Appendix A

A closer look at Serf

Website: https://www.serf.io
Chat: Gitter
Mailing list: Google Groups

Serf is a decentralized solution for service discovery and orchestration, written in GO, and available as a an open source project at https://github.com/hashicorp/Serf. Serf is built on top of well-cited academic research where the pros, cons, failure scenarios, scalability, etc. are all well defined

Serf runs on Linux, Mac OS X, and Windows. It is lightweight, highly available, and fault tolerant. Serf is completely masterless with no single point of failure.

Its full documentation is viewable on the Serf website: https://www.Serf.io/docs

Serf uses an efficient gossip protocol to solve three major problems:

- **Membership**: Serf maintains cluster membership lists and is able to execute custom handler scripts when that membership changes. For example, Serf can maintain the list of web servers for a load balancer and notify that load balancer whenever a node comes online or goes offline.
- **Failure detection and recovery**: Serf automatically detects failed nodes within seconds, notifies the rest of the cluster, and executes handler scripts allowing you to handle these events. Serf will attempt to recover failed nodes by reconnecting to them periodically.
- **Custom event propagation**: Serf can broadcast custom events and queries to the cluster. These can be used to trigger deploys, propagate configuration, etc. Events are simply fire-and-forget broadcast, and Serf makes a best effort to deliver messages in the face of offline nodes or network partitions. Queries provide a simple realtime request/response mechanism.

Here are some use-cases for Serf, among others:

- Discovering web servers and automatically adding them to a load balancer
- Organizing many memcached or redis nodes into a cluster, perhaps with something like twemproxy or maybe just configuring an application with the address of all the nodes
- Triggering web deployments using the event system built on top of Serf
- Propagating changes to configuration to relevant nodes.
- Updating DNS records to reflect cluster changes as they occur.
- ...

Use-cases are further detailed in the following section.

Use-cases

Web Servers and Load Balancers

Using Serf, it is trivial to create a Serf cluster consisting of web servers and load balancers. The load balancers can listen for membership changes and when a web server comes online or goes offline, they can update their node list.

Clustering Memcached or Redis

Servers such as Memcached or Redis can be easily clustered by creating a Serf cluster for these nodes. When membership changes, you can update something like twemproxy or your own application's list of available servers.

Triggering Deploys

Serf can send custom events to a Serf cluster. If you cluster your web applications into a single cluster, you can use Serf's event system to trigger things such as deploys. Just call Serf event deploy and have event handlers installed on all the nodes and the entire cluster will receive this message within seconds and begin deploying.

Updating DNS Records

Keeping your internal DNS servers updated can be a finicky process. By using Serf, the DNS server can know within seconds when nodes join, leave, or fail, and can update records appropriately. No more stale DNS records, or waiting for a Chef or Puppet run to clear out the records within X minutes. With Serf, the records can be updated nearly instantly.

Simple Observability

Serf provider queries which can be used as a simple request/response mechanism. It can be used very simply to provide cluster and application observability. Calling Serf query load can trigger all the nodes to call uptime and send their load averages to the query initiator, making it easy to check on the cluster health.

A Building Block for Service Discovery

One of the most difficult parts of service discovery is simply knowing what nodes are online, at what addresses, and for what purpose. An effective service discovery layer can easily be built on top of Serf's seamless membership system. Serf handles all the problems of keeping an up-to-date node list along with some information about those nodes. The service discovery layer then only needs to answer basic questions above that.

Quick Start

1) Download a pre-built Serf binary from https://www.serf.io/downloads.html for your OS, unzip the file and move `` `serf' `` to `/usr/bin/`
You can also compile Serf yourself: see
https://github.com/hashicorp/Serf/blob/master/README.md#developing-Serf)

2) Start a couple of Serf agents.
Agents run until they're told to quit. They handle the maintenance tasks communication of Serf.

In a real-world Serf setup, each node in your system will run one or more Serf agents. A node may run multiple agents if you're running multiple clusters. e.g. web servers vs. mem-cached servers, possibly with different gossip response time constraints.

Start each Serf agent in a separate terminal session so that we can see the output of each.
- Start the first agent:

```
$ serf agent -node=foo -bind=127.0.0.1:5000 -rpc-addr=127.0.0.1:7373
```

- Start the second agent in another terminal session (while the first is still running):

```
$ serf agent -node=bar -bind=127.0.0.1:5001 -rpc-addr=127.0.0.1:7374
```

At this point two Serf agents are running independently but are still unaware of each other.
Note that we specify the RPC[1] option, `-rpc-addr=127.0.0.1:`*port-number*, so local calls can be distinguished from remote calls and remote calls can be distinguished between themselves.

3) Tell the first agent to join an existing cluster (the second agent).
When starting a Serf agent, you must join an existing cluster by specifying at least one existing member.
After this, Serf gossips and the remainder of the cluster becomes aware of the join. Run the following commands in a third terminal session.

```
$ serf join 127.0.0.1:5001
```

If you're watching your terminals, you should see both Serf agents become aware of the join. You can prove it by running Serf members to see the members of the Serf cluster:

```
$ Serf members
foo    127.0.0.1:5000    alive
bar    127.0.0.1:5001    alive
```

At this point, you can ctrl-C or force kill either Serf agent, and they'll update their membership lists appropriately.
- If you ctrl-C a Serf agent, it will gracefully leave by notifying the cluster of its intent to leave.
- If you force kill an agent, another member of the cluster will eventually notice.

It will then notify the cluster of the node failure. This normally happen seconds after the kill cmd was issued.

1 An RPC or *remote procedure call* occurs when a process causes a procedure to execute in another address space (commonly on another computer on a shared network). Location transparency is central to RPCs so calling procedures is largely the same whether it is local or remote.

Appendix B

Security Groups, Limits and Access Control

Security Groups

AWS automatically creates a security group[2] associated to your instances, i.e. a software-defined FW that controls incoming and outgoing traffic for every launched instance, or virtual server in the AWS cloud. The default security group admits incoming SSH traffic from any IP address and allow all outbound traffic.

Once on the EC2-Dashboard, check the various *Limits* imposed by AWS. They can vary in time and depend on:
- which of the *14 regions* actually hosts your VPS,
- on the nature of the resources utilized (instance type) and
- on the instantaneous computing and I/O loads at any given time.

A *Security Group* acts as a virtual FW for an instance, controlling inbound and outbound traffic. An instance launched in a VPC can be assigned to up to 5 Security Groups, each acting at instance level, not at subnet level. In practice, each instance in a VPC subnet could be assigned to a different set of Security Groups. Configuring Security Groups consists in adding rules for inbound traffic and distinct rules for outbound traffic. By default all traffic is denied: inbound and outbound, except for return traffic in both directions (Security Groups are *stateful*). Rules selectively allow traffic to pass ("allow" rules only). Rules are evaluated globally, meaning that all rules are evaluated before traffic can be allowed or denied.

