нагрузок.
2. Отказоустойчивое управление кластером.

Решение первой задачи основывается на логике управления кластером. В отличии от Docker, здесь администратор не запускает контейнеры явным образом. Вместо этого он описывает желаемое состояние (desired state) кластера, в котором указывает какие "*поды*" должны быть запущены. Система управления кластером сравнивает текущее состояние с желаемым. Если эти состояния различаются, то система выполняет действия по приведению желаемого к текущему с учетом доступных ресурсов.

Поясним на примере. Администратор во время конфигурации кластера указал, что желает видеть запущенными два "*пода*" с контейнерами nginx. После получения воли высшего существа система управления кластером смотрит – "*подов*" нет, nginx нет – не дела, скачивает nginx из репозитория, настраивает по конфигурации админа "*поды*", затем запускает их. Вот теперь желаемое = текущее. Если далее по каким-то мистическим причинам узел кластера, на котором крутились "*поды*", откажет, то система управления увидит, что текущее состояние отличается от желаемого, и автоматически запустит недостающие "*поды*" на другом доступном узле кластера, вновь делая желаемое = текущее. Вот таким интересным способом и достигается отказоустойчивость выполнения

рабочих нагрузок.

Решение второй задачи разбивается на подзадачи:

1. Организация отказоустойчивого хранения и использования конфигурации кластера.
2. Организация отказоустойчивого доступа к API системы управления кластером.

Первая задача решается путем применения системы распределенного хранения данных etcd. Данная система хранит конфигурацию кластера одновременно на нескольких узлах и обеспечивает непротиворечивость имеющихся копий. Она позволяет продолжать нормальную работу кластера при выходе из строя некоторых узлов с репликами данных, а потом их горячее переподключение после восстановления. При использовании *kubeadm* для настройки кластера вся предварительная настройка etcd полностью ложится на его кремниевые плечи, и нам ничего делать не нужно.

Решение второй задачи базируется на использовании внешнего компонента – балансировщика нагрузки. С его помощью создается виртуальный IP-адрес, запросы к которому автоматически транслируются на один из управляющих узлов кластера. Вся магия заключается в том, что все управляющие узлы кластера равноправны, и кластер может находиться под управлением любого из них. Поэтому и создается виртуальный IP-адрес, через который будет доступен как минимум один управляющий узел, а иногда и все управляющие узлы кластера в режиме очередности (round robin).
К сожалению, настроить балансировщик нагрузки *kubeadm* не может, и эту работу нам придется делать самим.

6.3.B.2. Настройка балансировщика нагрузки

Для создания виртуального IP адреса и перераспределение нагрузки между управляющими узлами будем использовать комбинацию двух демонов: keepalived и haproxy. Этих ребят мы с вами установили ранее на узлы: **node1**, **node2**, **node3**.

Реализуемый нами балансировщик нагрузки будет работать следующим образом (Рисунок 12):

1. Демон keepalived обеспечит функционирование виртуального IP-адреса и его привязку к одному из управляющих узлов. Виртуальный IP будет вторым адресом на сетевом интерфейсе узла. Если данный узел откажет, то keepalived обнаружит это и перекинет виртуальный IP-адрес на другой доступный узел.
2. Поступающие на управляющий узел запросы будут обрабатываться демоном haproxy, который, выполняя роль реверс-прокси (reverse proxy), будет поочередно (round robin) пересылать их на API сервера управляющих узлов Kubernetes.

