10 Network simulation: Realistic environment testing without the pain

This chapter covers

- Coming to grips with Docker Compose
- Testing your applications on troublesome networks
- Taking a first look at Docker network drivers
- Creating a substrate network for seamless communications across Docker hosts

As part of your DevOps workflow, you'll likely need to use the network in some way. Whether you're trying to find out where the local memcache container is, connecting to the outside world, or plumbing together Docker containers running on different hosts, you're likely to want to reach out to the wider network sooner or later.

After reading this chapter you'll know how to manage containers as a unit using Docker Compose, and simulate and manage networks by using Docker's virtual network tooling. This chapter is a small first step toward orchestration and service discovery—subjects we'll take a deeper dive into in part 4 of this book.

10.1 Container communication: Beyond manual linking

In technique 8 you saw how to connect containers with links, and we mentioned the advantages provided by a clear statement of container dependencies. Unfortunately, links have a number of disadvantages. Links have to be manually specified when starting each container, containers have to be started in the correct order, loops in links are forbidden, and there's no way to replace a link (if a container dies, every dependent container must be restarted to recreate the links). On top of everything else, they're deprecated!

Docker Compose is currently the most popular replacement for anything that previously involved a complex links setup, and we'll take a look at it now.

TECHNIQUE 76 A simple Docker Compose cluster

Docker Compose started life as *fig*, a now-deprecated independent effort to ease the pain of starting multiple containers with appropriate arguments for linking, volumes, and ports. Docker Inc. liked this so much that they acquired it, gave it a makeover, and released it with a new name.

This technique introduces you to Docker Compose using a simple example of Docker container orchestration.

PROBLEM

You want to coordinate connected containers on your host machine.

SOLUTION

Use Docker Compose, a tool for defining and running multicontainer Docker applications.

The central idea is that rather than wiring up container startup commands with complex shell scripts or Makefiles, you declare the application's startup configuration, and then bring the application up with a single, simple command.

> **NOTE** We assume you have Docker Compose installed—refer to the official instructions (http://docs.docker.com/compose/install) for the latest advice.

In this technique we're going to keep things as simple as possible with an echo server and client. The client sends the familiar "Hello world!" message every five seconds to the echo server, and then receives the message back.

> **TIP** The source code for this technique is available at https://github.com/docker-in-practice/docker-compose-echo.

The following commands create a directory for you to work in while creating the server image:

```
$ mkdir server
$ cd server
```

Create the server Dockerfile with the code shown in the following listing.

Listing 10.1 Dockerfile—a simple echo server

Installs the nmap package, which provides the ncat program used here

```
.FROM debian
RUN apt-get update && apt-get install -y nmap
 CMD ncat -l 2000 -k --exec /bin/cat
```

Runs the ncat program by default when starting the image

The `-l 2000` arguments instruct `ncat` to listen on port 2000, and `-k` tells it to accept multiple client connections simultaneously and to continue running after clients close their connections so more clients can connect. The final arguments, `--exec /bin/cat`, will make `ncat` run `/bin/cat` for any incoming connections and forward any data coming over the connection to the running program.

Next, build the Dockerfile with this command:

```
$ docker build -t server .
```

Now you can set up the client image that sends messages to the server. Create a new directory and place the client.py file and Dockerfile in there:

```
$ cd ..
$ mkdir client
$ cd client
```

We'll use a simple Python program as the echo server client in the next listing.

Listing 10.2 client.py—a simple echo client

Imports the Python packages needed

```
import socket, time, sys
 while True:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
     s.connect(('talkto',2000))
     s.send('Hello, world\n')
     data = s.recv(1024)
     print 'Received:', data
     sys.stdout.flush()
     s.close()
     time.sleep(5)
```

Creates a socket object

Uses the socket to connect to the 'talkto' server on port 2000

Sends a string with a newline character to the socket

Creates a buffer of 1024 bytes to receive data, and places the data into the data variable when a message is received

Prints the received data to standard out

Flushes the standard out buffer so you can see messages as they come in

Closes the socket object

Waits 5 seconds and repeats

The Dockerfile for the client is straightforward. It installs Python, adds the client.py file, and then defaults it to run on startup, as shown in the following listing.

Listing 10.3 Dockerfile—a simple echo client

```
FROM debian
RUN apt-get update && apt-get install -y python
ADD client.py /client.py
CMD ["/usr/bin/python","/client.py"]
```

Build the client with this command:

```
docker build -t client .
```

To demonstrate the value of Docker Compose, we'll first run these containers by hand:

```
docker run --name echo-server -d server
docker run --name client --link echo-server:talkto client
```

When you're finished, Ctrl-C out of the client, and remove the containers:

```
docker rm -f client echo-server
```

Many things can go wrong even in this trivial example: starting the client first will result in the failure of the application to start; forgetting to remove the containers will result in problems when you try to restart; and naming containers incorrectly will result in failure. These kinds of orchestration problems will only increase as your containers and their architecture get more sophisticated.

Compose helps with this by encapsulating the orchestration of these containers' startup and configuration within a simple text file, managing the nuts and bolts of the startup and shutdown commands for you.

Compose takes a YAML file. You create this in a new directory:

```
cd ..
mkdir docker-compose
cd docker-compose
```

The YAML file's contents are shown in the following listing.

Listing 10.4 docker-compose.yml—Docker Compose echo server and client YAML file

```
version: "3"
 services:
  echo-server:
     image: server
     expose:
     - "2000"

  client:
```

This Docker Compose file follows version 3 of the specification.

The reference names of the running services are their identifiers: echo-server and client, in this case.

Each section must define the image used: the client and server images, in this case.

Exposes the echo-server's port 2000 to other services

```
                 image: client
                 links:
    - echo-server:talkto
```

Each section must define the image used: the client and server images, in this case.

Defines a link to the echo-server. References to "talkto" within the client will be sent to the echo server. The mapping is done by setting up the /etc/hosts file dynamically in the running container.

The syntax of docker-compose.yml is fairly easy to grasp: each service is named under the `services` key, and configuration is stated in an indented section underneath. Each item of configuration has a colon after its name, and attributes of these items are stated either on the same line or on the following lines, beginning with dashes at the same level of indentation.

The key item of configuration to understand here is `links` within the client definition. This is created in the same way as the `docker run` command sets up links, except Compose handles startup order for you. In fact, most of the Docker command-line arguments have direct analogues in the docker-compose.yml syntax.

We used the `image:` statement in this example to define the image used for each service, but you can also get docker-compose to rebuild the required image dynamically by defining the path to the Dockerfile in a `build:` statement. Docker Compose will perform the build for you.

TIP A YAML file is a text configuration file with a straightforward syntax. You can read more about it at http://yaml.org.