Limits

• Limits, indicated between parentheses for region "us-west-2", apply to the number of on-demand, spot and reserved instances that can be created within the AWS Educate session; e.g.:

r*unning On-Demand i3.4xlarge instances*	*(2)*
running On-Demand i3.large instances	*(16)*
running On-Demand m4.2xlarge instances	*(20)*
running On-Demand m4.xlarge instances	*(40)*
running On-Demand p2.8xlarge instances	*(0)*

There are currently 58 different types of instances as detailed in AWS' documentation[3]. They are best used selectively for either *general purpose*, *compute optimized*, *memory optimized*, *storage optimized* or *accelerated computing* use-cases.

In addition to the above, a set of *micro instances* present special characteristics and limits well documented in AWS.

• Limits also apply to *Elastic Block Storage* (EBS), with caps on the maximum number of iops, storage and/or throughput for a given region.

• Limits also apply to *Networking*, with caps on the maximum number of:
- *Virtual Private Clusters* (VPCs) (5) per region and elastic IPs (5) per VPC
- subnets (200) and security groups (500) per VPC
- rules (50) per VPC security group
- etc.

• Limits also apply to *Auto-scaling groups*[4] (20) and *Launch Configurations* (100) on a per account basis.
An a*uto-scaling group* contains a collection of EC2 instances that share similar characteristics and are treated as a

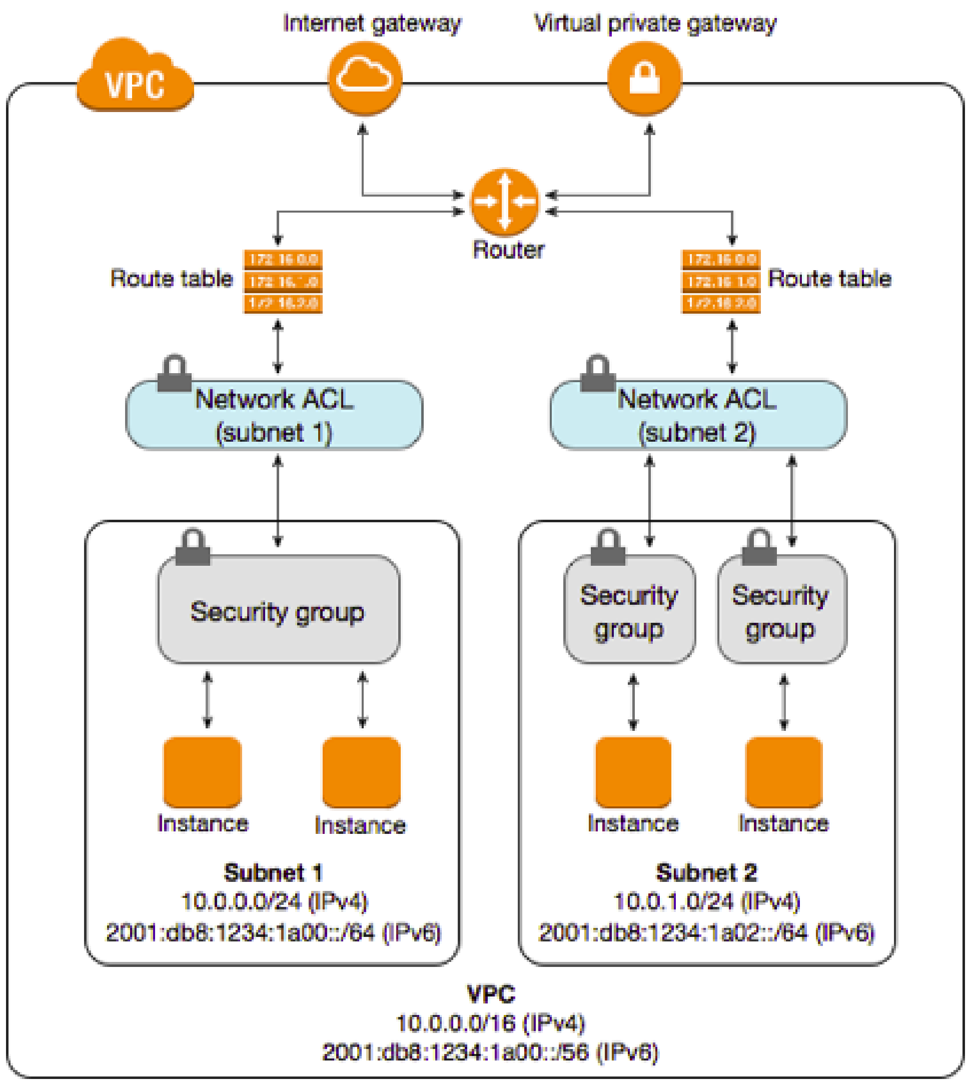
2 An AWS instance may have more than one security groups associated to it and up to 100 rules can be added to each security group. Those rules allow traffic to or from its associated instances. They can be modified for any security group and at any time; the new rules are automatically applied to all instances associated with the security group. When deciding whether to allow traffic to reach an instance, evaluate all the rules from all the security groups associated with said instance's network interface. The 2 instance-types, EC2-classic and EC2-VPC, are backed by distinct security groups. There can not be more than 500 security groups defined per region for any given AWS account and instance-type.

3 See https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html

4 See https://docs.aws.amazon.com/autoscaling/latest/userguide/AutoScalingGroup.html

logical grouping for the purpose of instance scaling and management. Auto-scaling policies permit increasing or decreasing the number of running EC2 instances automatically to meet changing conditions

Network ACLs with their own sets of “deny” or “allow” rules act at subnet level and can co-exist with security groups. They constitute an additional layer of security. In contrast to Security Groups however, they are stateless and return traffic must be explicitley allowed. Rules are processed sequentially following the order in which they are drafted, when deciding whether or not to allow/deny traffic to all instances in a corresponding subnet.



Appendix C
Install and run Docker

Docker is a well documented FOSS project that automates the deployment of applications inside software containers. A Docker container (DC) is a set of restricted namespaces (a process namespace, a filesystem namespace, etc) where processes can run in isolation. Docker implements at its core a separation between application (and any dependencies needed to run it) from the operating system itself. To make this possible Docker uses *containers* as running instances of *images*. A Docker image is basically a stand-alone, executable template for a filesystem, including code, RTE, libraries, and config files. When you run a Docker image, an instance of this filesystem is made live and runs on your system inside a DC. By default this container cannot modify with the original image itself or the filesystem of the host where Docker is running. It is a self-contained environment.