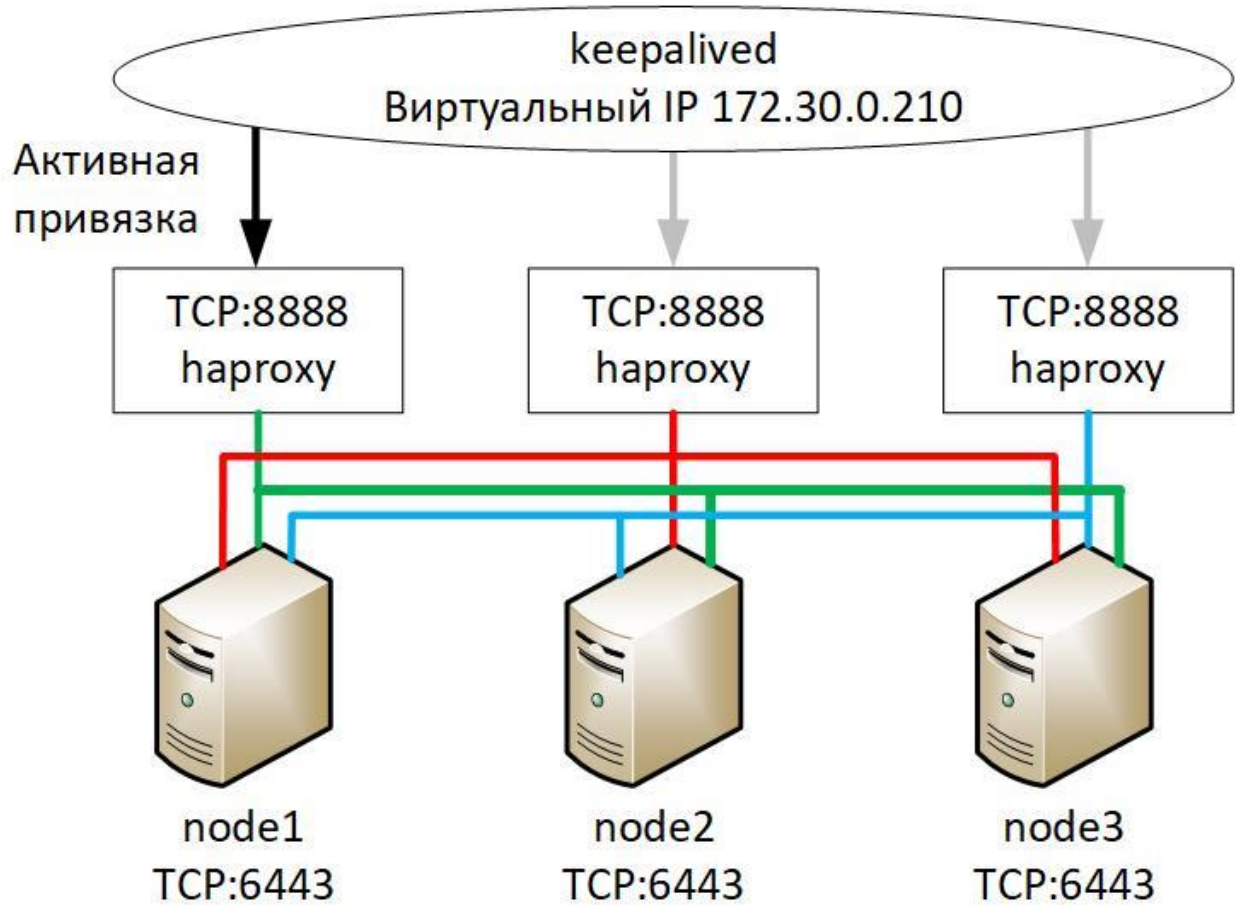

Рисунок 12

С одной стороны, одного keepalived должно хватить для построения полноценного балансировщика нагрузки, но официальный гайд регламентирует его использование в паре с haproxy. Народная мудрость мотивирует использование сразу двух демонов тем, что так якобы лучше балансируется нагрузка. Сложно сказать, так это или нет, на самом деле, но в учебных целях мы все же прислушаемся к рекомендациям и построим балансировщик с использованием обоих демонов.

Как во всем, что касается Kubernetes, демонов можно внедрить различными способами:

1. Демоны могут быть реализованы в виде "*подов*" Kubernetes.
2. Демоны могут устанавливаться отдельно в операционную систему управляющих узлов.

Для большей наглядности выберем второй вариант. При развертывания балансировщика нагрузки будем использовать следующие сетевые настройки:

- в качестве виртуального адреса будет использоваться 172.30.0.210;
- связанное с виртуальным адресом DNS имя: k8s-cp.internal;
- TCP порт для доступа к системе управления: 8888;
- в качестве бэкендов будут использоваться порты 6443 на управляющих узлах, другими словами, 172.30.0.201:6443, 172.30.0.202:6443, 172.30.0.203:6443;

При проведении работ все параметры будут жестко вбиты в конфигурационные файлы и скрипты.

6.3.B.2.1. Настройка демона keepalived

На узлах node1, node2, node3 создадим и отредактируем основной конфигурационный файл демона keepalived */etc/keepalived/keepalived.conf* следующим образом:

```
# File: /etc/keepalived/keepalived.conf

global_defs {
    enable_script_security
    script_user nobody
}

vrrp_script check_apiserver {
  script "/etc/keepalived/check_apiserver.sh"
  interval 3
}

vrrp_instance VI_1 {
    state BACKUP
    interface ens33
    virtual_router_id 5
    priority 100
    advert_int 1
    nopreempt
    authentication {
        auth_type PASS
        auth_pass ZqSj#f1G
    }
    virtual_ipaddress {
        172.30.0.210
    }
    track_script {
        check_apiserver
    }
}
```

На узлах node1, node2, node3 создадим и отредактируем скрипт */etc/keepalived/check_apiserver.sh*, предназначенный для проверки доступности серверов.

```
#!/bin/sh
# File: /etc/keepalived/check_apiserver.sh

APISERVER_VIP=172.30.0.210
APISERVER_DEST_PORT=8888
PROTO=http

errorExit() {
    echo "*** $*" 1>&2
    exit 1
}
```

```
curl --silent --max-time 2 --insecure
${PROTO}://localhost:${APISERVER_DEST_PORT}/ -o /dev/null || errorExit "Error
GET ${PROTO}://localhost:${APISERVER_DEST_PORT}/"
if ip addr | grep -q ${APISERVER_VIP}; then
    curl --silent --max-time 2 --insecure
${PROTO}://${APISERVER_VIP}:${APISERVER_DEST_PORT}/ -o /dev/null || errorExit
"Error GET ${PROTO}://${APISERVER_VIP}:${APISERVER_DEST_PORT}/"
fi
```

На узлах node1, node2, node3 установим атрибут, разрешающий исполнение скрипта, и запустим демона keepalived.

```
chmod +x /etc/keepalived/check_apiserver.sh
systemctl enable keepalived
systemctl start keepalived
```

6.3.B.2.2. Настройка демона haproxy

На узлах node1, node2, node3 отредактируем основной конфигурационный файл демона haproxy */etc/haproxy/haproxy.cfg* следующим образом:

```
# File: /etc/haproxy/haproxy.cfg
#---------------------------------------------------------------------
# Global settings
#---------------------------------------------------------------------
global
    log /dev/log local0
    log /dev/log local1 notice
    daemon

#---------------------------------------------------------------------
# common defaults that all the 'listen' and 'backend' sections will
# use if not designated in their block
#---------------------------------------------------------------------
defaults
    mode                    http
    log                     global
    option                  httplog
    option                  dontlognull
    option http-server-close
    option forwardfor       except 127.0.0.0/8
    option                  redispatch
    retries                 1
    timeout http-request    10s
    timeout queue           20s
    timeout connect         5s
    timeout client          20s
    timeout server          20s
    timeout http-keep-alive 10s
    timeout check           10s

#---------------------------------------------------------------------
# apiserver frontend which proxys to the control plane nodes
#---------------------------------------------------------------------
frontend apiserver
    bind *:8888
    mode tcp
```

```
    option tcplog
    default_backend apiserver

#---------------------------------------------------------------------
# round robin balancing for apiserver
#---------------------------------------------------------------------
backend apiserver
    option httpchk GET /healthz
    http-check expect status 200
    mode tcp
    option ssl-hello-chk
    balance     roundrobin
        server node1 172.30.0.201:6443 check
        server node2 172.30.0.202:6443 check
        server node3 172.30.0.203:6443 check
```

На узлах node1, node2, node3 запустим демона haproxy, выполнив команды:

```
systemctl enable haproxy
systemctl restart haproxy
```

Примечание. Демон будет ругаться, что не обнаружены backend сервера. Это нормально, так как Kubernetes API еще не запущен.

6.3.B.3. Установка управляющих узлов кластера

Установка производится по официальной документации, выбран режим stacked control plane.