Now that all the infrastructure is set up, running the application is easy:

```
$ docker-compose up
Creating dockercompose_server_1...
Creating dockercompose_client_1...
Attaching to dockercompose_server_1, dockercompose_client_1
client_1 | Received: Hello, world
client_1 |
client_1 | Received: Hello, world
client_1 |
```

TIP If you get an error when starting docker-compose that looks like "Couldn't connect to Docker daemon at http+unix://var/run/docker.sock—is it running?" the issue may be that you need to run it with sudo.

When you've seen enough, press Ctrl-C a few times to exit the application. You can bring it up again at will with the same command, without worrying about removing containers. Note that it will output "Recreating" rather than "Creating" if you rerun it.

DISCUSSION

We mentioned a possible need for `sudo` in the previous section—you may want to revisit technique 41 if this applies to you, as it makes using tools that interact with the Docker daemon significantly easier.

Docker Inc. advertises Docker Compose as ready for use in production, either on a single machine as shown here, or deploying it across multiple machines with swarm mode—you'll see how to do this in technique 87.

Now that you've come to grips with Docker Compose, we'll move on to a more complex and real-world scenario for docker-compose: using socat, volumes, and the replacement for links to add server-like functionality to a SQLite instance running on the host machine.

TECHNIQUE 77 A SQLite server using Docker Compose

SQLite doesn't come with any concept of a TCP server by default. By building on previous techniques, this technique provides you with a means of achieving TCP server functionality using Docker Compose.

Specifically, it builds on these previously covered tools and concepts:

- Volumes
- Proxying with socat
- Docker Compose

We'll also introduce the replacement for links—networks.

NOTE This technique requires SQLite version 3 to be installed on your host. We also suggest that you install rlwrap to make line editing friendlier when interacting with your SQLite server (this is optional). These packages are freely available from standard package managers.

The code for this technique is available for download here: https://github.com/docker-in-practice/docker-compose-sqlite.

PROBLEM

You want to efficiently develop a complex application referencing external data on your host using Docker.

SOLUTION

Use Docker Compose.

Figure 10.1 gives an overview of this technique's architecture. At a high level there are two running Docker containers: one responsible for executing SQLite clients, and the other for proxying separate TCP connections to these clients. Note that the container executing SQLite isn't exposed to the host; the proxy container achieves that. This kind of separation of responsibility into discrete units is a common feature of microservices architectures.

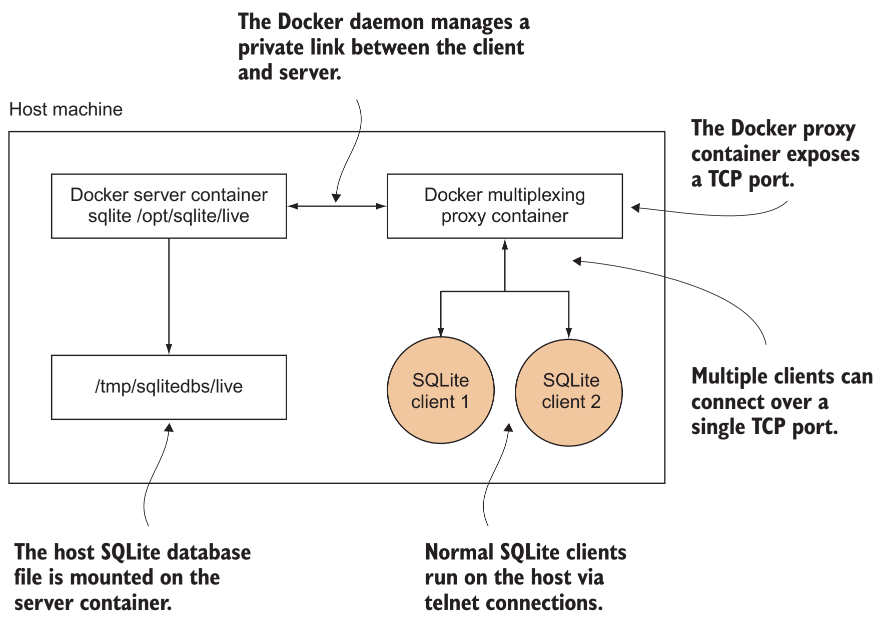

The Docker daemon manages a private link between the client and server.
Host machine
Docker server container sqlite /opt/sqlite/live
Docker multiplexing proxy container
The Docker proxy container exposes a TCP port.
/tmp/sqlitedbs/live
SQLite client 1
SQLite client 2
Multiple clients can connect over a single TCP port.
The host SQLite database file is mounted on the server container.
Normal SQLite clients run on the host via telnet connections.

Figure 10.1 How the SQLite server works

We're going to use the same image for all our nodes. Set up the Dockerfile in the next listing.

Listing 10.5 All-in-one SQLite server, client, and proxy Dockerfile

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install -y rlwrap sqlite3 socat
 EXPOSE 12345
```

Installs required applications

Exposes port 12345 so that the nodes can communicate via the Docker daemon

The following listing shows docker-compose.yml, which defines how the containers should be started up.

Listing 10.6 SQLite server and proxy docker-compose.yml

```
version: "3"
services:
  server:
     command: socat TCP-L:12345,fork,reuseaddr >
EXEC:'sqlite3 /opt/sqlite/db',pty
     build: .
     volumes:
     - /tmp/sqlitedbs/test:/opt/sqlite/db
    networks:
```

The server and proxy containers are defined in this stanza.

Builds the image on startup from the Dockerfile in the same directory

Creates a socat proxy to link the output of a SQLite call to a TCP port

Mounts the test SQLite db file to /opt/sqlite/db within the container

```
    - sqlnet                       ◁ Both services will become part of the sqlnet Docker network.
  proxy:                           ◁ The server and proxy containers are defined in this stanza.
    command: socat TCP-L:12346,fork,reuseaddr TCP:server:12345   ◁ Creates a socat proxy to pass data from port 12346 to the server container's 12345 port
    build: .                       ◁ Builds the image on startup from the Dockerfile in the same directory
    ports:                         ◁ Publishes port 12346 to the host
    - 12346:12346
   networks:
    - sqlnet                       ◁ Both services will become part of the sqlnet Docker network.
networks:                          ◁ Defines a list of networks that containers in this Docker Compose file can join
  sqlnet:
   driver: bridge
```

The socat process in the server container will listen on port 12345 and permit multiple connections, as specified by the `TCP-L:12345,fork,reuseaddr` argument. The part following `EXEC:` tells `socat` to run SQLite on the /opt/sqlite/db file for every connection, assigning a pseudo-terminal to the process. The socat process in the client container has the same listening behavior as the server container (except on a different port), but instead of running something in response to an incoming connection, it will establish a TCP connection to the SQLite server.

One notable difference from the previous technique is the use of networks rather than links—networks present a way to create new virtual networks inside Docker. Docker Compose will always use a new "bridge" virtual network by default; it's just been named explicitly in the preceding Compose configuration. Because any new bridge network allows access to containers using their service names, there's no need to use links (though you still can if you want aliases for services).

Although this functionality could be achieved in one container, the server/proxy container setup allows the architecture of this system to grow more easily, as each container is responsible for one job. The server is responsible for opening SQLite connections, and the proxy is responsible for exposing the service to the host machine.

The following listing (simplified from the original in the repository, https://github.com/docker-in-practice/docker-compose-sqlite) creates two minimal SQLite databases, test and live, on your host machine.

Listing 10.7 setup_dbs.sh

