Национальный исследовательский ядерный университет «МИФИ» (Московский Инженерно–Физический Институт)

Кафедра №42 «Криптология и кибербезопасность»

**Лабораторная работа №2:**

**«Введение в docker-compose и nginx»**

Антон Гатченко Б22-525

2024 г.

*Используемая рабочая среда:*

* Процессор - AMD Ryzen 5 5600H (laptop), 6c/12t
* Оперативная память – DDR4 16 ГБ
* ОС - Windows 10 Pro 22H2 19045.4780, 64 bit

*Ход работы:*

В ходе работы было проведено тестирование производительности при вертикальном масштабировании. Контейнеру consumer выделялись разные количества ядер процессора, память и скорость сети не ограничивались. По полученным данным была построена гистограмма зависимости времени как работы всего процесса, так и конкретно перемножения матрицы от количества выделенных ядер для разного размера матриц.

Далее было проведено тестирование производительности при горизонтальном масштабировании. Consumer был модифицирован для приема данных, а для их обработки был создан новый контейнер compute, в который данные пересылались одним сообщением со всеми собранными данными от каждого consumer. Изменялось количество consumer, размер матрицы и ограничение по количеству ядер процессора для каждого consumer и compute одновременно, т.к. ранее они были в одном контейнере, и ограничение действовало как на ресурсы сервера-приемщика, так и на вычисление матрицы.

В POST запросы от producer было добавлено дополнительное поле с номером сообщения, чтобы корректно собирать матрицу после отправки в compute с разных consumer.

Для nginx использовалась балансировка по умолчанию – Round Robin, т.к. все серверы-приемщики одинаковые, и нужно было распределить нагрузку равномерно. На больших размерах матрицы зачастую возникали ошибки либо nginx, либо segfault в compute, хотя память выделялась правильно, поэтому таймирование было выполнено для малых размеров.

*Графики выполнения программы при вертикальном масштабировании:*

Результаты при количестве ядер >2 совпадали с результатами при 1-2 ядрах, поэтому они не включены в графики, чтобы улучшить читаемость.

*Оценка преимуществ и недостатков методики вертикального масштабирования:*

Вертикальное масштабирование легко реализуемо, однако ограничено ресурсами одного устройства, и у него есть реальный предел – максимально мощное оборудование.

*Config file для nginx:*

**nginx.conf:**

user nginx; # Указываем пользователя  
worker\_processes auto; # Устанавливаем количество рабочих процессов на автоматическое (все доступные процессоры)  
  
pid /var/run/nginx.pid;  
  
events {  
 worker\_connections 65536; # Устанавливаем максимальное количество соединений (4096 не хватало)  
}  
  
http {  
 # это было взято из файла по умолчанию в nginx контейнере  
 include /etc/nginx/mime.types;  
 default\_type application/octet-stream;  
  
 log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '  
 '$status $body\_bytes\_sent "$http\_referer" '  
 '"$http\_user\_agent" "$http\_x\_forwarded\_for"';  
  
 access\_log /var/log/nginx/access.log main;  
 sendfile on;  
 keepalive\_timeout 65;  
  
 # Подключаем дополнительные конфигурационные файлы (default.conf)  
 include /etc/nginx/conf.d/\*.conf;  
}

**default.conf:**

# Определяем группу серверов для обработки запросов к consumer  
upstream consumers {  
 server consumer1:8080;  
 server consumer2:8080;  
 server consumer3:8080;  
  
 # Используем дефолтную round-robin балансировку  
 # В данном случае NGINX будет поочередно передавать запросы к каждому из указанных серверов  
  
}  
  
# Аналогично для compute  
upstream computers {  
 server compute:8081;  
}  
  
server {  
 # прослушиваем порт 80 для обработки входящих HTTP-запросов  
 listen 80;  
  
 location / {  
 # перенаправляем запросы по пути '/' к upstream "consumers"  
 proxy\_pass http://consumers;  
 access\_log off; # Отключаем логи запросов, чтобы не засоряли терминал  
 }  
  
 # Аналогично для остальных путей запросов  
 location /end {  
 proxy\_pass http://consumers;  
 }  
   
 location /compute {  
 proxy\_pass http://computers;  
 }  
}

*Графики выполнения программы при горизонтальном масштабировании:*

*Оценка преимуществ и недостатков методики горизонтального масштабирования:*

Горизонтальное масштабирование позволяет легко расширять систему, просто добавляя новые серверы. Система становится более отказоустойчивой, хорошо подходит для облачных решений, что сейчас очень распространено. Однако его сложнее реализовать, могут появляться проблемы с синхронизацией данных между серверами, и всю эту большую систему сложнее обслуживать, чем единственное устройство.

*Заключение:*

Общее время работы намного лучше (6-7с против 14с) у варианта с горизонтальным масштабированием при сильном ограничении ресурсов (0.25, 0.5 ядра), поскольку серверов-приемщиков несколько, и сервер-вычислитель помещен в отдельный контейнер, что суммарно дает больше ресурсов. При ограничении в 1+ ядро разница практически исчезает.