Примечание! При использовании Docker + cri-dockerd к строке инициализации нужно добавить параметр: --cri-socket unix:///var/run/cri-dockerd.sock

6.3.B.3.1. Установка первого управляющего узла

На node1
При использовании контейнерных движков cri-o и containerd выполняем следующую команду:

```
kubeadm init \
               --pod-network-cidr=10.244.0.0/16 \
               --control-plane-endpoint "172.30.0.210:8888" \
               --upload-certs
```

Примечание 1. --pod-network-cidr=10.244.0.0/16 выбрано для упрощения дальнейшей установки сетевого плагина flannel.

Примечание 2. --control-plane-endpoint «172.30.0.210:8888» указывает на виртуальный IP адрес, используемый для управления кластером.

На случай использования Docker + cri-dokerd команда инициализации будет выглядеть так:

```
kubeadm init \
               --cri-socket unix:///var/run/cri-dockerd.sock \
               --pod-network-cidr=10.244.0.0/16 \
               --control-plane-endpoint "172.30.0.210:8888" \
               --upload-certs
```

По окончанию процедуры должна появиться строка для добавления **управляющих** узлов в кластер.

```
….
You can now join any number of the control-plane node running the following command
on each as root:

  kubeadm join 172.30.0.210:8888 --token 4uvhjf.pmq742i3rofly0qr \
        --discovery-token-ca-cert-hash
sha256:9cf1614b335f50f8a0014d45534f4ab702319c32111d2124285655cc7cbcdf60 \
        --control-plane --certificate-key
15489535ff4f00324eb23808585d3b9acddf38801069f92bf39f8404c677ffa9
….
```

Кроме указанной строки, будет показана строка для добавления **рабочих** узлов в кластер.

```
...
Then you can join any number of worker nodes by running the following on each as
root:

kubeadm join 172.30.0.210:8888 --token 4uvhjf.pmq742i3rofly0qr \
        --discovery-token-ca-cert-hash
sha256:9cf1614b335f50f8a0014d45534f4ab702319c32111d2124285655cc7cbcdf60
...
```

Примечание. Приведенные выше строки будут изменяться от инсталляции к инсталляции. Здесь они приведены для примера, копировать их отсюда не имеет смысла.

6.3.B.3.2. Установка последующих управляющих узлов

На node2, node3
Используем строку подключения, полученную после создания первого управляющего узла кластера. При использовании в качестве движка связки Docker + cri-dokerd не забудьте добавить к этой строке параметр --cri-socket unix:///var/run/cri-dockerd.sock

Внимание! В строке содержится конфиденциальная информация. Сертификаты для подключения будут автоматически удалены после 2-х часов с момента первичной инициализации кластера.

В случае успешного добавления узла среди вывода *kubeadm* должна быть строка:

```
…
This node has joined the cluster and a new control plane instance was created:
…
```

6.3.B.4. Установка рабочих узлов кластера

На узлах node4, node5 запускаем команду добавления рабочих узлов, полученную при установке первого управляющего узла.

Примечание. Если по каким-то причинам вы ее потеряли, то на любом узле кластера введите команду «kubeadm token create --print-join-command», и она отобразится снова.

6.3.B.5. Настройка kubectl

На узлах node1, node2, node3 выполним команду:

```
echo "export KUBECONFIG=/etc/kubernetes/admin.conf" > /etc/environment
export KUBECONFIG=/etc/kubernetes/admin.conf
```

Примечание. Выполнение этой команды на рабочих узлах не имеет смысла, так как на них отсутствует файл */etc/kubernetes/admin.conf*, автоматически создаваемый *kubeadm* при добавлении управляющего узла.

6.3.B.6. Установка сетевого плагина

Данная процедура полностью повторяет аналогичную для случая вырожденного кластера.

На node1 запускаем команду:

```
kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-
flannel.yml
```

Отказоустойчивый кластер Kubernetes готов. Рекомендуется на всех узлах стенда сделать снимок состояния виртуальных машины и назвать его «HA CLUSTER».

7. Проверка работы кластера Kubernetes

Приведенные ниже проверки можно запускать на любом узле кластера, на котором настроена работа kubectl.