```
#!/bin/bash
echo "Creating directory"
SQLITEDIR=/tmp/sqlitedbs
rm -rf $SQLITEDIR              ◁ Removes any directory from a previous run
 if [ -a $SQLITEDIR ]          ◁ Throws an error if the directory still exists
 then
    echo "Failed to remove $SQLITEDIR"
    exit 1
fi
```

```
mkdir -p $SQLITEDIR
cd $SQLITEDIR
echo "Creating DBs"
echo 'create table t1(c1 text);' | sqlite3 test
 echo 'create table t1(c1 text);' | sqlite3 live
 echo "Inserting data"
echo 'insert into t1 values ("test");' | sqlite3 test
 echo 'insert into t1 values ("live");' | sqlite3 live
 cd - > /dev/null 2>&1
 echo "All done OK"
```

Creates the test DB with one table

Creates the live DB with one table

Inserts one row with the string "test" into the table

Inserts one row with the string "live" into the table

Returns to the previous directory

To run this example, set up the databases and call docker-compose up, as shown in the following listing.

Listing 10.8 Run up the Docker Compose cluster

```
$ chmod +x setup_dbs.sh
$ ./setup_dbs.sh
$ docker-compose up
Creating network "tmpnwxqlnjvdn_sqlnet" with driver "bridge"
Building proxy
Step 1/3 : FROM ubuntu:14.04
14.04: Pulling from library/ubuntu
[...]
Successfully built bb347070723c
Successfully tagged tmpnwxqlnjvdn_proxy:latest
[...]
Successfully tagged tmpnwxqlnjvdn_server:latest
[...]
Creating tmpnwxqlnjvdn_server_1
Creating tmpnwxqlnjvdn_proxy_1 ... done
Attaching to tmpnwxqlnjvdn_server_1, tmpnwxqlnjvdn_proxy_1
```

Then, in one or more other terminals, you can run Telnet to create multiple sessions against one SQLite database.

Listing 10.9 Connecting to the SQLite server

```
$ rlwrap telnet localhost 12346
 Trying 127.0.0.1...
 Connected to localhost.
 Escape character is '^]'.
 SQLite version 3.7.17
 Enter ".help" for instructions
sqlite> select * from t1;
 select * from t1;
test
sqlite>
```

Makes a connection to the proxy using Telnet, wrapped in rlwrap to gain the editing and history functionality of a command line

Output of the Telnet connection

Connects to SQLite here

Runs a SQL command against the sqlite prompt

If you want to switch the server to live, you can change the configuration by changing the `volumes` line in docker-compose.yml from this,

```
- /tmp/sqlitedbs/test:/opt/sqlite/db
```

to this:

```
- /tmp/sqlitedbs/live:/opt/sqlite/db
```

Then rerun this command:

```
$ docker-compose up
```

> **WARNING** Although we did some basic tests with this multiplexing of SQLite clients, we make no guarantees about the data integrity or performance of this server under any kind of load. The SQLite client wasn't designed to work in this way. The purpose of this technique is to demonstrate the general approach of exposing a binary in this way.

This technique demonstrates how Docker Compose can take something relatively tricky and complicated, and make it robust and straightforward. Here we've taken SQLite and given it extra server-like functionality by wiring up containers to proxy SQLite invocations to the data on the host. Managing the container complexity is made significantly easier with Docker Compose's YAML configuration, which turns the tricky matter of orchestrating containers correctly from a manual, error-prone process to a safer, automated one that can be put under source control. This is the beginning of our journey into orchestration, which you'll be hearing much more about in part 4 of the book.

DISCUSSION

Using networks with the `depends_on` feature of Docker Compose allows you to effectively emulate the functionality of links by controlling startup order. For a full treatment of all possible options available to Docker Compose, we recommend you read the official documentation at https://docs.docker.com/compose/compose-file/.

To find out more about Docker virtual networks, take a look at technique 80—it goes into the details of what Docker Compose is doing behind the scenes to set up your virtual networks.

10.2 Using Docker to simulate real-world networking

Most people who use the internet treat it as a black box that somehow retrieves information from other places around the world and puts it on their screens. Sometimes they experience slowness or connection drops, and it's not uncommon to observe cursing of the ISP as a result.

When you build images containing applications that need to be connected, you likely have a much firmer grasp of which components need to connect to where, and how the overall setup looks. But one thing remains constant: you can still experience slowness and connection drops. Even large companies, with data centers they own and operate, have observed unreliable networking and the issues it causes with applications.

We'll look at a couple of ways you can experiment with flaky networks to help determine what problems you may be facing in the real world.

TECHNIQUE 78 Simulating troublesome networks with Comcast

As much as we might wish for perfect network conditions when we distribute applications across many machines, the reality is much uglier—tales of packet loss, connection drops, and network partitions abound, particularly on commodity cloud providers.

It's prudent to test your stack before it encounters these situations in the real world to see how it behaves—an application designed for high availability shouldn't grind to a halt if an external service starts experiencing significant additional latency.

PROBLEM

You want to be able to apply varying network conditions to individual containers.

SOLUTION

Use Comcast (the networking tool, not the ISP).

Comcast (https://github.com/tylertreat/Comcast) is a humorously named tool for altering network interfaces on your Linux machine in order to apply unusual (or, if you're unfortunate, typical!) conditions to them.

Whenever Docker creates a container, it also creates virtual network interfaces—this is how all your containers have different IP addresses and can ping each other. Because these are standard network interfaces, you can use Comcast on them, as long as you can find the network interface name. This is easier said than done.

The following listing shows a Docker image containing Comcast, all its prerequisites, and some tweaks.

Listing 10.10 Preparing to run the `comcast` image

