blockade Documentation

Release 0.1.0

Dell Cloud Manager

January 28, 2015

Blockade is a utility for testing network failures and partitions in distributed applications. Blockade uses Docker containers to run application processes and manages the network from the host system to create various failure scenarios.

A common use is to run a distributed application such as a database or cluster and create network partitions, then observe the behavior of the nodes. For example in a leader election system, you could partition the leader away from the other nodes and ensure that the leader steps down and that another node emerges as leader.

Blockade features:

- A flexible YAML format to describe the containers in your application
- Support for dependencies between containers, using named links
- A CLI tool for managing and querying the status of your blockade
- Creation of arbitrary partitions between containers
- Giving a container a flaky network connection to others (drop packets)
- Giving a container a slow network connection to others (latency)
- While under partition or network failure control, containers can freely communicate with the host system – so you can still grab logs and monitor the application.

Blockade is written and maintained by the Dell Cloud Manager (formerly Enstratius) team and is used internally to test the behaviors of our software. We also release a number of other internal components as open source, most notably Dasein Cloud.

Blockade is inspired by the excellent Jepsen article series.

Get started with the *Blockade Guide*!

CHAPTER 1

Reference Documentation

1.1 Requirements

Docker must be installed and configured to use the LXC driver. The `native` libcontainer driver is not yet supported by Blockade because it does not expose the necessary network parameters.

Configure the LXC driver by adding `-e lxc` to your Docker daemon options. On Ubuntu, this is done by adding the following to `/etc/default/docker`:

```
DOCKER_OPTS="-e lxc ${DOCKER_OPTS}"
```

Note you will also need to have the LXC command-line utilities installed. On Ubuntu, this is done by installing `lxc` package.

1.2 Installing

Blockade can be installed via `pip` or `easy_install`:

```
$ pip install blockade
```

Because Blockade directly executes `iptables` and `tc` commands, it must be installed on a Linux system or VM and run as root.

1.2.1 OSX

If you are using OSX, Blockade and Docker cannot yet be truly run natively. Use the included `Vagrantfile` or another approach to get Docker and Blockade installed into a Linux VM. If you have Vagrant installed, running `vagrant up` from the Blockade checkout directory should get you started. Note that this may take a while, to download needed VMs and Docker containers.

1.3 Blockade Guide

This guide walks you through a simple example that highlights the power of Blockade. We will start a fake "application" consisting of three Docker containers. The first runs a simple `sleep` command. The other two containers ping the first. With this simple structure, we can easily see what happens when we introduce partitions and network failures between the containers.

1.3.1 Check your Blockade install

This guide assumes you have functional installation of Blockade and Docker, and can run as root (or via sudo). To check, run the following commands:

```
# check docker
$ sudo docker info

# check blockade
$ sudo blockade -h
```

If you get an error from either command, you'll need to fix this before proceeding. See the Docker installation docs and *Requirements*.

1.3.2 Set up your Blockade config

Now create a new directory and in it create a `blockade.yml` file with these contents:

```
containers:
  c1:
    image: ubuntu
    command: /bin/sleep 300000
    ports: [10000]

  c2:
    image: ubuntu
    command: sh -c "ping $C1_PORT_10000_TCP_ADDR"
    links: ["c1"]

  c3:
    image: ubuntu
    command: sh -c "ping $C1_PORT_10000_TCP_ADDR"
    links: ["c1"]
```

This configuration specifies the three containers we described above. Note that we rely on Docker named links which require at least one open port. Hence our sleeping `c1` container has a fake port 10000 open. The `ubuntu` image must exist in your Docker installation. You can download it using the docker pull command `sudo docker pull ubuntu`.