Проверка включения узлов в кластер:

```
kubectl get nodes

## Ожидаемый ответ:
# NAME              STATUS   ROLES           AGE   VERSION
# node1.internal    Ready    control-plane   33m   v1.26.1
# node2.internal    Ready    control-plane   11m   v1.26.1
# node3.internal    Ready    control-plane   11m   v1.26.1
# node4.internal    Ready    <none>          93s   v1.26.1
# node5.internal    Ready    <none>          92s   v1.26.1
```

Примечание. Если узел в кластер добавлен недавно, то ответ команды может отличаться от ожидаемого. Необходимо некоторое время, чтобы новый узел «освоился».

При использовании *kubeadm* все управляющее ПО кластера работает на нем в виде "*подов*". Очень важно, чтобы все "*поды*" работали правильным образом, и не было их циклических перезапусков (restarts).

Проверить состояние "*подов*" можно с помощью команды:

```
kubectl get pods -A

# Ожидаемый результат
# NAMESPACE      NAME                                READY   STATUS    RESTARTS      AGE
# kube-flannel   kube-flannel-ds-2p64k               1/1     Running   0             2m58s
# kube-flannel   kube-flannel-ds-77vxc               1/1     Running   0             17m
# kube-flannel   kube-flannel-ds-9pk6s               1/1     Running   0             12m
# kube-flannel   kube-flannel-ds-dhlzp               1/1     Running   0             11m
# kube-flannel   kube-flannel-ds-sts89               1/1     Running   0             2m58s
# kube-system    coredns-787d4945fb-sx2kd            1/1     Running   0             34m
# kube-system    coredns-787d4945fb-xj2kq            1/1     Running   0             34m
# kube-system    etcd-node1.internal                 1/1     Running   0             34m
# kube-system    etcd-node2.internal                 1/1     Running   0             12m
# kube-system    etcd-node3.internal                 1/1     Running   0             12m
# kube-system    kube-apiserver-node1.internal       1/1     Running   0             34m
```

```
# kube-system    kube-apiserver-node2.internal            1/1     Running   0              12m
# kube-system    kube-apiserver-node3.internal            1/1     Running   0              12m
# kube-system    kube-controller-manager-node1.internal   1/1     Running   1 (12m ago)    34m
# kube-system    kube-controller-manager-node2.internal   1/1     Running   0              12m
# kube-system    kube-controller-manager-node3.internal   1/1     Running   0              12m
# kube-system    kube-proxy-5m2lt                         1/1     Running   0              2m58s
# kube-system    kube-proxy-bwvk6                         1/1     Running   0              2m58s
# kube-system    kube-proxy-d4f89                         1/1     Running   0              12m
# kube-system    kube-proxy-gx9fd                         1/1     Running   0              11m
# kube-system    kube-proxy-w2skj                         1/1     Running   0              34m
# kube-system    kube-scheduler-node1.internal            1/1     Running   1 (12m ago)    34m
# kube-system    kube-scheduler-node2.internal            1/1     Running   0              12m
# kube-system    kube-scheduler-node3.internal            1/1     Running   0              12m
```

Типовые проблемы:

1. Часть "*подов*", например, coredns, запустившись, не переходит в состояние готовности. Это может происходить из-за проблем с сетевым плагином. Например, вы забыли его установить.
2. "*Поды*" циклически перезагружаются. Одной из вероятных причин подобного события при использовании в качестве контейнерного движка containerd является неправильная настройка cgroup драйвера в нем. Переставьте containerd заново, в точности выполнив все указания настоящего гайда.

8. Тестовые запуски "*подов*" в Kubernetes

8.1. Тест 1. Запуск "*пода*" в интерактивном режиме.

Данный тест позаимствован из официальной документации.

```
kubectl run -i --tty busybox --image=busybox -- sh

## Ожидаемый результат:
# If you don't see a command prompt, try pressing enter.
# / #
# / #
```

Вспомогательные команды:

Переподключение к "*поду*" при выходе из интерактивного режима:

```
kubectl attach busybox -i
```

Удаление "*пода*":

```
kubectl delete pod busybox
```

8.2. Тест 2. Запуск NGINX

Тест заключается в запуске Web-сервера nginx и обращения к нему после этого.

```
kubectl create deployment nginx-app --image=nginx
kubectl expose deployment nginx-app --type=NodePort --port=80 --external-
ip=10.10.10.10
sleep 5s
curl http://10.10.10.10

## Ожидаемый результат
# <!DOCTYPE html>
# <html>
# <head>
# <title>Welcome to nginx!</title>
# ...
```

Примечание. Зафиксированы случаи, когда репозиторий накладывает ограничения на возможность скачивания базовых образов, из-за чего тесты проваливаются. Если вы попали в подобную ситуацию, попробуйте сменить IP-адрес, например, с помощью VPN.