```
$ IMG=dockerinpractice/comcast
$ docker pull $IMG
latest: Pulling from dockerinpractice/comcast
[...]
Status: Downloaded newer image for dockerinpractice/comcast:latest
$ alias comcast="docker run --rm --pid=host --privileged \
-v /var/run/docker.sock:/var/run/docker.sock $IMG"
$ comcast -help
Usage of comcast:
  -cont string
        Container ID or name to get virtual interface of
  -default-bw int
        Default bandwidth limit in kbit/s (fast-lane) (default -1)
  -device string
        Interface (device) to use (defaults to eth0 where applicable)
  -dry-run
        Specifies whether or not to actually commit the rule changes
  -latency int
        Latency to add in ms (default -1)
  -packet-loss string
        Packet loss percentage (e.g. 0.1%)
  -stop
        Stop packet controls
  -target-addr string
        Target addresses, (e.g. 10.0.0.1 or 10.0.0.0/24 or >
10.0.0.1,192.168.0.0/24 or 2001:db8:a::123)
  -target-bw int
        Target bandwidth limit in kbit/s (slow-lane) (default -1)
  -target-port string
        Target port(s) (e.g. 80 or 1:65535 or 22,80,443,1000:1010)
  -target-proto string
        Target protocol TCP/UDP (e.g. tcp or tcp,udp or icmp) (default >
"tcp,udp,icmp")
  -version
        Print Comcast's version
```

The tweaks we added here provide the `-cont` option, which allows you to refer to a container rather than having to find the name of a virtual interface. Note that we've had to add some special flags to the `docker run` command in order to give the container more permissions—this is so Comcast is freely able to examine and apply changes to network interfaces.

To see the difference Comcast can make, we'll first find out what a normal network connection looks like. Open a new terminal and run the following commands to set your expectations for baseline network performance:

The connection between this machine and www.example.com seems to be reliable, with no packets lost.

```
$ docker run -it --name c1 ubuntu:14.04.2 bash
root@0749a2e74a68:/# apt-get update && apt-get install -y wget
[...]
root@0749a2e74a68:/# ping -q -c 5 www.example.com
PING www.example.com (93.184.216.34) 56(84) bytes of data.

--- www.example.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, >
time 4006ms
 rtt min/avg/max/mdev = 86.397/86.804/88.229/0.805 ms
 root@0749a2e74a68:/# time wget -o /dev/null https://www.example.com

real    0m0.379s
 user    0m0.008s
sys     0m0.008s
root@0749a2e74a68:/#
```

The average round trip time is about 100 ms for www.example.com.

The total time taken to download the HTML homepage of www.example.com is about 0.7 s.

Once you've done this, leave the container running and you can apply some network conditions to it:

```
$ comcast -cont c1 -default-bw 50 -latency 100 -packet-loss 20%
Found interface veth62cc8bf for container 'c1'
sudo tc qdisc show | grep "netem"
sudo tc qdisc add dev veth62cc8bf handle 10: root htb default 1
sudo tc class add dev veth62cc8bf parent 10: classid 10:1 htb rate 50kbit
sudo tc class add dev veth62cc8bf parent 10: classid 10:10 htb rate 1000000kb
➥ it
sudo tc qdisc add dev veth62cc8bf parent 10:10 handle 100: netem delay 100ms
➥ loss 20.00%
sudo iptables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p tcp
sudo iptables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p udp
sudo iptables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p icmp
sudo ip6tables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p tcp
sudo ip6tables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p udp
sudo ip6tables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p icmp
Packet rules setup...
Run `sudo tc -s qdisc` to double check
Run `comcast --device veth62cc8bf --stop` to reset
```

The preceding command applies three different conditions: 50 KBps bandwidth cap for all destinations (to bring back memories of dial-up), an added latency of 100 ms (on top of any inherent delay), and a packet loss percentage of 20%.

Comcast first identifies the appropriate virtual network interface for the container and then invokes a number of standard Linux command-line networking utilities to apply the traffic rules, listing what it's doing as it goes along. Let's see how our container reacts to this:

```
root@0749a2e74a68:/# ping -q -c 5 www.example.com
PING www.example.com (93.184.216.34) 56(84) bytes of data.

--- www.example.com ping statistics ---
5 packets transmitted, 2 received, 60% packet loss, time 4001ms
rtt min/avg/max/mdev = 186.425/189.429/195.008/3.509 ms
root@0749a2e74a68:/# time wget -o /dev/null https://www.example.com

real    0m1.993s
user    0m0.011s
sys     0m0.011s
```

Success! An additional 100 ms of latency is reported by ping, and the timing from `wget` shows a slightly greater than 5x slowdown, approximately as expected (the bandwidth cap, latency addition, and packet loss will all impact on this time). But there's something odd about the packet loss—it seems to be three times greater than expected. It's important to bear in mind that the ping is sending a few packets, and that packet loss isn't a precise "one in five" counter—if you increase the ping count to 50, you'll find that the resulting loss is much closer to what's expected.

Note that the rules we've applied apply to *all* network connections via this network interface. This includes connections to the host and other containers.

Let's now instruct Comcast to remove the rules. Comcast is sadly not yet able to add and remove individual conditions, so altering anything on a network interface means completely removing and re-adding rules on the interface. You also need to remove the rules if you want to get your normal container network operation back.

Don't worry about removing them if you exit the container, though—they'll be automatically deleted when Docker deletes the virtual network interface.

```
$ comcast -cont c1 -stop
Found interface veth62cc8bf for container 'c1'
[...]
Packet rules stopped...
Run `sudo tc -s qdisc` to double check
Run `comcast` to start
```

If you want to get your hands dirty, you can dig into Linux traffic control tools, possibly using Comcast with `-dry-run` to generate example sets of commands to use. A full treatment of the possibilities is outside the scope of the technique, but remember, if you can put it in a container, and it hits the network, you can toy with it.

DISCUSSION

With some implementation effort, there's no reason you can't use Comcast for more than just manual control of container bandwidth. For example, imagine you were using a tool like btsync (technique 35) but wanted to limit the available bandwidth so it doesn't saturate your connection—download Comcast, put it in the container, and use `ENTRYPOINT` (technique 49) to set up the bandwidth limits as part of container startup.

To do this, you'll need to install the dependencies of Comcast (listed for the `alpine` image in our Dockerfile at https://github.com/docker-in-practice/docker-comcast/blob/master/Dockerfile) and likely give at least network admin capabilities to the container—you can read more about capabilities in technique 93.

TECHNIQUE 79 Simulating troublesome networks with Blockade

Comcast is an excellent tool with a number of applications, but there's an important use case it doesn't solve—how can you apply network conditions to containers en masse? Manually running Comcast against tens of containers would be painful, and hundreds would be unthinkable! This is a particularly relevant problem for containers, because they're so cheap to spin up—if you're trying to run a large network simulation on a single machine with hundreds of VMs rather than containers, you may find you have bigger problems, like a lack of memory!

On the subject of simulating a network with many machines, there's a particular kind of network failure that becomes interesting at this scale—a network partition. This is when a group of networked machines splits into two or more parts, such that all machines in the same part can talk to each other, but different parts can't communicate. Research indicates that this happens more than you might think, particularly on consumer-grade clouds!

Going down the classic Docker microservices route brings these problems into sharp relief, and having the tools to do experiments is crucial for understanding how your service will deal with it.

PROBLEM

You want to orchestrate setting network conditions for large numbers of containers, including creating network partitions.

SOLUTION

Use Blockade (https://github.com/worstcase/blockade)—an open source piece of software originally from a team at Dell, created for "testing network failures and partitions."

Blockade works by reading a configuration file (blockade.yml) in your current directory that defines how to start containers and what conditions to apply to them. In order to apply conditions, it may download other images with required utilities installed. The full configuration details are available in the Blockade documentation, so we'll only cover the essentials.

First you need to create a blockade.yml file.

Listing 10.11 The blockade.yml file