1.3.3 Start the Blockade

Now use the `blockade up` command to stand up our containers:

```
$ sudo blockade up

NODE            CONTAINER ID    STATUS  IP              NETWORK    PARTITION
c1              b9794aaeed42    UP      172.17.0.2      NORMAL
c2              875885f54593    UP      172.17.0.4      NORMAL
c3              9b7227b42466    UP      172.17.0.3      NORMAL
```

You should see output like above. Note that you get the local IP address and Docker container ID for each container.

Now let's take a look at the output of `c2`, which is pinging `c1`. We'll use the `blockade logs` command, but pipe it through tail so we just get the last several lines:

```
$ sudo blockade logs c2 | tail
64 bytes from 172.17.0.2: icmp_req=59 ttl=64 time=0.067 ms
64 bytes from 172.17.0.2: icmp_req=60 ttl=64 time=0.077 ms
64 bytes from 172.17.0.2: icmp_req=61 ttl=64 time=0.077 ms
64 bytes from 172.17.0.2: icmp_req=62 ttl=64 time=0.073 ms
64 bytes from 172.17.0.2: icmp_req=63 ttl=64 time=0.076 ms
64 bytes from 172.17.0.2: icmp_req=64 ttl=64 time=0.070 ms
64 bytes from 172.17.0.2: icmp_req=65 ttl=64 time=0.078 ms
64 bytes from 172.17.0.2: icmp_req=66 ttl=64 time=0.073 ms
64 bytes from 172.17.0.2: icmp_req=67 ttl=64 time=0.109 ms
```

The `blockade logs` command is the same as the `docker logs` command, it grabs any stderr and or stdout output from the container.

1.3.4 Mess with the network

Now let's try a couple network filters. We'll make the network to `c2` be slow and the network to `c3` be flaky.

```
$ sudo blockade slow c2

$ sudo blockade flaky c3

$ sudo blockade status
NODE            CONTAINER ID    STATUS  IP              NETWORK    PARTITION
c1              b9794aaeed42    UP      172.17.0.2      NORMAL
c2              875885f54593    UP      172.17.0.4      SLOW
c3              9b7227b42466    UP      172.17.0.3      FLAKY
```

Now look at the logs for `c2` and `c3` again:

```
$ sudo blockade logs c2 | tail
64 bytes from 172.17.0.2: icmp_req=358 ttl=64 time=126 ms
64 bytes from 172.17.0.2: icmp_req=359 ttl=64 time=0.077 ms
64 bytes from 172.17.0.2: icmp_req=360 ttl=64 time=64.5 ms
64 bytes from 172.17.0.2: icmp_req=361 ttl=64 time=265 ms
64 bytes from 172.17.0.2: icmp_req=362 ttl=64 time=158 ms
64 bytes from 172.17.0.2: icmp_req=363 ttl=64 time=64.8 ms
64 bytes from 172.17.0.2: icmp_req=364 ttl=64 time=3.47 ms
64 bytes from 172.17.0.2: icmp_req=365 ttl=64 time=90.2 ms
64 bytes from 172.17.0.2: icmp_req=366 ttl=64 time=0.067 ms

$ sudo blockade logs c3 | tail
64 bytes from 172.17.0.2: icmp_req=415 ttl=64 time=0.075 ms
64 bytes from 172.17.0.2: icmp_req=416 ttl=64 time=0.079 ms
64 bytes from 172.17.0.2: icmp_req=419 ttl=64 time=0.063 ms
64 bytes from 172.17.0.2: icmp_req=420 ttl=64 time=0.065 ms
64 bytes from 172.17.0.2: icmp_req=421 ttl=64 time=0.063 ms
64 bytes from 172.17.0.2: icmp_req=425 ttl=64 time=0.062 ms
64 bytes from 172.17.0.2: icmp_req=426 ttl=64 time=0.079 ms
64 bytes from 172.17.0.2: icmp_req=427 ttl=64 time=0.056 ms
64 bytes from 172.17.0.2: icmp_req=428 ttl=64 time=0.066 ms
```

Note how the time value of the `c2` pings is erratic, while `c3` is missing many packets (look at the `icmp_req` value – it should be sequential).

Now let's use `blockade fast` to fix the network:

```
$ sudo blockade fast --all

$ sudo blockade status
NODE            CONTAINER ID    STATUS  IP              NETWORK    PARTITION
c1              6367a903f093    UP      172.17.0.2      NORMAL
c2              35efaf92bba0    UP      172.17.0.4      NORMAL
c3              e8ed611a38de    UP      172.17.0.3      NORMAL
```

1.3.5 Partition the network

Blockade can also create partitions between containers. This is valuable for testing split-brain behaviors. To demonstrate, let's partition node `c2` off from the other two containers. It will no longer be able to ping `c1`, but `c3` will continue unhindered.

Partitions are specified as groups of comma-separated container names:

```
$ sudo blockade partition c1,c3 c2

$ sudo blockade status
NODE            CONTAINER ID    STATUS  IP              NETWORK    PARTITION
c1              6367a903f093    UP      172.17.0.2      NORMAL     1
c2              35efaf92bba0    UP      172.17.0.4      NORMAL     2
c3              e8ed611a38de    UP      172.17.0.3      NORMAL     1
```

Note the partition column: `c1` and `c3` are in partition #1 while `c2` is in partition #2.

You can now use `blockade logs` to check the output of `c2` and `c3` and see the partition in effect.

Restore the network with the `join` command:

```
$ sudo blockade join
$ sudo blockade status
NODE            CONTAINER ID    STATUS  IP              NETWORK    PARTITION
c1              6367a903f093    UP      172.17.0.2      NORMAL
c2              35efaf92bba0    UP      172.17.0.4      NORMAL
c3              e8ed611a38de    UP      172.17.0.3      NORMAL
```

1.3.6 Tear down the Blockade

Once finished, kill the containers and restore the network with the `destroy` command:

```
$ sudo blockade destroy
```

1.3.7 Next steps

Next, check out the reference details in *Configuration* and *Commands*.

1.4 Configuration

The blockade configuration file is conventionally named `blockade.yaml` and is used to describe the containers in your application. Here is an example:

```
containers:
  c1:
    image: my_docker_image
    command: /bin/myapp
    volumes: {"/opt/myapp": "/opt/myapp_host"}
    expose: [80]
    ports: {8080: 80}
    environment: {"IS_MASTER": 1}

  c2:
    image: my_docker_image
    command: /bin/myapp
    volumes: ["/data"]
    links: {c1: master}

  c3:
    image: my_docker_image
    command: /bin/myapp
    links: {c1: master}

network:
  flaky: 30%
  slow: 75ms 100ms distribution normal
```

The format is YAML and there are two important sections: `containers` and `network`.

1.4.1 Containers

Containers are described as a map with the key as the Blockade container name (`c1`, `c2`, `c3` in the example above). This key is used for commands to manipulate the Blockade and is also used as the hostname of the container.

Each entry in the `containers` section is a single Docker container in the Blockade. Each container parameter controls how the container is launched. Most are simply pass-throughs to Docker. Many valuable details can be found in the Docker run command documentation.

1.4.2 `image`

`image` is required and specifies the Docker image name to use for the container. The image must exist in your Docker installation.

1.4.3 `command`

`command` is optional and specifies the command to run within the container. If not specified, a default command must be part of the image you are using. You may include environment variables in this command, but to do so you must typically wrap the command in a shell, like `sh -c "/bin/myapp $MYENV"`.

1.4.4 `volumes`

`volumes` is optional and specifies the volumes to mount in the container, *from the host*. Volumes can be specified as either a map or a list. In map form, the key is the path *on the host* to expose and the value is the mountpoint *within the container*. In list form, the host path and container mountpoint are assumed to be the same. See the Docker volumes documentation for details about how this works.

1.4.5 `expose`

`expose` is optional and specifies ports to expose from the container. Ports must be exposed in order to use the Docker named links feature.

1.4.6 `links`

`links` is optional and specifies links from one container to another. A dependent container will be given environment variables with the parent container's IP address and port information. See named links documentation for details.

1.4.7 `ports`

`ports` is optional and specifies ports published to the host machine.

1.4.8 Network

The `network` configuration block controls the settings used for network filter commands like `slow` and `flaky`. If unspecified, defaults will be used. There are two parameters:

1.4.9 `slow`

`slow` controls the amount and distribution of delay for network packets when a container is in the Blockade slow state. It is specified as an expression understood by the tc netem traffic control `delay` facility. See the man page for details, but the pattern is:

```
TIME [ JITTER [ CORRELATION ] ]
    [ distribution { uniform | normal | pareto |  paretonormal } ]
```

`TIME` and `JITTER` are expressed in milliseconds while `CORRELATION` is a percentage.

1.4.10 `flaky`

`flaky` controls the lossiness of network packets when a contrainer is in the Blockade flaky state. It is specified as an expression understood by the tc netem traffic control `loss` facility. See the man page for details, but the simplified pattern is:

```
random PERCENT [ CORRELATION ]
```

`PERCENT` and `CORRELATION` are both expressed as percentages.

1.5 Commands

The Blockade CLI is built to make it easy to manually manage your containers, and is also easy to wrap in scripts as needed. All commands that produce output support a `--json` flag to output in JSON instead of plain text.

For the most up to date and detailed command help, use the built-in CLI help system (`blockade --help`).

1.5.1 `up`

```
usage: blockade up [--json]

Start the containers and link them together

  --json      Output in JSON format
```

1.5.2 `destroy`

```
usage: blockade destroy

Destroy all containers and restore networks
```

1.5.3 `status`

```
usage: blockade status [--json]

Print status of containers and networks

optional arguments:
  --json      Output in JSON format
```

1.5.4 `logs`

```
usage: blockade logs CONTAINER

Fetch the logs of a container

  CONTAINER    Container to fetch logs for
```

1.5.5 `flaky`

```
usage: blockade flaky [--all] [CONTAINER [CONTAINER ...]]

Make the network flaky for some or all containers

  CONTAINER   Container to select

  --all       Select all containers
```

1.5.6 `slow`

```
usage: blockade slow [--all] [CONTAINER [CONTAINER ...]]

Make the network slow for some or all containers

  CONTAINER   Container to select

  --all       Select all containers
```

1.5.7 `fast`

```
usage: blockade fast [--all] [CONTAINER [CONTAINER ...]]

Restore network speed and reliability for some or all containers

  CONTAINER   Container to select

  --all       Select all containers
```

1.5.8 `partition`

```
usage: blockade partition PARTITION [PARTITION ...]

Partition the network between containers

    Replaces any existing partitions outright. Any containers NOT specified
    in arguments will be globbed into a single implicit partition. For
    example if you have three containers: c1, c2, and c3 and you run:

        blockade partition c1

    The result will be a partition with just c1 and another partition with
    c2 and c3.


  PARTITION   Comma-separated partition
```

1.5.9 `join`

```
usage: blockade join

Restore full networking between containers
```

1.6 Changelog

1.6.1 0.1.2 (2015-1-28)

- #6: Change *ports* config keyword to match docker usage. It now publishes a container port to the host. The *expose* config keyword now offers the previous behavior of *ports*: it makes a port available from the container, for linking to other containers. Thanks to Simon Bahuchet for the contribution.
- #9: Fix logs command for Python 3.
- Updated dependencies.

1.6.2 0.1.1 (2014-02-12)

- Support for Python 2.6 and Python 3.x

1.6.3 0.1.0 (2014-02-11)

- Initial release of Blockade!

CHAPTER 2

Development and Support

Blockade is available on github. Bug reports should be reported as issues there.

CHAPTER 3

License

Blockade is offered under the Apache License 2.0.