9. Диагностика балансировщика нагрузки

Для того чтобы определить узел, на котором сейчас активен виртуальный IP-адрес, поочередно запускаем на управляющих узлах команду:

```
ip a

## Ожидаемый ответ:
# 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen
# 1000
#     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
#    inet 127.0.0.1/8 scope host lo
#       valid_lft forever preferred_lft forever
#    inet6 ::1/128 scope host
#       valid_lft forever preferred_lft forever
#2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
group #default qlen 1000
#   link/ether 00:0c:29:a5:2b:48 brd ff:ff:ff:ff:ff:ff
#    altname enp2s1
#    inet 172.30.0.203/24 brd 172.30.0.255 scope global ens33
#       valid_lft forever preferred_lft forever
### !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
### Строка ниже показывает присвоение виртуального IP-адреса
### vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
#    inet 172.30.0.210/32 scope global ens33
#       valid_lft forever preferred_lft forever
#    inet6 fe80::20c:29ff:fea5:2b48/64 scope link
#    valid_lft forever preferred_lft forever
```

```
# ….
```

Для проверки открытия сетевых сокетов воспользуемся следующей командой:

```
ss -lt

## Ожидаемый результат:
# State   Recv-Q  Send-Q   Local Address:Port    Peer Address:Port Process
# LISTEN  0       4096        127.0.0.1:10248        0.0.0.0:*
# LISTEN  0       4096        127.0.0.1:10249        0.0.0.0:*
# LISTEN  0       4096     172.30.0.202:2379         0.0.0.0:*
# LISTEN  0       4096        127.0.0.1:2379         0.0.0.0:*
# LISTEN  0       4096     172.30.0.202:2380         0.0.0.0:*
# LISTEN  0       4096        127.0.0.1:2381         0.0.0.0:*
# LISTEN  0       4096        127.0.0.1:10257        0.0.0.0:*
# LISTEN  0       4096        127.0.0.1:10259        0.0.0.0:*
# LISTEN  0       128           0.0.0.0:ssh          0.0.0.0:*
# LISTEN  0       4096          0.0.0.0:8888         0.0.0.0:*
# LISTEN  0       4096        127.0.0.1:35099        0.0.0.0:*
# LISTEN  0       4096                *:10250              *:*
# LISTEN  0       4096                *:6443               *:*
# LISTEN  0       4096                *:10256              *:*
# LISTEN  0       128              [::]:ssh             [::]:*
```

Для фактической проверки доступности API по виртуальному IP-адресу можно воспользоваться командой:

```
curl --silent --max-time 2 --insecure https://172.30.0.210:8888/

## Ожидаемый результат:
# {
#   "kind": "Status",
#   "apiVersion": "v1",
#   "metadata": {},
#   "status": "Failure",
#   "message": "forbidden: User \"system:anonymous\" cannot get path \"/\"",
#   "reason": "Forbidden",
#   "details": {},
#   "code": 403
# }
```

X. Заключение

Kubernetes — большая и довольно сложная тема, но маленький первый шаг на пути его покорения вы уже сделали — у вас появился работающий стенд для экспериментов. Дело осталось за малым — продолжать учиться. В этом вам могут помочь следующие статьи:

- Основы Kubernetes
- Just-in-Time Kubernetes: Руководство начинающим для понимания основных концепций Kubernetes
- Различия между Docker, containerd, CRI-O и runc
- Визуальное руководство по диагностике неисправностей в Kubernetes

- Записки о containerd
- Зачем нужен контейнер pause в Kubernetes
- Как я клонировал Томми Версетти, или запускаем GUI/GPU приложения в Kubernetes
- Отказоустойчивый кластер с балансировкой нагрузки с помощью keepalived