```
containers:
  server:
    container_name: server
    image: ubuntu:14.04.2
    command: /bin/sleep infinity

  client1:
    image: ubuntu:14.04.2
    command: sh -c "sleep 5 && ping server"

  client2:
    image: ubuntu:14.04.2
    command: sh -c "sleep 5 && ping server"

network:
  flaky: 50%
  slow: 100ms
  driver: udn
```

The containers in the preceding configuration are set up to represent a server being connected to by two clients. In practice, this could be something like a database server with client applications, and there's no inherent reason you have to limit the number of components you want to model. Chances are, if you can represent it in a compose .yml file (see technique 76), you can probably model it in Blockade.

We've specified the network driver as `udn` here—this makes Blockade mimic the behavior of Docker Compose in technique 77, creating a new virtual network so containers can ping each other by container name. To this end, we've had to explicitly specify `container_name` for the server, as Blockade generates one itself by default. The `sleep 5` commands are to make sure the server is running before starting the clients—if you prefer to use links with Blockade, they will ensure the containers are started up in the correct order. Don't worry about the `network` section for now; we'll come back to it shortly.

As usual, the first step in using Blockade is to pull the image:

```
$ IMG=dockerinpractice/blockade
$ docker pull $IMG
latest: Pulling from dockerinpractice/blockade
[...]
Status: Downloaded newer image for dockerinpractice/blockade:latest
$ alias blockade="docker run --rm -v \$PWD:/blockade \
-v /var/run/docker.sock:/var/run/docker.sock $IMG"
```

You'll notice that we're missing a few arguments to `docker run`, compared to the previous technique (like `--privileged` and `--pid=host`). Blockade uses other containers to perform the network manipulation, so it doesn't need permissions itself. Also note the argument to mount the current directory into the container, so that Blockade is able to access blockade.yml and store state in a hidden folder.

NOTE If you're running on a networked filesystem, you may encounter strange permission issues when you start Blockade for the first time—this is likely because Docker is trying to create the hidden state folder as root, but the networked filesystem isn't cooperating. The solution is to use a local disk.

Finally we come to the moment of truth—running Blockade. Make sure you're in the directory you've saved blockade.yml into:

```
$ blockade up
NODE     CONTAINER ID   STATUS  IP          NETWORK  PARTITION
client1  613b5b1cdb7d   UP      172.17.0.4  NORMAL
client2  2aeb2ed0dd45   UP      172.17.0.5  NORMAL
server   53a7fa4ce884   UP      172.17.0.3  NORMAL
```

NOTE On startup, Blockade may sometimes give cryptic errors about files in /proc not existing. The first thing to check is whether a container has immediately exited on startup, preventing Blockade from checking its network status. Additionally, try to resist any temptation to use the Blockade `-c` option to specify a custom path to the config file—only subdirectories of the current directory are available inside the container.

All of the containers defined in our config file have been started, and we've been given a bunch of helpful information about the started containers. Let's now apply some basic network conditions. Tail the logs of client1 in a new terminal (with `docker logs -f 613b5b1cdb7d`) so you can see what happens as you change things:

```
$ blockade flaky --all
$ sleep 5
$ blockade slow client1
$ blockade status
```

Makes the network flaky (drops packets) for all containers

Delays the next command to give the previous one time to take effect and log some output

Makes the network slow (adds a delay to packets) for container client1

Checks the status the containers are in

Reverts all the containers to normal operation

```
 NODE     CONTAINER ID   STATUS  IP           NETWORK  PARTITION
client1  613b5b1cdb7d   UP      172.17.0.4   SLOW
client2  2aeb2ed0dd45   UP      172.17.0.5   FLAKY
server   53a7fa4ce884   UP      172.17.0.3   FLAKY
$ blockade fast --all
```

The `flaky` and `slow` commands use the values defined in the `network` section of the preceding configuration file (listing 10.11)—there's no way to specify a limit on the command line. If you want, it's possible to edit blockade.yml while containers are running and then selectively apply the new limits to containers. Be aware that a container can either be on a slow *or* flaky network, not both. These limitations aside, the convenience of running this against hundreds of containers is fairly significant.

If you look back at your logs from `client1`, you should now be able to see when the different commands took effect:

`icmp_seq` starts skipping numbers—the `flaky` command has taken effect.

`icmp_seq` is sequential (no packets being dropped) and `time` is low (a small delay).

```
64 bytes from 172.17.0.3: icmp_seq=638 ttl=64 time=0.054 ms
 64 bytes from 172.17.0.3: icmp_seq=639 ttl=64 time=0.098 ms
64 bytes from 172.17.0.3: icmp_seq=640 ttl=64 time=0.112 ms
64 bytes from 172.17.0.3: icmp_seq=645 ttl=64 time=0.112 ms
 64 bytes from 172.17.0.3: icmp_seq=652 ttl=64 time=0.113 ms
64 bytes from 172.17.0.3: icmp_seq=654 ttl=64 time=0.115 ms
64 bytes from 172.17.0.3: icmp_seq=660 ttl=64 time=100 ms
 64 bytes from 172.17.0.3: icmp_seq=661 ttl=64 time=100 ms
64 bytes from 172.17.0.3: icmp_seq=662 ttl=64 time=100 ms
64 bytes from 172.17.0.3: icmp_seq=663 ttl=64 time=100 ms
```

`time` has taken a big jump—the `slow` command has taken effect.

All this is useful, but it's nothing we couldn't have done already with some (likely painful) scripting on top of Comcast, so let's take a look at the killer feature of Blockade—network partitions:

```
$ blockade partition server client1,client2
$ blockade status
NODE     CONTAINER ID   STATUS  IP           NETWORK  PARTITION
client1  613b5b1cdb7d   UP      172.17.0.4   NORMAL   2
client2  2aeb2ed0dd45   UP      172.17.0.5   NORMAL   2
server   53a7fa4ce884   UP      172.17.0.3   NORMAL   1
```

This has put our three nodes in two boxes—the server in one and clients in the other—with no way of communicating between them. You'll see that the log for `client1` has stopped doing anything because all of the ping packets are being lost. The clients can still talk to each other, though, and you can verify this by sending a few ping packets between them:

```
$ docker exec 613b5b1cdb7d ping -qc 3 172.17.0.5
PING 172.17.0.5 (172.17.0.5) 56(84) bytes of data.
```

```
--- 172.17.0.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2030ms
rtt min/avg/max/mdev = 0.109/0.124/0.150/0.018 ms
```

No packet loss, low delay … looks like a good connection. Partitions and other network conditions operate independently, so you can play with packet loss while your apps are partitioned. There's no limit to the number of partitions you can define, so you can play with complex scenarios to your heart's content.

DISCUSSION

If you need more power than Blockade and Comcast can individually provide, you can combine the two. Blockade is excellent at creating partitions and doing the heavy lifting of starting up containers; adding Comcast to the mix gives you fine-grained control over the network connections of each and every container.

It's also worth looking into the complete help for Blockade—it offers other things you may find useful, like "chaos" functionality to impact random containers with assorted conditions and a `--random` argument to commands so you could (for example) see how your application reacts when containers are killed at random. If you've heard of Netflix's Chaos Monkey, this is a way to mimic it on a smaller scale.

10.3 Docker and virtual networks

Docker's core functionality is all about isolation. Previous chapters have shown some of the benefits of process and filesystem isolation, and in this chapter you've seen network isolation.

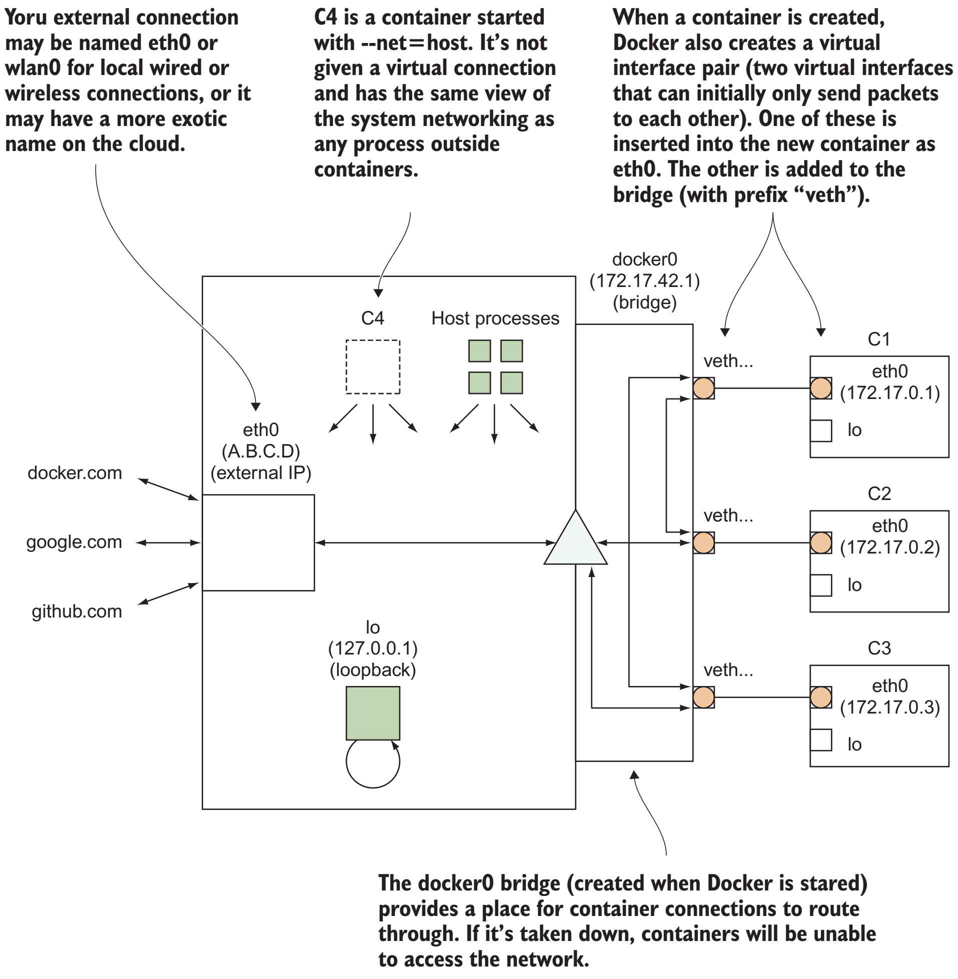
You could think of there being two aspects to network isolation:

- *Individual sandbox*—Each container has its own IP address and set of ports to listen on without stepping on the toes of other containers (or the host).
- *Group sandbox*—This is a logical extension of the individual sandbox—all of the isolated containers are grouped together in a private network, allowing you to play around without interfering with the network your machine lives on (and incurring the wrath of your company network administrator!).

The previous two techniques provide some practical examples of these two aspects of network isolation—Comcast manipulated individual sandboxes to apply rules to each container, whereas partitioning in Blockade relied on the ability to have complete oversight of the private container network to split it into pieces. Behind the scenes, it looks a bit like figure 10.2.

The exact details of how the bridge works aren't important. Suffice it to say that the bridge creates a flat network between containers (it allows direct communication with no intermediate steps) and it forwards requests to the outside world to your external connection.

Docker Inc. then altered this model based on feedback from users to allow you to create your own virtual networks with *network drivers*, a plugin system to extend the networking capabilities of Docker. These plugins are either built-in or provided by third



Yoru external connection may be named eth0 or wlan0 for local wired or wireless connections, or it may have a more exotic name on the cloud.

C4 is a container started with --net=host. It's not given a virtual connection and has the same view of the system networking as any process outside containers.

When a container is created, Docker also creates a virtual interface pair (two virtual interfaces that can initially only send packets to each other). One of these is inserted into the new container as eth0. The other is added to the bridge (with prefix "veth").

The docker0 bridge (created when Docker is stared) provides a place for container connections to route through. If it's taken down, containers will be unable to access the network.

Figure 10.2 Internal Docker networking on a host machine

parties and should do all the necessary work to wire up the network, letting you get on with using it.

New networks you create can be thought of as additional group sandboxes, typically providing access within the sandbox but not allowing cross-sandbox communication (though the precise details of the network behavior depend on the driver).

TECHNIQUE 80 Creating another Docker virtual network

When people first learn about the ability to create their own virtual networks, one common response is to ask how they can create a copy of the default Docker bridge, to allow sets of containers to communicate but be isolated from other containers.

Docker Inc. realized this would be a popular request, so it was implemented as one of the first features of virtual networks in the initial experimental release.

PROBLEM

You want a solution supported by Docker Inc. for creating virtual networks.

SOLUTION

Use the set of Docker subcommands nested under `docker network` to create your own virtual network.

The built-in "bridge" driver is probably the most commonly used driver—it's officially supported and allows you to create fresh copies of the default built-in bridge. There's one important difference we'll look at later in this technique, though—in non-default bridges, you can ping containers by name.

You can see the list of built-in networks with the `docker network ls` command:

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
100ce06cd9a8        bridge              bridge              local
d53919a3bfa1        host                host                local
2d7fcd86306c        none                null                local
```

Here you can see the three networks that are always available for containers to join on my machine. The `bridge` network is where containers end up by default, with the ability to talk to other containers on the bridge. The `host` network specifies what happens when you use `--net=host` when starting a container (the container sees the network as any normal program running on your machine would), and `none` corresponds to `--net=none`, a container that only has the loopback interface.

Let's add a new `bridge` network, providing a new flat network for containers to freely communicate in:

```
$ docker network create --driver=bridge mynet
770ffbc81166d54811ecf9839331ab10c586329e72cea2eb53a0229e53e8a37f
$ docker network ls | grep mynet
770ffbc81166        mynet               bridge              local
$ ip addr | grep br-
522: br-91b29e0d29d5: <NO-
     CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group
➥ default
    inet 172.18.0.1/16 scope global br-91b29e0d29d5
$ ip addr | grep docker
5: docker0: <NO-
     CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group
➥ default
    inet 172.17.0.1/16 scope global docker0
```

This has created a new network interface that will use a different IP address range than the normal Docker bridge. For bridges, the new network interface name will currently begin with `br-`, but this may change in future.

Let's now start up two containers attached to the network:

Starts a container with name c1 (on the default bridge)

Connects container c1 with the mynet network

```
$ docker run -it -d --name c1 ubuntu:14.04.2 bash
 87c67f4fb376f559976e4a975e3661148d622ae635fae4695747170c00513165
$ docker network connect mynet c1
 $ docker run -it -d --name c2 \
--net=mynet ubuntu:14.04.2 bash
 0ee74a3e3444f27df9c2aa973a156f2827bcdd0852c6fd4ecfd5b152846dea5b
$ docker run -it -d --name c3 ubuntu:14.04.2 bash
```

Starts a container with name c3 (on the default bridge)

Creates a container named c2 inside the mynet network

The preceding commands demonstrate two different ways of connecting a container to a network—starting the container and then attaching the service, and creating and attaching in one step.

There's a difference between these two. The first will join the default network on startup (usually the Docker bridge, but this is customizable with an argument to the Docker daemon), and then will add a new interface so it can access mynet as well. The second will *just* join mynet—any containers on the normal Docker bridge will be unable to access it.

Let's do some connectivity checks. First we should take a look at the IP addresses of our container:

Lists the interfaces and IP addresses for c1—one on the default bridge, one in mynet

Lists the interface and IP address for c2, inside mynet

Lists the interface and IP address for c3, on the default bridge

```
$ docker exec c1 ip addr | grep 'inet.*eth'
     inet 172.17.0.2/16 scope global eth0
    inet 172.18.0.2/16 scope global eth1
$ docker exec c2 ip addr | grep 'inet.*eth'
     inet 172.18.0.3/16 scope global eth0
$ docker exec c3 ip addr | grep 'inet.*eth'
    inet 172.17.0.3/16 scope global eth0
```

Now we can do some connectivity tests:

Attempts to ping the name for container 1 from container 2 (success)

```
$ docker exec c2 ping -qc1 c1
 PING c1 (172.18.0.2) 56(84) bytes of data.

--- c1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.041/0.041/0.041/0.000 ms
$ docker exec c2 ping -qc1 c3
 ping: unknown host c3
$ docker exec c2 ping -qc1 172.17.0.3
 PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.

--- 172.17.0.3 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
$ docker exec c1 ping -qc1 c2
 PING c2 (172.18.0.3) 56(84) bytes of data.
```

Attempts to ping the name and IP address for container 3 from container 2 (failure)

Attempts to ping the name for container 2 from container 1 (success)

```
--- c2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.047/0.047/0.047/0.000 ms
$ docker exec c1 ping -qc1 c3
 ping: unknown host c3
$ docker exec c1 ping -qc1 172.17.0.3
 PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.

--- 172.17.0.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.095/0.095/0.095/0.000 ms
```

Attempts to ping the name and IP address for container 3 from container 1 (failure, success)

There's a lot going on here! These are the key takeaways:

- On the new bridge, containers can ping each other with IP address and name.
- On the default bridge, containers can only ping each other by IP address.
- Containers straddling multiple bridges can access containers from any network they're a member of.
- Containers can't access each other at all across bridges, even with an IP address.

DISCUSSION

This new bridge creation functionality was used in technique 77 with Docker Compose and technique 79 with Blockade to provide the ability for containers to ping each other by name. But you've also seen that this is a highly flexible piece of functionality with the potential to model reasonably complex networks.

For example, you might want to experiment with a *bastion host*, a single locked-down machine that provides access to another higher-value network. By putting your application services in a new bridge and then only exposing services via a container connected to both the default and new bridges, you can start running somewhat realistic penetration tests while staying isolated on your own machine.

TECHNIQUE 81 Setting up a substrate network with Weave

A substrate network is a software-level network layer built on top of another network. In effect, you end up with a network that appears to be local, but under the hood it's communicating across other networks. This means that performance-wise, the network will behave less reliably than a local network, but from a usability point of view it can be a great convenience: you can communicate with nodes in completely different locations as though they were in the same room.

This is particularly interesting for Docker containers—containers can be seamlessly connected across hosts in the same way as connecting hosts across networks. Doing this removes any urgent need to plan how many containers you can fit on a single host.

PROBLEM

You want to seamlessly communicate between containers across hosts.

SOLUTION

Use Weave Net (referred to as just "Weave" for the rest of this technique) to set up a network that lets containers talk to each other as if they're on a local network together.

We're going to demonstrate the principle of a substrate network with Weave (https://www.weave.works/oss/net/), a tool designed for this purpose. Figure 10.3 shows an overview of a typical Weave network.

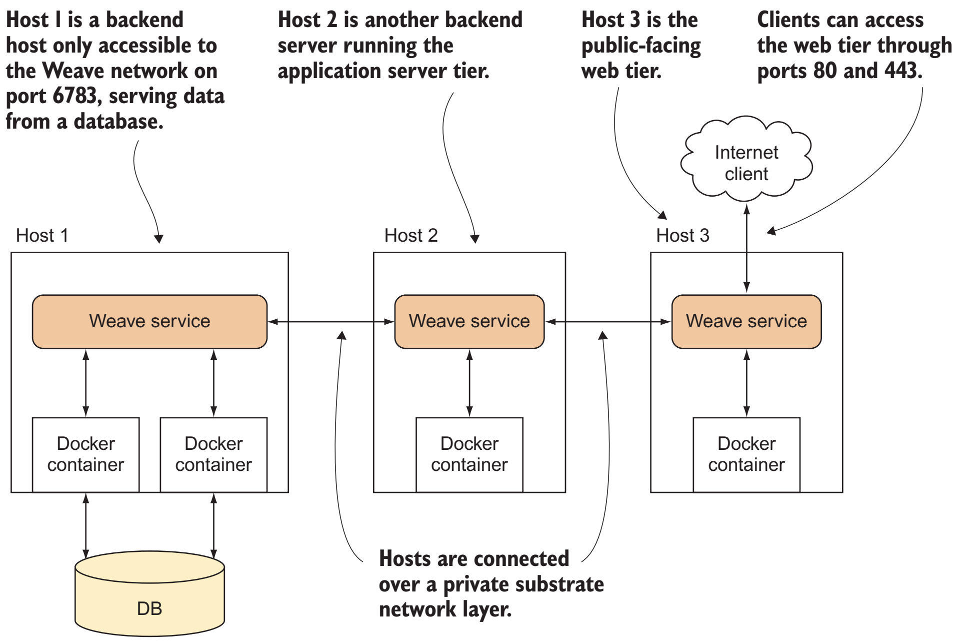

Host 1 is a backend host only accessible to the Weave network on port 6783, serving data from a database.
Host 2 is another backend server running the application server tier.
Host 3 is the public-facing web tier.
Clients can access the web tier through ports 80 and 443.
Internet client
Host 1
Host 2
Host 3
Weave service
Weave service
Weave service
Docker container
Docker container
Docker container
Docker container
DB
Hosts are connected over a private substrate network layer.

Figure 10.3 A typical Weave network

In figure 10.3, host 1 has no access to host 3, but they can talk to each other over the Weave network as though they were locally connected. The Weave network isn't open to the public—only to those containers started up under Weave. This makes the development, testing, and deployment of code across different environments relatively straightforward, because the network topology can be made the same in each case.

INSTALLING WEAVE

Weave is a single binary. You can find installation instructions at https://www.weave.works/docs/net/latest/install/installing-weave/.

The instructions in the preceding link (and listed below for convenience) worked for us. Weave needs to be installed on every host that you want to be part of your Weave network:

```
$ sudo curl -L git.io/weave -o /usr/local/bin/weave
$ sudo chmod +x /usr/local/bin/weave
```

WARNING If you experience problems with this technique, there may already be a Weave binary on your machine that's part of another software package.

Setting up Weave

To follow this example, you'll need two hosts. We'll call them `host1` and `host2`. Make sure they can talk to each other by using ping. You'll need the IP address of the first host you start Weave on.

A quick way to get a host's public IP address is by accessing https://ifconfig.co/ with a browser, or by running `curl https://ifconfig.co`, but be aware that you'll probably need to open firewalls for the two hosts to connect across the open internet. You can also run Weave on a local network if you select the correct IP address.

TIP If you experience problems with this technique, it's likely that the network is firewalled in some way. If you're not sure, talk to your network administrator. Specifically, you'll need to have port 6783 open for both TCP and UDP, and 6784 for UDP.

On the first host, you can run the first Weave router:

Determines host1's IP address

Launches the Weave service on host1. This needs to be done once on each host, and it will download and run some Docker containers to run in the background to manage the substrate network.

Sets up the docker command in this shell to use Weave. If you close your shell or open a new one, you'll need to run this command again.

Starts the container

Retrieves the IP address of the container on the Weave network

```
host1$ curl https://ifconfig.co
 1.2.3.4
host1$ weave launch
 [...]
host1$ eval $(weave env)
 host1$ docker run -it --name a1 ubuntu:14.04 bash
 root@34fdd53a01ab:/# ip addr show ethwe
 43: ethwe@if44: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1376 qdisc noqueue
➥ state UP group default
    link/ether 72:94:41:e3:00:df brd ff:ff:ff:ff:ff:ff
    inet 10.32.0.1/12 scope global ethwe
       valid_lft forever preferred_lft forever
```

Weave takes care of inserting an additional interface into the container, `ethwe`, which provides an IP address on the Weave network.

You can perform similar steps on `host2`, but telling Weave about the location of host1:

Launches the Weave service on host2 as root. This time you add the first host's public IP address so it can attach to the other host.

Sets up your environment appropriately for Weave's service

Continues as with host1

```
host2$ sudo weave launch 1.2.3.4
 host2$ eval $(weave env)
 host2$ docker run -it --name a2 ubuntu:14.04 bash
 root@a2:/# ip addr show ethwe
 553: ethwe@if554: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1376 qdisc noqueue
➥ state UP group default
```

```
    link/ether fe:39:ca:74:8a:ca brd ff:ff:ff:ff:ff:ff
    inet 10.44.0.0/12 scope global ethwe
       valid_lft forever preferred_lft forever
```

The only difference on `host2` is that you tell Weave that it should peer with the Weave on `host1` (specified with the IP address or hostname, and optional `:port`, by which host2 can reach it).

TESTING YOUR CONNECTION

Now that you've got everything set up, you can test whether your containers can talk to each other. Let's take the container on `host2`:

```
root@a2:/# ping -qc1 10.32.0.1
 PING 10.32.0.1 (10.32.0.1) 56(84) bytes of data.

--- 10.32.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
 rtt min/avg/max/mdev = 1.373/1.373/1.373/0.000 ms
```

Pings the other server's assigned IP address

A successful ping response

If you get a successful ping, you've proven connectivity within your self-assigned private network spanning two hosts. You're also able to ping by container name, as with a custom bridge.

TIP It's possible that this won't work due to ICMP protocol (used by ping) messages being blocked by a firewall. If this doesn't work, try telnetting to port 6783 on the other host to test whether connections can be made.

DISCUSSION

A substrate network is a powerful tool for imposing some order on the occasionally chaotic world of networks and firewalls. Weave even claims to intelligently route your traffic across partially partitioned networks, where some host B can see A and C, but A and C can't talk—this may be familiar from technique 80. That said, bear in mind that sometimes these complex network setups exist for a reason—the whole point of bastion hosts is isolation for the sake of security.

All of this power comes at a cost—there are reports of Weave networks sometimes being significantly slower than "raw" networks, and you have to run additional management machinery in the background (because the plugins model for networks doesn't cover all use cases).

The Weave network has many additional pieces of functionality, from visualization to integration with Kubernetes (we'll introduce Kubernetes as an orchestrator in technique 88). We recommend you look at the Weave Net overview to find out more and get the most out of your network—https://www.weave.works/docs/net/latest/overview/.

One thing we haven't covered here is the built-in overlay network plugin. Depending on your use case, this may be worth some research as a possible replacement for Weave, though it requires either use of Swarm mode (technique 87) or setting up a globally accessible key/value store (perhaps etcd, from technique 74).

Summary

- Docker Compose can be used to set up clusters of containers.
- Comcast and Blockade are both useful tools for testing containers in bad networks.
- Docker virtual networks are an alternative to linking.
- You can manually model networks in Docker with virtual networks.
- Weave Net is useful for stringing containers together across hosts.