In this section, you will learn how to:
- install Docker and to run it, pulling images,
- make your own Docker images and push them to DockerHub, the Docker registry or repository (repo).

In a series of online tutorials, Docker offers a more in-depth presentation for beginners to get started[5].

1 -- *Installation of Docker*

Installation on your platform is not mandatory for the application ***pixColor*** to run well. However it will simplify your configuration tasks and reduces the amount of scripting work, you will need to do to get everything rolling. Following this section will also equip you with useful information and hands-on experience about how to put Docker to use.

At the time of writing the latest Docker version was v17.03. Before installing the Docker Community Edition (*Docker CE*) , make sure you comply with prerequisites. When those prerequisites are satisfied, do from terminal:

```
$ alias sag='sudo apt-get'
```

In case older versions of Docker are present, remove them:

```
$ sag remove docker docker-engine
```

Update the list of available packages:

```
$ sag update; $ sag upgrade -y
```

Install extra packages to allow Docker to use the `'aufs'` storage drivers:

```
$ sag install linux-image-extra-$(uname -r) linux-image-extra-virtual
```

Set up Docker's repositories and install (here, the amd64 compatible version) from them, for ease of installation and future upgrades:

```
$ sag install apt-transport-https ca-certificates curl software-properties-common
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
$ sudo apt-key fingerprint 0EBFCD88  # Verify the official GPG key fingerprint
  pub   4096R/0EBFCD88 2017-02-22
  Key fingerprint = 9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
  uid                  Docker Release (CE deb) <docker@docker.com>
  sub   4096R/F273FCD8 2017-02-22
$ sudo add-apt-repository "deb [arch=amd64] \
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

Update the list of available packages:

```
$ sag update; $ sag upgrade -y
```

Install the latest version of Docker:

```
$ sag install docker-ce
```

5 https://docs.docker.com/get-started/

Or, install a specific version of Docker, if it suits your needs better:
```
$ sag install docker-ce=<VERSION>
```

To to avoid having to type `sudo` whenever you run the `docker` command, add your username to the docker group:
```
$ sudo usermod -aG docker $(whoami)
```
To add a user other than you to the docker group, declare that <username> explicitly:
```
$ sudo usermod -aG docker <username>
```

At this point, it is advisable to perform a full log out from session followed by a log in.

In the following sub-section, we assume that the user is part of the `docker` group. If not, prepend all commands with `sudo.`

Finally, to uninstall Docker:
```
$ sudo apt-get remove --purge docker-ce
$ sudo rm -rf /var/lib/docker
```

2 – *Getting started with Docker*
Understanding the structure of the command (cmd) line in Docker is paramount.

First let us verify that Docker is (i) running) and (ii) installed correctly. First check that the daemon runs; second run the `hello-world` DC image.
```
$ /etc/init.d/docker status
  docker start/running, process 17930
$ docker run hello-world
  ...[output here]...
```

Using `docker` consists in passing options to it, then a command followed by arguments. The syntax takes this form:
```
$ docker [opts] [cmd] [args]
```

To view all available sub-cmds, type:
```
$ docker
```

To view the switches available to a specific sub-cmd, type:
```
$ docker <docker-sub-cmd> --help
```
For instance, to learn about how to use `build`, type:
```
$ docker build --help
  Usage:    docker build [OPTIONS] PATH | URL | -
  Build an image from a Dockerfile
  Options: ...
```

To view system-wide information about Docker, use:
```
$ docker info
```

3 – *Working with Docker* starts with understanding that Docker containers are run from Docker images. The rest of this sub-section is essentially a rewrite of Digital Ocean's how-to[6] on the subject.

By default, Docker pulls images from its repo, DockerHub, a Docker registry managed by Docker. Anybody can build and host their Docker images on DockerHub. Most applications and Linux distros you'll need to run with Docker containers have images hosted on DockerHub.

You can search for images available on DockerHub by using the `docker` cmd with the `search` sub-cmd. For example, to search for the Ubuntu image, type:
```
$ docker search ubuntu
```

6 https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-16-04

```
      ...[output here]...
```

The script will crawl DockerHub and return a listing of all images whose name match the search string `'ubuntu'`.

In the OFFICIAL column of the output, "OK" indicates an image built and supported by the company behind the project. Identify the image you need, and download it to your computer using the `pull` sub-cmd:

```
    $ docker pull ubuntu
```

After downloading an image of interest, run a container using that image using the `run` sub-cmd. If an image has not been downloaded when `docker` is executed with the `run` sub-cmd, the Docker client will first download the image, then run a container using it:

```
    $ docker run ubuntu
```

To see the images that have been downloaded to your computer, type:

```
    $ docker images
```

Images that you use to run DCs can be modified and used to generate new images, which may then be *pushed* to DockerHub or other Docker registries.

4 – *Running a Docker container (DC)* is as easy as what you already did at the beginning of subsection 1.2.2.

```
    $ docker run hello-world
      ...[output here]...
```

In the above, a DC ran (very briefly) and exited immediately after emitting a message.

Containers can be run in interactive (attached) or non-interactive (detached) mode. To exemplify this, we now run an attached DC using an image of the latest Ubuntu operating system. The combination of the **-i** and **-t** switches gives **interactive shell** access, allocates a pseudo **tty** and keeps **stdin** open even if the container[7] is not attached (**-a**):

```
    $ docker run -it --rm ubuntu
      Unable to find image 'ubuntu:latest' locally
      latest: Pulling from library/ubuntu
      b6f892c0043b: Downloading  2.85 MB/46.89 MB ...
      55010f332b04: Download complete
      ...[more output here]...
    root@d9b100f2f636:/# uname -a
      Linux 2a3747ea8be5 4.4.0-75-generic #96~14.04.1-Ubuntu SMP Thu Apr 20 11:06:30
      UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
    root@d9b100f2f636:/#
```

The command prompt changed to reflect the fact that you're now working inside the container. Note that the **DC id** (in this case `'d9b100f2f636'`) is now included in the new command prompt.

You may now run any command inside the DC provided it is recognized by the resources included in the corresponding Docker image. For example, let's update the package database inside the container and install `nodeJS`. There is no need to prefix any command with `sudo`, because you are operating inside the container with root privileges:

```
    root@d9b100f2f636:/# apt-get update
    root@d9b100f2f636:/# apt-get install -y nodejs
```

To stop an active container from within its environment (i.e. from inside the DC), just exit it:

```
    $ root@d9b100f2f636:/# exit          # to stop an active container in interactive mode
```

To re-start a stopped (exited) container identified by its id, attach tty and stdin to it, in interactive mode:

```
    $ docker start -a -i container-id
or  $ docker start -a -i "$(docker ps -q -l)"     # to restart the last exited container RTE
or  $ docker start -a -i "$(docker ps -aq --filter "status=exited")"   # to re-start all exited container
```

7 https://docs.docker.com/articles/basics/#running-an-interactive-shell

As already mentioned, one may also run container images in detached (non-interactive) mode, by including the **-d** switch. This allows you to return to your underlying session shell, while the container keeps running:

```
$ docker run -d -it ubuntu
  746fd4aceacffe3a60cc835ce95c97377884d2a96f23ad519c2e40de6fdc395f
$
```

To re-attach the DC, do:

```
$ docker ps [-a|l]          # to list containers: -a = all (active and inactive), -l = latest created
CONTAINER ID   IMAGE   COMMAND     CREATED     STATUS    PORTS   NAMES
746fd4aceacf   ubuntu  "/bin/bash" 19sec ago   Up 18sec          heuristic_archimedes

$ docker attach --sig-proxy=false 746fd4aceacf      # see Docker docs[8] for more info
root@746fd4aceacf:/#
```

To detach the DC while in its RTE, simply issue CTRL+p (^p) CTRL+q (^q) to recover your usual host shell-prompt neither killing the container nor any process running inside it.

To stop an active DC running non-interactively (detached mode) from the normal shell prompt, simply issue:

```
      $ docker stop container-id              # container-id = 746fd4aceacf
or    $ docker stop container-name         # container-name = heuristic_archimedes
      $ docker kill $(docker ps -q)        # stop all running containers at once
      $ docker rm $(docker ps -a -q)   # remove all containers (running and stopped) at once
```

5 – ***Creating a Docker image*** consists in committing changes in a Docker container to a file. It is recommended, if you mean to preserve for future re-use any configuration work made in the DC's virtual environment.

After installing `nodeJS` inside the previous Ubuntu DC, you now have a running container, whose composition is different from that of the image you used to create it. To save the state of the container as a new image, first exit from it:

```
$ exit
```

Then commit the changes to a new Docker image instance using the following cmd:

```
$ docker commit -m "commit_message" -a "author" container-id repo/new_image_name
```

Specify:	**-m**	"commit message" that helps you and others know what changes you made,
	-a	"author".
	container-id	as noted earlier when starting the interactive docker session.

Unless you created additional repositories on Docker Hub, the repository is usually your DockerHub username:

```
$ docker commit -m "added node.js" -a "skb" d9b100f2f636 skb/ubuntu-nodejs
  sha256:a6f9c14d6ce51033b7798011b5e442848ed287b51d7e93332cb2716f7f58710d
```

When you *commit* an image, the new image is saved ***locally***. Later in this tutorial, you will push an image to a Docker registry like DockerHub, so it becomes publicly accessible.

After the commit operation has completed, listing the Docker images now on your computer should show the new image, as well as the old one that it was derived from:

```
$ docker images
  REPOSITORY         TAG        IMAGE ID        CREATED          SIZE
  skb/ubuntu-nodejs  latest     a6f9c14d6ce5    2 minutes ago    206 MB
  ubuntu             latest     ebcd9d4fca80    24 hours ago     118 MB
  hello-world        latest     48b5124b2768    4 months ago     1.84 kB
```

8 https://docs.docker.com/engine/reference/commandline/attach/#attach-to-and-detach-from-a-running-container

In the above example, `ubuntu-nodejs` is a new local image, derived from the existing ubuntu image from DockerHub. The size difference reflects the change as `nodeJS` got installed. To run a DC using Ubuntu with `nodeJS` pre-installed, use the new image:

```
$ docker run -it --name foobar_dc skb/ubuntu-nodejs
root@7d6db03013ba:/#
```

Note that the `container-id` above is different from before, as you now run the container image under a new parent-PID. Note too that we added the new option **`'--name'`** which allows you to choose a name for each DC instance you may want to create (here "foobar_dc"). That name may be used at your convenience to detach and re-attach DCs, commit changes, stop a DC, as you would for these and more operations, when using the DC id.

To rename a DC, issue:

```
$ docker rename old_name new_name
```

To remove an image:

```
$ docker rmi image_id
```

To remove all images at once:

```
$ docker rmi $(docker images -q)
```

6 – *Building images from Dockerfile* is another convenient way to automate the build of an image and to include a context in it.

• Docker reads instructions from the `Dockerfile`, a text document which contains the commands a user could call on cli to assemble an image. Users can read command-line instructions from a Dockerfile located anywhere on the host, with:

```
$  docker build  -f /path/to/Dockerfile -t yr-dockerhub-repo/yr-image-name:yr-tag .
```

• The context is the set of files at the build location, typically specified by `.` to signify "present directory and recursively parsed sub-tree". It can however be any directory and sub-tree specified by its path or a URL.

When no fully qualified path for `Dockerfile` is provided, `docker` will revert to the default Dockerfile location, which is the root of the context.

The content of the Dockerfile is parsed and run by the Docker daemon, not by the host's shell. It conforms to a set of specific syntactic rules and uses case-insensitive reserved words. The complete documentation on this subject is available at https://docs.docker.com. A Dockerfile is similar in concept to the recipes and manifests found in infrastructure automation (IA) tools like Chef or Puppet.

7 -- *Pushing DC images to a Docker repo* is the next logical step, after creating a new container image.

To push an image to DockerHub or any other Docker image repo, you must have an account on that repo.

To learn how to create your own private Docker repo[9], see How to set up a private docker registry on Ubuntu. In the remainder of this guide, we will only concern ourselves with pushing Docker images to DockerHub.

First create an account on DockerHub by registering on the Docker's website. In order to push a container image, you must have logged into DockerHub from your local terminal session. You'll be prompted to authenticate:

```
$ docker login -u docker-registry-username
  Enter password:_
```

9 https://www.digitalocean.com/community/tutorials/how-to-set-up-a-private-docker-registry-on-ubuntu-14-04

Then you may push your own image using:
```
        $ docker push docker-registry-username/docker-image-name
          The push refers to a repository [docker.io/skb/ubuntu-nodejs]
          ...[output]...
```

After pushing an image to your GitHub repo, it should be listed on your account's dashboard.

8 – *Docker networking* is highly configurable. It involves NAT IPTables, port mapping between the contenerized apps and the host network space, virtual interfaces and default networks created upon installation of the Docker engine in the host environement.

8.1 – Docker Networks

Docker comes with 4 built-in network drivers: (i) **bridge**, (ii) **host**, (iii) **macvlan** and (iv) **overlay**.

- "bridge" is the default network in which all containers are launched.
- "host" lets containers access the host's network stack (i.e. using the exact same url:port network space)
- "macvlan" gives DCs a direct access the host interface or sub-interface and permits trunking.
- "overlay" allows networks to be built across multiple hosts running Docker (usually in the case of a *Docker swarm cluster*). In this configuration DCs have their own subnet and network addresses and can communicate directly with one another, even when running on physically different hosts.

For this work, the 2 networks of interest are *bridge* and *overlay*. To keep our configuration as simple as possible, we will focus on the bridge-network identified "`bridge`" created automatically when installing Docker. As described on https://securitynik.blogspot.co.uk/, "the Linux bridge interfaces are similar to switches in their function in that they connect different interfaces to the same subnet, and forward traffic based on MAC addresses. As we shall see below, each container connected to a bridge network will have its own virtual interface created on the docker host, and the docker engine will connect all containers in the same network to the same bridge interface, which will allow them to communicate with each other."

Even in the absence of a running container, query the default network configuration built into Docker as follows:

```
        $ docker network ls
          NETWORK ID          NAME                DRIVER              SCOPE
          a2e715a00b4b        bridge              bridge              local
          939bff610adb        host                host                local
          2874386a03ef        none                null                local
```

The "`none`" network permits adding a container to a container-specific network stack, so that container lacks a network interface.

Beside its default network configuration, Docker allows you to create user-defined networks in addition to the default bridge network. In any case when running docker, use the "-- `network`" flag to specify which network you would like your DC to connect to. By default DCs connect to the bridge network, whose interface is known as "`docker0`".

```
        $ ifconfig   # run from outside a container, i.e. from the host's environment
        docker0  Link encap:Ethernet  HWaddr 02:42:95:f7:81:b6
                 inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
                 inet6 addr: fe80::42:95ff:fef7:81b6/64 Scope:Link
                 UP BROADCAST MULTICAST  MTU:1500  Metric:1
                 RX packets:14 errors:0 dropped:0 overruns:0 frame:0
                 TX packets:37 errors:0 dropped:0 overruns:0 carrier:0
                 collisions:0 txqueuelen:0
                 RX bytes:992 (992.0 B)  TX bytes:17411 (17.4 KB)
                 […]
```

The "`docker network inspect network-name`" cmd returns information about the named network:
```
        $ docker network inspect bridge
```

will reveal the bridge network gateway IP address, its name (default is "`docker0`"), its subnet (e.g. 172.17.0.0/16), its gateway (e.g. 172.17.0.1) and various option settings pertaining to network traffic, such as ICMP[10], masquerading for NAT and more.

> *"com.docker.network.bridge.default_bridge": "true",*
> *"com.docker.network.bridge.enable_icc": "true",*
> *"com.docker.network.bridge.enable_ip_masquerade": "true",*
> *"com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",*
> *"com.docker.network.bridge.name": "docker0",*
> *"com.docker.network.driver.mtu": "1500"*

8.2 – Docker IPTables[11]

As a rule you will only need to configure IPTables in *production* environments or when the bridge network for containers is manually set up at container deployment time. Otherwise you will mostly rely on Docker defaults. In such a case, this sub-section will only serve informative purposes. In any case this sub-section should not be considered an attempt to review IPTables. Other online resources[12] are better suited for this purpose.

Docker uses IPTables to control communication to and from the interfaces and networks it creates. IPTables consist of different tables, two of which are of particular interest to us: `filter` and `nat`.

- `filter` is the security rules table used to allow or deny traffic to IP addresses, networks or interfaces,
- `nat` contains the rules responsible for masking IP addresses or ports. Docker uses Network Address Translation (NAT) to allow containers on bridge networks to communicate with destinations outside the docker host (otherwise routes pointing to the container networks would have to be added in the docker host's network, a potential source of vulnerabilities, in particular when running several containers on the same host).

Note that `iptables` settings are lost across system reboots.

IPTables: `filter`

Tables in IPTables consist of different *chains* that correspond to different conditions or stages in processing a packet on the docker host. The `filter` table has 3 chains by default:

- chain: INPUT – to process packets arriving at the host and destined for the same host,
- chain: OUTPUT – for packets originating on the host to an outside destination,
- chain: FORWARD – for packets entering the host but with a destination outside the host.

Each chain consists of a list of rules that dictate some action to be taken on the packet (for example reject or accept the packet) as well as conditions for matching the rule. Rules are processed in sequence up until a match is found, otherwise the default policy of the chain is applied.

It is possible to define custom chains in a table. This is outside the scope of this short tutorial.

To view the currently configured rules and default policies for chains in the `filter` table,

```
$ sudo iptables -t filter -L        # same as 'iptables -L' as the filter table is used by default
```

The output shows that Docker adds two custom chains: DOCKER and DOCKER-ISOLATION, and inserts rules in the FORWARD chain with those 2 new chains as *targets*. The DOCKER-ISOLATION chains contains rules that restrict

10 ICMP or Internet Control Message Protocol is a transport layer control protocol of the internet protocol suite, used by network devices, including routers, to send error messages and operational information indicating, for example, that a requested service is not available or that a host or router cannot be reached. For instance it is involved in verifying whether ping queries are correctly answered by the queried host. The utility `traceroute' also relies on it. (http://www.networksorcery.com/enp/protocol/icmp.htm)

11 Subsection 8.2 is a shortened version of a highly recommended blog post by Nik Alleyne. Its conclusion is quoted in full.

12 https://wiki.archlinux.org/index.php/Iptables#Tables

access between the different container networks. To illustrate that in details, consider the following example where we ran the previous command in verbose mode:

```
        $ sudo iptables -t filter -L -v | grep -e “Chain DOCKER-ISOLATION” -A10
          […]
          Chain DOCKER-ISOLATION (1 references)
  pkts bytes target prot opt in      out              source          destination
    0   0   DROP    all  --  br-e6bc7d6b75f3 docker0  anywhere        anywhere
    0   0   DROP    all  --  docker0 br-e6bc7d6b75f3  anywhere        anywhere
    0   0   DROP    all  --  docker_gwbridge docker0  anywhere        anywhere
    0   0   DROP    all  --  docker0 docker_gwbridge  anywhere        anywhere
    0   0   DROP    all  --  docker_gwbridge br-e6bc7d6b75f3  anywhere       anywhere
    0   0   DROP    all  --  br-e6bc7d6b75f3 docker_gwbridge  anywhere       anywhere
36991 3107K RETURN  all  --  any     any              anywhere        anywhere
```

In the above a number of DROP rules block traffic between various bridge interfaces created by Docker, thus making sure that container networks cannot communicate. At an early development stage (NOT at production or pre-production stages), it is convenient to relax those rules so that all containers in a cluster may talk to one another. In that case the output would simply be:

```
        $ sudo iptables -t filter -L -v | grep -e “Chain DOCKER-ISOLATION” -A3
          Chain DOCKER-ISOLATION (1 references)
  pkts bytes target  prot opt in      out              source          destination
 36991 3107K RETURN  all  --  any     any              anywhere        anywhere
```

<u>icc=false</u>

The cmd `` `docker network create' `` admits several options. One of them is instrumental in denying or permiting inter-container communication on one single network: `com.docker.network.bridge.enable_icc.`
`icc` stands for inter-container communication. Setting this option to *`false`* blocks containers on the same network from communicating with each other. In that case Docker adds a drop rule in the `FORWARD` chain that matches packets which come from the bridge interface associated with the network and are destined for the same interface.

For example, create a new network called “`no-icc-network`” with:

```
        $ docker network create --driver bridge --subnet 192.168.200.0/24 –ip-range   \
        192.168.200.0/24 -o "com.docker.network.bridge.enable_icc"="false" no-icc-network
```

That newly created bridge network uses an interface whose name (`br-8e3f0d353353`) you can discover with:

```
        $ ifconfig | grep 192.168.200 -B1
          br-8e3f0d353353 Link encap:Ethernet  HWaddr 02:42:c4:6b:f1:40
                 inet addr:192.168.200.1  Bcast:0.0.0.0  Mask:255.255.255.0
```

Then check that a new rule was appended to the `FORWARD` chain:

```
        $ sudo iptables -t filter -S FORWARD | grep “br-8e3f0d353353”
          -A FORWARD -i br-8e3f0d353353 -o br-8e3f0d353353 -j DROP
```

thereby dropping all packets simultaneously originating at and destined to the network bridge interface: `br-8e3f0d353353.`

Again for early development purposes, we make sure that, when appropriate:

```
        "com.docker.network.bridge.enable_icc"="true"
```

In addition, although this may not always be necessary, depending on the particulars of a multi-host cluster configuration, inter-container communication is set to ACCEPT when issuing:

```
        $ sudo iptables -P FORWARD ACCEPT
```

This however is not a secure configuration as any compromised Docker container will then gain the ability to also compromise other containers across hosts in the same cluster.

iptables:nat

NAT allows the host to change the IP address or port of a packet. In this instance, it is used to mask the source IP address of packets coming from docker bridge networks (for example hosts in the 172.17.0.0/24 subnet) destined to the outside world, behind the IP address of the docker host.

The `com.docker.network.bridge.enable_ip_masquerade` option controls this. As in the above it can be passed to `'docker network create'` and defaults to "true" when not specified.

You can see the effect of this command in the nat table of iptables:

```
         $ sudo iptables -t nat -L | grep -e POSTROUTING -A3
           Chain POSTROUTING (policy ACCEPT
target      prot  opt source              destination
MASQUERADE  all   --  172.17.0.0/16       anywhere
MASQUERADE  all   --  10.0.3.0/24         !10.0.3.0/24
```

In the above masquerading is effective everywhere from subnet 172.17.0.0/16, but not for local traffic within subnet 10.0.3.0/24.

8.3 – Docker containers' virtual interfaces

Containers connected to the same bridge network (e.g. to the default bridge network with bridge interface "`docker0`") can communicate with each other by their IP address, as revealed by the above cmd's full ouput. Docker does not support automatic service discovery on the default bridge network, so for containers to be able to resolve IP addresses by container name, a user-defined fully specified network should be set up instead of the default one.

Each running container belonging to the same bridge network with bridge network interface "`docker0`" has a virtual interface named "`veth`xxxxxxx", where xxxxxxx is an 7-character alphanumerical chain unique to each container. Virtual interfaces are revealed, e.g. in the presence of 2 running Ubuntu-based containers ("dc1" and "dc2"), by:

```
$ docker pull ubuntu
$ docker run -it --name "dc1" ubuntu
root@a754719db594:/# apt-get install -y ethtool iputils-ping bridge-utils
^p ^q
$ docker run -it --name "dc2" ubuntu
root@976041ec420f :/# apt-get install -y ethtool iputils-ping bridge-utils
^p ^q

$ docker ps -q    # query running containers' ID
  a754719db594
  976041ec420f

$ sudo brctl show docker0
  bridge name     bridge id               STP enabled     interfaces
  docker0         8000.02424488bd75       no              veth2177159
                                                          vethd8e05dd
```

Listening on either one of the two virtual interfaces or on both (e.g. with `tcpdump`) is the same as listening on the bridge network interface. By way of example, imagine pinging the RIPE Network Coordination Center from inside container id `a754719db594`.

```
$ docker exec a754719db594 ping -c 40 80.58.61.254 >& /dev/null &
  [1] 13599

$ sudo tcpdump -i veth2177159 icmp           # listen in from the host, outside the container
  listening on vethc955099, link-type EN10MB (Ethernet), capture size 262144 bytes
  22:55:56.027337 IP 172.17.0.2 > 254.red-80-58-61.staticip.rima-tde.net: ICMP echo
request, id 436, seq 261, length 64
```

```
          22:55:56.030896 IP 254.red-80-58-61.staticip.rima-tde.net > 172.17.0.2: ICMP echo reply,
id 436, seq 261, length 64
          22:55:57.029147 IP 172.17.0.2 > 254.red-80-58-61.staticip.rima-tde.net: ICMP echo
request, id 436, seq 262, length 64
          22:55:57.030912 IP 254.red-80-58-61.staticip.rima-tde.net > 172.17.0.2: ICMP echo reply,
id 436, seq 262, length 64
          …
```

To kill the ping process above, re-attach your tty and stdin to the container, identify the process, isolate its PID and kill it:

```
        $ docker attach a754719db594
        root@a754719db594:/# kill -9 $(awk ‘/ping/ {print $1}’ <<<“$(ps -e)”)
        ^p ^q
```

In the above example, we magically “guessed” what the virtual interface name was (we had the choice between 2 such interfaces, since there are only two containers running). In the `` `tcpdump' `` output, the IP address of the container where the ping process runs also appears; it is `172.17.0.2` .

In production situations however, there may be tens or hundreds of running containers in a cluster. In this case, we can automate the following cmds in a script to build a table of correspondence between container ID, container IP and container virtual interface, veth__. Starting from a container ID obtained from `` `$ docker ps -q' `` :

```
        $ docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' a754719db594
          172.17.0.2
        $ sudo ip link | grep $(docker exec 14dd5e0b3063 ethtool -S eth0 | awk '/peer_ifindex/
{print $2;}') | cut -d " " -f 2 | cut -d "@" -f 1
          veth2177159
```

The resulting bash script is listed in Appendix D.

8.4 – Listening to Docker inter-container communication

Let us now simulate message-passing between 2 arbitrary containers among many in a cluster. For instance one may decide to ping a container from another container.

Start a ping from one container (ID `a754719db594`, IP 172.17.0.2) to another (ID `a754719db594`, IP 172.17.0.3):

```
        $ docker exec a754719db594 ping -w60 172.17.0.3
          PING 172.18.0.3 (172.17.0.3) 56(84) bytes of data.
           bytes from 172.17.0.3: icmp_seq=1 ttl=64 time=0.070 ms
          64 bytes from 172.17.0.3: icmp_seq=2 ttl=64 time=0.053 ms
```

To see this traffic from the DC host, we can capture packets on either one of the known virtual interfaces corresponding to the containers, or we can capture traffic on the bridge network interface (`'docker0'`). The bridge network interface shows all inter-container communication on the corresponding subnet 172.17.0.0/16:

```
        $ sudo tcpdump -ni docker0 host 172.17.0.2 and host 172.17.0.3
          listening on docker0, link-type EN10MB (Ethernet), capture size 262144 bytes
          20:55:37.990831 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 14, seq 200, length 64
          20:55:37.990865 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 14, seq 200, length 64
          20:55:38.990828 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 14, seq 201, length 64
          20:55:38.990866 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 14, seq 201, length 64
```

`Equivalent data would be obtained by querying either containers’ virtual interface, using:`

```
        $ sudo tcpdump -i veth2177159 icmp  # for container ID a754719db594 with IP address 172.17.0.2
or      $ sudo tcpdump -i vethd8e05dd icmp  # for container ID e88238f68668 with IP address 172.17.0.3
```

8.5 – Summary

• A bridge network has a corresponding linux bridge interface on the docker host that acts as a layer2 switch, and which connects different containers on the same subnet.
• Each network interface in a container has a corresponding virtual interface on the docker host that is created while the container is running.
• A traffic capture from the docker host on the bridge interface is equivalent to configuring a SPAN port on a switch in that you can see all inter-container communication on that network.
• A traffic capture from the docker host on the virtual interface (veth-*) will show all traffic the container is sending on a particular subnet.
• Linux IPTables rules are used to block different networks (and sometimes hosts within the network) from communicating using the filter table. These rules are usually added in the DOCKER-ISOLATION chain.
• Containers communicating with the outside world through a bridge interface have their IP hidden behind the docker host's IP address. This is done by adding rules to the NAT table in IPTables.

9 – ***Miscellaneous***

For a refresher on Docker cmds, take a look at the excellent Docker Cheat Sheet on GitHub[13].
For security minded users, references on how to get started in that direction can be found here, in relation with *Docker Compose,* an orchestration tool for DCs.

13 https://github.com/wsargent/docker-cheat-sheet

Appendix D

Scope:

Launch containers from a publicly available image.
Map each container's ID, IP address and virtual interface in a host running multiple containers.
Join containers in a serf or etcd node cluster (implementation for serf only).

This script can be uploaded in an AWS instance. For best result it must be executed in an Debian based Linux environment. File output consists of:

```
setup.db
setup.log
```

```
#!/bin/bash
#
# ============================
# Author: Cedric Bhihe
# Email: cedric.bhihe@gmail.com
# Creation Date: 2017.05.30
# ============================

#                   initDC
#
# Script runs N serf agent(s) in so many Docker containers, creates
# cluster of serf nodes.
# Script executed from within the instance (host)  environment.
# Script called with exactly 2 arguments:
#   - the Docker Hub container image name
#   - N, nbr of DCs to create
# `/bin/bash' v4.0+ necessary for array support

# Update instance package environment
# sudo apt-get -q update; sudo apt-get -y upgrade
# sudo apt-get install -y bridge-utils iputils-ping

# Discover each container's ID, private and public IP and VETH
# Write them in $contDB txt file

datestamp="$(/bin/date +%Y%m%d-%H%M%S)"
# instance private IP
inst_privip="$(ifconfig eth0 | awk '/inet addr/ {print substr($2,6)}')"
# instance public IP
inst_pubip="$(wget -qO- http://instance-data/latest/meta-data/public-ipv4)"
# Docker containers and DC internal process-id database
contDB="setup.db"

# log file for container creation and serf agent configuration
/bin/cat <<EOF > setup.log
Log date: "$datestamp"
====================================
EOF

printf "Date: %s\nAWS EC2 instance host name: %s\n" "$datestamp" \
    "$(dig -x "$(curl -s checkip.amazonaws.com)" +short)" > "$contDB"
printf "Private instance IP: %s\nPublic instance IP: %s\n\n" \
    "$inst_privip" "$inst_pubip" >> "$contDB"
printf "DC-# DC-id     DC-privIP    DC-veth    Node-id\n" >> "$contDB"

declare -a serfID;  # declare array of serf/etcd agents' IDs
declare -a contID;  # declare array of DCs' IDs

# Nbr of DCs to be instantiated from image
dockerImage="$1"
maxDC="$2"

for j in $(seq 1 "$maxDC"); do

    # build node identification string
    node_id="$(printf "%s_%03d" "$inst_pubip" "$j")"
    # create containers, launch agents in each container and join them

    case "$j" in
         1)
                  serf_id="$(docker run -d --name serfDC"${j}" --rm \
                           -p 7946 -p 7373 -p 8443 \
                           "$dockerImage" agent -node="$node_id" -iface=eth0 )"
                  cont_id="$(docker ps -l -q)"
                  # discover container private IP
                  cont_privip="$(docker inspect -f \
                           '{{range .NetworkSettings.Networks}}{{.IPAddress}} {{en
d}}' \
                           "$cont_id")"
                  joinIP="$cont_privip"
                  ;;
         *)
                  serf_id="$(docker run -d --name serfDC"${j}" --rm \
                           -p 7946 -p 7373 -p 8443 \
                           "$dockerImage" agent -node="$node_id" -iface=eth0)"
                  cont_id="$(docker ps -l -q)"
                  docker exec -d "$cont_id" bash -c 'serf join '"$joinIP"
                  # discover container private IP
                  cont_privip="$(docker inspect -f \
                           '{{range .NetworkSettings.Networks}}{{.IPAddress}} {{en
d}}' \
                           "$cont_id")"
                  ;;
    esac

    # update container packages, store launch log
     serfID=("${serfID[@]}" "$serf_id")
     contID=("${contID[@]}" "$cont_id")
     printf "\nContainer ID: %s\n" "$cont_id" >> setup.log

     # update of packages inside DC and install ethtool
     docker exec "$cont_id" apk --update add --no-cache ethtool >> setup.log

    # discover metadata
    grep_str="$(docker exec "$cont_id" ethtool -S eth0 | \
        awk '/peer_ifindex/ {print $2}')"": veth"
     cont_veth="$(sudo ip link | grep "$grep_str"|awk '{print substr($2,1,11)}')"

    # clean up superfluous package
     docker exec "$cont_id" apk del ethtool >> setup.log
     docker exec "$cont_id" rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* \
          >> setup.log

    # create database
     printf "%d %s %s %s %s\n" \
         "$j" "$cont_id" "$cont_privip" "$cont_veth" "$serf_id" | tee -a "$contDB"

done

# printf "%d containers created.\n" "${#serfID[@]}"
docker ps
docker exec "$cont_id" serf members
```

Appendix E

List public IP addresses of running AWS instances

Install the AWS API on your local host. For that you can either set up a Python virtual environment (as described here) or install the API directly in your usual Python environment. We proceeded with the latter option.

```
$ pip install awscli
$ aws ec2 describe-instances \
--query "Reservations[*].Instances[*].PublicIpAddress" \
--output=text       # tab delimited text file
```

Controlling the output of AWS API cmds is well documented by AWS. Note that `' aws ec2 ...'` specifically queries the AWS EC2 service for running instances.

The following bash script fully automate the connextion to AWS and collects useful meta-data on te ongoing session. It is based on `'awscli'` and requires the previous configuration of an AWS-AMI, a AWS security group, credentials in the form of an access key-pair.

```
#!/bin/bash

# ==========================
# Author: Cedric Bhihe
# Email: cedric.bhihe@gamil.com
# Date: 2017.05.31
# ==========================

# Launch specific AWS instance from pre-built AMI with custom security group,
# connect to it and discover instance-ID and PublicIP
# Script requires bash v4+ or array-capable shell (zsh) for array support

instNbr=1
instType="t2.micro"
amiID="ami-d4b9b5ad"
sgID="sg-1187dd6a"
# keyFile="/home/ckb/Study/UPC/Subjects/DS_decentralized-systems/Project/upcfib_ds.pem"
keyFile="upcfib_ds.pem"
workDIR="$HOME/Study/UPC/Subjects/DS_decentralized-systems/Project/"

cd "$workDIR"

# Set credentials and settings locally if necessary
printf "AWS EC2 instance launch from an AMI.\n==================================\n\n"

read -p  " - Is package 'awscli' installed ? (y|n): " resp1

case "$resp1" in
    [nN] | [n|N][O|o] )
         printf "Please install 'awscli' first. Documentation is at:\n";
         printf "https://docs.aws.amazon.com/cli/latest/userguide/installing.htm
l.\n";
         printf "Script will exit now.\n"; exit 2
         ;;
    [yY] | [yY][Ee][Ss] )
         aws --version;
         read -p " - Is AWS CLI configured to access resources on AWS ? (y|n): "
 resp2;
         case $resp2 in
                  [nN] | [n|N][O|o] )
                         printf "Have your AWS Access Key ID and Secret Access K
ey ready...\n";
                         aws configure;
                         printf "\nProceed to launch instance ...\n"
                         ;;
                  [yY] | [yY][Ee][Ss] )
                         printf "Proceed to launch instance ...\n"
```

```
                          ;;
                 *)
                          printf "Answer (\"%s\") not understood. Start again.\n"
 "$resp2";
                          printf "Script will exit now.\n"; exit 3
                          ;;
         esac
         ;;
    * )
         printf "Answer (\"%s\") not understood. Start again.\n" "$resp1";
         printf "Script will exit now.\n"; exit 3
         ;;
esac

# Launch procedure
printf "\nVerify that relevant security group(s), key pair(s) and AMI(s)\n"
printf "already exist for your account. If not type CTRL+c to abort.\n"

#+ Assume that relevant security group(s), key pair(s) and AMI(s) already exist.
instID=""
instID="$(aws ec2 run-instances --image-id "$amiID" --security-group-ids "$sgID" --coun
t "$instNbr" --instance-type "$instType" --key-name "$keyFile" --query 'Instances[0].In
stanceId')"

time1="$(/bin/date +%s)"

printf "Launching %s instance(s) of type %s on AWS..." "$instNbr" "$instType"

# Discover launched instance's public IP
instPubIP="$(aws ec2 describe-instances --instance-ids "$instID" --query 'Reservations[
0].Instances[0].PublicIpAddress')"

printf "\nInstance with ID %s was launched.\n" "$instID"
printf "Instance public IP: %s\n" "$instPubIP"

# Discover launched instance's status

printf "Instance initialization (pending) ..."

instStat=""
waitcnt=0

while [ ! "$instStat" = "running" ]; do
    instStat="$(aws ec2 describe-instance-status --instance-ids "$instID" | \
    awk '/INSTANCESTATE/ {print $3}')"
    if [ "$((waitcnt%40))" -eq  "0" ]; then
         printf "\n="
    else
         printf "="
    fi
    waitcnt="$((waitcnt+1))"
done

time2="$(/bin/date +%s)"
printf "\n Instance RUNNING !\nTime elapsed till availability: %d\n" "$((time2-time1))"

# Discover instance reachability

declare -a instReach;
waitcnt=0
printf " ... building communication ...\n"
while [ ! "x${instReach[0]}" = "xpassed" -o ! "x${instReach[1]}" = "xpassed" ]; do
    instReach=( $(aws ec2 describe-instance-status --instance-ids "$instID" | awk '
/reachability/ {print $3}') )
    if [ "$((waitcnt%40))" -eq  "0" ]; then
         printf "\n="
    else
         printf "="
    fi
    waitcnt="$((waitcnt+1))"
done

time3="$(/bin/date +%s)"
printf " Instance REACHABLE !\nTime elapsed till reachability: %d\n" \
    "$((time3-time2))"

# SSH connect to instance (open instance shell in tty)

printf "\n SSH connect to instance, open instance shell in tty\n"

ssh -i "$keyFile" ubuntu@"$instPubIP"

exit 0
```