Скорость разработки в данной работе намного выше для вертикального масштабирования, потому что этот вариант готов «из коробки», достаточно только менять количество выделяемых ядер процессора в docker-compose.yaml. Вариант с nginx требует изменения логики работы системы, однако позволяет более эффективно распорядиться ресурсами.

По наблюдениям и результатам можно вынести предположение, что узким местом являются producer или сама сеть, поскольку в варианте с nginx всё также оставалось замедление после 11-14 тысяч отправленных запросов с каждого из producer. Из-за этого вариант с 3 consumer вместо 2 выступает чуть хуже, т.к. отправленные с них на compute запросы ждут обработки предыдущего больше (обрабатываются 3 запроса, а не 2), а скорость отправки данных с producer на consumer не меняется. Это подтверждается графиком нагрузки docker stats (приложение 2), где нагрузка на producer постоянно находится около 100-110%, на nginx не превышает 15% в пике, на consumer – 5-10%.

*Теоретическая справка по сетям Docker:*

Docker использует различные типы сетей для обеспечения взаимодействия контейнеров друг с другом, а также между контейнерами и внешним миром. Эти сети позволяют гибко управлять подключениями, доступом и маршрутизацией в инфраструктуре контейнеров.

**1. Основные типы сетей Docker**

Docker предоставляет несколько типов сетей, каждая из которых подходит для разных сценариев использования:

1. **Bridge (Мостовая сеть)**

Это сеть по умолчанию, которая создается для каждого контейнера, если не указана другая сеть. Контейнеры подключаются к этой сети через виртуальный мост (bridge), и они могут общаться друг с другом, но изолированы от внешнего мира. Для доступа извне используется NAT (Network Address Translation), который перенаправляет порты с хоста на контейнер. Пример использования: контейнеры, которые работают в пределах одного хоста и не требуют прямого доступа извне.

1. **Host (Хостовая сеть)**

Контейнеры используют сеть хостовой машины напрямую. Это означает, что контейнеры не имеют собственной сети, а работают на той же сетевой интерфейсе, что и хост. Контейнеры используют те же IP-адреса, что и хост, и могут слушать те же порты. Используется когда нужно минимизировать накладные расходы на виртуализацию сети и у контейнера должен быть полный доступ к сети хоста.

1. **None (Без сети)**

Контейнеры, подключенные к сети типа none, не получают никакой сетевой связи. Используется, если контейнеру не требуется подключение к сети, например, если контейнер работает в изолированном режиме.

1. **Overlay (Оверлейная сеть)**

Это тип сети, который используется для связи между контейнерами, расположенными на разных хостах Docker. Сети Overlay обычно используются в Docker Swarm или Kubernetes для создания распределенных приложений. Эта сеть позволяет контейнерам на разных физических хостах обмениваться данными, как если бы они находились на одном хосте.

1. **Macvlan (Маковлан)**

Сеть типа macvlan позволяет контейнерам иметь свои уникальные MAC-адреса и быть видимыми в физической сети. Это позволяет контейнерам работать как отдельные устройства в сети, с собственным IP-адресом.

**2. Основные параметры сети Docker**

* **IP-адресация**: Каждый контейнер в сети Docker имеет уникальный IP-адрес. Для сетей типа bridge IP-адреса контейнеров присваиваются динамически, а для macvlan контейнер может иметь статический IP-адрес, который виден в физической сети.
* **Port mapping (Проброс портов)**: В случае использования сети bridge или host, Docker позволяет пробрасывать порты с хоста на контейнер, что дает возможность получать доступ к сервисам, запущенным внутри контейнера, через порты хоста.
* **DNS**: Docker поддерживает внутренний DNS, который позволяет контейнерам общаться друг с другом по имени, а не по IP-адресу. Это особенно полезно в случае использования сетей типа overlay, где контейнеры могут быть перемещены между хостами.
* **Виртуальные сети**: В Docker можно создавать собственные виртуальные сети, настраивать маршруты, а также создавать мосты между контейнерами, чтобы обеспечить гибкость и масштабируемость.

**3. Команды для работы с сетями Docker**

* **docker network ls**: отображает список всех доступных сетей.
* **docker network create <network\_name>**: создает новую сеть.
* **docker network inspect <network\_name>**: предоставляет подробную информацию о сети.
* **docker network connect <network\_name> <container\_name>**: подключает контейнер к существующей сети.
* **docker network disconnect <network\_name> <container\_name>**: отключает контейнер от сети.

**4. Docker Compose и сети**

Docker Compose позволяет конфигурировать сети на уровне многоконтейнерных приложений. В файле docker-compose.yml можно определить, какие сети будут использовать контейнеры, и каким образом контейнеры будут взаимодействовать друг с другом.

*Приложение:*

1. git репозиторий с проектом для горизонтального масштабирования:

<https://github.com/Arondy/PARVPO_lab2>

1. Скриншот docker stats:

