6-Jul-2018
16-Jul-2018, first round finished

Table of Contents

Consul DevOps HandBook

This is a comprehensive cookbook for running Consul in various scenarios. This is a work in progress.

Contributors

https://www.gitbook.com/@srikanthchandika

https://www.gitbook.com/@yadavsms

https://www.gitbook.com/@arunchaudhary09

https://www.gitbook.com/@talonx

Setting up a Simple Cluster

Terminology

Datacenter : This is defined by your design. It should ideally be a set of nodes that have low latency connections between them and are part of a private network.

Agent: Any Consul process is an agent. It can run in one of two modes - server and client.

Server : A Consul process that maintains cluster state, responds to RPC queries from clients, elects leaders using the Raft consensus protocol and participates in WAN gossip between datacenters.

Client : A stateless Consul process that accepts queries from applications and forwards them to server nodes.

Deployment Strategy

The Consul documentation recommends that you have either 3 or 5 Consul servers running in each data-center to avoid data loss in the event of a server failure. Consul servers are the components that do the heavy lifting. They store information about services and key/value information. An odd number of servers is necessary to avoid stalemate issues during elections.

The examples that follow assume 64-bit Ubuntu 14.04 servers but they should be valid for any Linux distribution with minor changes. We start with 4 machines whose details are:

Servername	IP Address	Role
server1.example.com	192.168.1.11	Consul Server
server2.example.com	192.168.1.12	Consul Server
server3.example.com	192.168.1.13	Consul Server
client.example.com	192.168.1.21	Consul Client

You will need to run most commands as sudo.

Our objective is to provide a configuration for the nodes that requires minimal or no manual intervention to keep running after the initial install and configuration.

Assumptions

We assume that all the machines have private IP addresses as defined by RFC 1918. We'll address this assumption in more detail later in the Network Environment section. An additional assumption is that the IP addresses of the machines don't change when they reboot.

Installation

We will be installing Consul on each machine. We are using the latest Consul version as of this writing (0.6.4).

Download the consul program from the Consul project's page and follow the steps below:

Download the version corresponding to your operating system and architecture. In this guide, since we are using 64-bit servers, we will use the "amd64" link under "linux".

In your terminal, change your directory to /usr/local/bin, where we will keep the executable.

```
root@server1:~#cd /usr/local/bin

root@server1:~#wget https://releases.hashicorp.com/consul/0.6.4/consul_0.6.4_linux_amd
64.zip
```

Now, we can extract the binary package and we can then remove the zipped file.

`root@server1:~#unzip consul_0.6.4_linux_amd64.zip`

```
root@server1:~#consul version
Consul v0.6.4
Consul Protocol: 3 (Understands back to: 1)
```

Repeat these steps on all your machines. You should now have the consul command available on all of your nodes.

Create the necessary directories

We can easily try out Consul in an unstructured way by using the `consul` command. This will allow you to test out some functionality and to get familiar with the software. But in a real world environment having an organized directory structure makes things easier to maintain. Complete the following steps on each of your servers and clients.

We will run Consul processes with a dedicated user.

Create the user:

```
root@server1:~#useradd -m consul

root@server1:~#passwd consul
Enter new UNIX password: consul
Retype new UNIX password: consul
passwd: password updated successfully

root@server1:~#usermod -aG sudo consul
```

We will next create the configuration directories where we can store Consul configuration. We will make a parent `consul.d` directory in `/etc` and create a subdirectory `server` for the server nodes and a `client` directory for the client node. We can put our configuration as a JSON file inside these directories. Having this structure is not mandatory but it helps to keep things organized.

```
root@server1:~#mkdir -p /etc/consul.d/server
```

We also need to create a location where Consul can store persistant data between reboots. We will create this directory at `/var/consul` and change its ownership to the `consul` user so that it can manage the data:

```
root@server1:~#mkdir /var/consul
root@server1:~#chown consul: /var/consul
```

All our directories are now in place, now we will create configuration files on each node under `/etc/consul.d/server/`

Before creating the configuration files we need to understand what is bootstrapping.

Bootstrapping

An agent can run in both client and server mode. Server nodes are responsible for running the consensus protocol and storing the cluster state. The client nodes are mostly stateless and rely heavily on the server nodes.

Before a Consul cluster can begin to service requests, a server node must be elected leader. Thus, the first nodes that are started are generally the server nodes. Bootstrapping is the process of joining these initial server nodes into a cluster.

The recommended way to bootstrap is to use the -bootstrap-expect option. This option informs Consul of the expected number of server nodes and automatically starts the leader election process when that many servers are available. To prevent inconsistencies and split-brain situations (that is, clusters where multiple servers consider themselves leader), all servers should either specify the same value for -bootstrap-expect or specify no value at all. Only servers that specify a value will attempt to bootstrap the cluster.

Consul document recommend 3 or 5 total servers per datacenter. A single server deployment is highly discouraged as data loss is inevitable in a failure scenario. Please refer to the deployment table for more detail.

Here we are considering a 3 server cluster. We can start Node1, Node2, and Node3 with the -bootstrap-expect flag set to 3 in each. Once the first node is started, you should see a message like below:

```
[WARN] raft: EnableSingleNode disabled, and no known peers. Aborting election.
```

This indicates that the node is expecting 2 peers but none are known yet. You will see a similar message when you start the second node. To prevent a split-brain scenario, the servers will not elect themselves leader until the number specified in bootstrap-expect comes up.

Once you start the third node, you should see leader election taking place and one of the server nodes will elect itself the leader, you should see a INFO something similar to:

```
[INFO] raft: Election won. Tally: 2
[INFO] raft: Node at 192.168.1.11:8300 [Leader] entering Leader state
[INFO] consul: cluster leadership acquired
[INFO] consul: New leader elected: ConsulServer1
```

Manual Bootstrapping

In versions of Consul prior to 0.4, bootstrapping was a manual process. For details on using the -bootstrap flag directly, see the manual bootstrapping guide. Manual bootstrapping is not recommended as it is more error-prone than automatic bootstrapping with -bootstrap-expect. It is recommended to use -bootstrap-expect but we mention this for completeness.

Server Configuration

The configuration files are stored as JSON files. Consul has built-in encryption support using a shared-secret system for the gossip protocol.

In the terminal, we can use the Consul command to generate a key of the necessary length and encoding as follows:

```
root@server1:~#consul keygen
EXz7LFN8hpQ4id8EDYiFoQ==
```

It should be the same for all the servers and clients in a datacenter. If it's different the consul members will refuse to join.

Create the first file in the /etc/consul.d/server directory in server1.example.com (IP Address: 192.168.1.11). Use the encryption key generated above as the value for the "encrypt" key in the JSON.

server1.example.com

```
root@server1:~#vim /etc/consul.d/server/config.json
{
"bind_addr": "192.168.1.11",
"datacenter": "dc1",
"data_dir": "/var/consul",
"encrypt": "EXz7LFN8hpQ4id8EDYiFoQ==",
"log_level": "INFO",
"enable_syslog": true,
"enable_debug": true,
"node_name": "ConsulServer1",
"server": true,
"bootstrap_expect": 3,
"leave_on_terminate": false,
"skip_leave_on_interrupt": true,
"rejoin_after_leave": true,
"retry_join": [
  "192.168.1.11:8301",
  "192.168.1.12:8301",
  "192.168.1.13:8301"
  ]
 }
```

More information on what each key means can be found in the documentation.

You should copy the contents of this configuration file to the other machines that will be acting as your Consul servers. Place them in the same location just like you did in the first server.

The only value you need to modify on the other servers is the IP addresses that it should attempt to listen on - the value of the 'bind_addr' key.

server2.example.com

```
root@server2:~#vim /etc/consul.d/server/config.json
{
"bind_addr": "192.168.1.12",
"datacenter": "dc1",
"data_dir": "/var/consul",
"encrypt": "EXz7LFN8hpQ4id8EDYiFoQ==",
"log_level": "INFO",
"enable_syslog": true,
"enable_debug": true,
"node_name": "ConsulServer2",
"server": true,
"bootstrap_expect": 3,
"leave_on_terminate": false,
"skip_leave_on_interrupt": true,
"rejoin_after_leave": true,
"retry_interval": "30s",
"retry_join": [
  "192.168.1.11:8301",
  "192.168.1.12:8301",
  "192.168.1.13:8301"
  ]
 }
```

server3.example.com

```
root@server3:~#vim /etc/consul.d/server/config.json
{
"bind_addr": "192.168.1.13",
"datacenter": "dc1",
"data_dir": "/var/consul",
"encrypt": "EXz7LFN8hpQ4id8EDYiFoQ==",
"log_level": "INFO",
"enable_syslog": true,
"enable_debug": true,
"node_name": "ConsulServer3",
"server": true,
"bootstrap_expect": 3,
"leave_on_terminate": false,
"skip_leave_on_interrupt": true,
"rejoin_after_leave": true,
"retry_interval": "30s",
"retry_join": [
  "192.168.1.11:8301",
  "192.168.1.12:8301",
  "192.168.1.13:8301"
  ]
 }
```

Starting the cluster

Now we have everything in place to get a consul cluster up and running.

Start server1.example.com first.

```
root@server1:~#su consul

consul@server1:~$consul agent -config-dir /etc/consul.d/server/

==> WARNING: Expect Mode enabled, expecting 3 servers
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Joining cluster...
    Join completed. Synced with 1 initial agents
==> Consul agent running!
     Node name: 'ConsulServer1'
    Datacenter: 'dc1'
        Server: true (bootstrap: false)
   Client Addr: 127.0.0.1 (HTTP: 8500, HTTPS: -1, DNS: 8600, RPC: 8400)
  Cluster Addr: 192.168.1.11 (LAN: 8301, WAN: 8302)
Gossip encrypt: true, RPC-TLS: false, TLS-Incoming: false
         Atlas: <disabled>
```

The service should start up and occupy the terminal window. It will attempt to connect to the two other servers and keep retrying until they are up.

Now start Consul in the other two servers (server2.example.com, server3.example.com) in the same manner.

```
server2.example.com

root@server2:~#su consul

consul@server2:~$consul agent -config-dir /etc/consul.d/server/

==> WARNING: Expect Mode enabled, expecting 3 servers
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Joining cluster...
    Join completed. Synced with 1 initial agents
==> Consul agent running!
     Node name: 'ConsulServer2'
    Datacenter: 'dc1'
        Server: true (bootstrap: false)
   Client Addr: 127.0.0.1 (HTTP: 8500, HTTPS: -1, DNS: 8600, RPC: 8400)
  Cluster Addr: 192.168.1.12 (LAN: 8301, WAN: 8302)
Gossip encrypt: true, RPC-TLS: false, TLS-Incoming: false
         Atlas: <disabled>

server3.example.com

root@server3:~#su consul

consul@server3:~$consul agent -config-dir /etc/consul.d/server/

==> WARNING: Expect Mode enabled, expecting 3 servers
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Joining cluster...
    Join completed. Synced with 1 initial agents
==> Consul agent running!
     Node name: 'ConsulServer3'
    Datacenter: 'dc1'
        Server: true (bootstrap: false)
   Client Addr: 127.0.0.1 (HTTP: 8500, HTTPS: -1, DNS: 8600, RPC: 8400)
  Cluster Addr: 192.168.1.13 (LAN: 8301, WAN: 8302)
Gossip encrypt: true, RPC-TLS: false, TLS-Incoming: false
         Atlas: <disabled>
```

These servers (server2.example.com, server3.example.com) will connect to the server1.example.com, completing the cluster with one of the servers as the leader and the other two as followers.

You can see the members of the cluster (servers and clients) by asking consul for its members on any of the machines:

```
root@server3:~#consul members
```

Node	Address	Status	Type	Build	Protocol	DC
ConsulServer1	192.168.1.11:8301	alive	server	0.6.4	2	dc1
ConsulServer2	192.168.1.12:8301	alive	server	0.6.4	2	dc1
ConsulServer3	192.168.1.13:8301	alive	server	0.6.4	2	dc1

The cluster is fully operational now and applications can connect. We'll configure and start the client next.

Client Configuration

The client is also a member of the system, and can connect to servers for information about the infrastructure.

Consul clients are very light-weight and simply forward requests to the servers. They provide a method of insulating your servers and offload the responsibility of knowing the servers' addresses from the applications that use Consul.

Create the configuration file under the /etc/consul.d/client directory.

```
root@client:~#vim /etc/consul.d/client/config.json
{
 "bind_addr": "192.168.1.21",
 "datacenter": "dc1",
 "data_dir": "/var/consul",
 "encrypt": "EXz7LFN8hpQ4id8EDYiFoQ==",
 "log_level": "INFO",
 "enable_syslog": true,
 "enable_debug": true,
 "node_name": "ConsulClient",
 "server": false,
   "service": {"name": "Apache", "tags": ["HTTP"], "port": 80,
   "check": {"script": "curl localhost >/dev/null 2>&1", "interval": "10s"}},
 "rejoin_after_leave": true,
 "retry_join": [
    "192.168.1.11",
    "192.168.1.12",
    "192.168.1.13"
    ]
 }
```

Start Consul

```
root@client:~#su consul

consul@client:~$consul agent -config-dir /etc/consul.d/client

consul@client:~$consul members
```

Node	Address	Status	Type	Build	Protocol	DC
ConsulClient	192.168.1.21:8301	alive	client	0.6.4	2	dc1
ConsulServer1	192.168.1.11:8301	alive	server	0.6.4	2	dc1
ConsulServer2	192.168.1.12:8301	alive	server	0.6.4	2	dc1
ConsulServer3	192.168.1.13:8301	alive	server	0.6.4	2	dc1

You can see that the client has joined the cluster. Let us try to list the available services from this client using the REST API.

```
consul@client:~$curl -s http://client.example.com:8500/v1/catalog/services
{"ConsulClient": {
   "ID":"ConsulClient",
   "Service":"Apache",
   "Tags":["HTTP"],
   "Address":"",
   "Port":80,
   "EnableTagOverride":false,
   "CreateIndex":0,
   "ModifyIndex":0}
   }
```

We can also list the members via the same REST endpoint.

```
consul@client:~$curl -s http://client.example.com:8500/v1/agent/members

 [{"Name":"ConsulClient",
    "Addr":"192.168.1.21",
    "Port":8301,
    "Tags": {"build":"0.6.4:26a0ef8c",
        "dc":"dc1",
        "role":"node",
        "vsn":"2",
        "vsn_max":"3",
        "vsn_min":"1"},
    "Status":1,
    "ProtocolMin":1,
    "ProtocolMax":3,
    "ProtocolCur":2,
    "DelegateMin":2,
    "DelegateMax":4,
    "DelegateCur":4
   },
 {"Name":"ConsulServer1",
    "Addr":"192.168.1.11",
    "Port":8301,
    "Tags":{"build":"0.6.4:26a0ef8c",
        "dc":"dc1",
```

```
          "expect":"3",
          "port":"8300",
          "role":"consul",
          "vsn":"2",
          "vsn_max":"3",
          "vsn_min":"1"},
      "Status":1,
      "ProtocolMin":1,
      "ProtocolMax":3,
      "ProtocolCur":2,
      "DelegateMin":2,
      "DelegateMax":4,
      "DelegateCur":4
    },
  {"Name":"ConsulServer2",
     "Addr":"192.168.1.12",
     "Port":8301,
     "Tags":{"build":"0.6.4:26a0ef8c",
          "dc":"dc1",
          "expect":"3",
          "port":"8300",
          "role":"consul",
          "vsn":"2",
          "vsn_max":"3",
          "vsn_min":"1"},
     "Status":1,
     "ProtocolMin":1,
     "ProtocolMax":3,
     "ProtocolCur":2,
     "DelegateMin":2,
     "DelegateMax":4,
     "DelegateCur":4
    },
   {"Name":"ConsulServer3",
     "Addr":"192.168.1.13",
     "Port":8301,
     "Tags":{"build":"0.6.4:26a0ef8c",
          "dc":"dc1",
          "expect":"3",
          "port":"8300",
          "role":"consul",
          "vsn":"2",
          "vsn_max":"3",
          "vsn_min":"1"},
    "Status":1,
    "ProtocolMin":1,
    "ProtocolMax":3,
    "ProtocolCur":2,
    "DelegateMin":2,
    "DelegateMax":4,
    "DelegateCur":4
    }]
```

In case the client fails or is shutdown, after a restart it will rejoin the cluster automatically, as the required parameters are in JSON configuration.

Starting as a Boot-time Service

Till now we have downloaded and run Consul from the command line, but we want it to autostart whenever the machine reboots. The cluster should keep on functioning whenever this happens. We need to have the right mix of configuration and auto-start scripts so that no manual intervention becomes necessary once the cluster has been booted once.

Ubuntu

Upstart is an event-based replacement for the /sbin/init daemon which handles starting of tasks and services during boot, stopping them during shutdown and supervising them while the system is running.

The Upstart script will be identical on all the servers. The client script will be slightly different.

Create a file within the `/etc/init` directory to hold Consul configuration on all servers and clients.

In Servers

```
root@server1:~#sudo vim /etc/init/consul.conf

description "Consul server process"

start on (local-filesystems and net-device-up IFACE=eth0)
stop on runlevel [!12345]

respawn

setuid consul
setgid consul

exec consul agent -config-dir /etc/consul.d/server
```

A brief description of what each command means follows:

```
    * "start" - a job or emit an event.
    * "local-filesystems" - A service that wishes to be running once local filesystems
 are mounted.
    * "net-device-up IFACE=eth0" - A service that wishes to be running when device eth
0 is running.
    * "stop on runlevel [!12345]" - stop the process when halting or rebooting the ser
ver.
    * "respawn" - restart the process if it ever dies unexpectedly.
    * "setuid & setgid" - specifying the user and group that the process should run un
der.
    * "exec" - providing the actual command that we want to run.
```

Before proceeding, make sure that

- The upstart package is installed.
- The user and group which are specified in the `init/consul.conf` are created.
- The `/var/consul` directory belongs to the consul user.

Copy the contents of the file called `/etc/init/consul.conf` to each of the servers and the client.

We need to change the configuration directory that is passed into the actual `consul` command in the client's version of the file.

In Client

```
#vim /etc/init/consul.conf

description "Consul Client process"

start on (local-filesystems and net-device-up IFACE=eth0)
stop on runlevel [!12345]

respawn

setuid consul
setgid consul

exec consul agent -config-dir /etc/consul.d/client
```

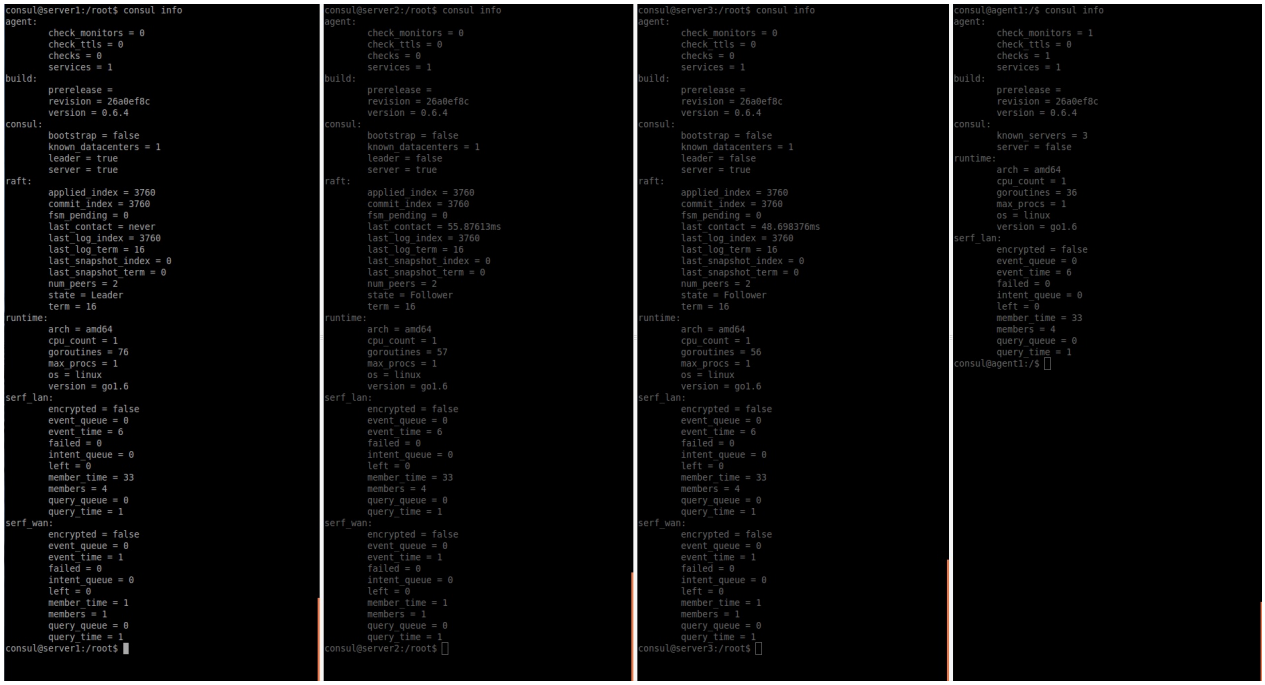
Starting the cluster with upstart scripts

As root, start the Consul service in all the servers and the client.

```
root@server1:~#start consul

root@server1:~#su consul

consul@client:~$service consul status
consul start/running, process 5657

consul@client:~$consul members
```

Servername	IP Address	Role
server1.example.com	192.168.1.11	Consul Server
server2.example.com	192.168.1.12	Consul Server
server3.example.com	192.168.1.13	Consul Server
client.example.com	192.168.1.21	Consul Client

```
consul info
```



* Centos

TBD

* RHEL

TBD

Adding/Removing Servers

We can add Consul servers to the existing cluster but this has to be done carefully. A minimum quorum of servers (N/2)+1 must be available for processing changes. This means that if there are 3 server nodes, at least 2 must be available.

Removing servers must be done carefully too to avoid causing an availability outage. For a cluster of N servers, at least (N/2)+1 must be available for the cluster to function. See this deployment table. If you have 3 servers, and 1 of them is currently failed, removing any servers will cause the cluster to become unavailable.

For this reason, the Consul documentation recommends adding servers first when you're both adding and removing.

Adding New Servers

In the previous chapters we have setup a 3-server Consul cluster. Now we will add 2 new servers to it.

Our desired final configuration is as follows:

Servername	IP Address	Role
server1.example.com	192.168.1.11	Consul Server
server2.example.com	192.168.1.12	Consul Server
server3.example.com	192.168.1.13	Consul Server
server4.example.com	192.168.1.14	Consul Server
server5.example.com	192.168.1.15	Consul Server

On the new server nodes, create the config.json file as before. Everything remains same except for the retry_join and the start_join parameters, which will refer to the 5 servers now and the bootstrap_expect option, which was set to 3.

Here the question arises as to whether we should have the bootstrap_expect 3 or bootstrap_expect 5?

If we set it to 5, should we restart agents 1, 2 and 3 with the new bootstrap_expect value? The bootstrap_expect is used only once in the history of the cluster. It is used for the initial bootstrap, for example the first time that a leader is elected. So we would start with

bootstrap_expect 3, but then the new servers do not even need the bootstrap_expect flag. They are joining an existing cluster which has a leader, so they are not performing a bootstrapping of the leader.

`/etc/consul.d/server/config.json` will look like:

server4.example.com

```
root@server4:~#vim /etc/consul.d/server/config.json
{
"bind_addr": "192.168.1.14",
"datacenter": "dc1",
"encrypt": "EXz7LFN8hpQ4id8EDYiFoQ==",
"data_dir": "/var/consul",
"log_level": "INFO",
"enable_syslog": true,
"enable_debug": true,
"node_name": "ConsulServer4",
"server": true,
"leave_on_terminate": false,
"skip_leave_on_interrupt": true,
"rejoin_after_leave": true,
"retry_join": [
  "192.168.1.11:8301",
  "192.168.1.12:8301",
  "192.168.1.13:8301",
  "192.168.1.14:8301",
  "192.168.1.15:8301"
  ]
 }
```

server5.example.com

```
root@server5:~#vim /etc/consul.d/server/config.json
{
"bind_addr": "192.168.1.15",
"datacenter": "dc1",
"data_dir": "/var/consul",
"encrypt": "EXz7LFN8hpQ4id8EDYiFoQ==",
"log_level": "INFO",
"enable_syslog": true,
"enable_debug": true,
"node_name": "ConsulServer5",
"server": true,
"leave_on_terminate": false,
"skip_leave_on_interrupt": true,
"rejoin_after_leave": true,
"retry_join": [
  "192.168.1.11:8301",
  "192.168.1.12:8301",
  "192.168.1.13:8301",
  "192.168.1.14:8301",
  "192.168.1.15:8301"
  ]
 }
```

It is best to add servers one at a time, allowing them to catch up. This avoids the possibility of data loss in case the existing servers fail while bringing the new servers up-to-date.

Start the server4.example.com to add it to the existing cluster.

```
root@server4:~# consul agent -config-dir /etc/consul.d/server/
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Joining cluster...
    Join completed. Synced with 4 initial agents
==> Consul agent running!
     Node name: 'ConsulServer4'
    Datacenter: 'dc1'
        Server: true (bootstrap: false)
   Client Addr: 127.0.0.1 (HTTP: 8500, HTTPS: -1, DNS: 8600, RPC: 8400)
  Cluster Addr: 192.168.1.14 (LAN: 8301, WAN: 8302)
Gossip encrypt: true, RPC-TLS: false, TLS-Incoming: false
         Atlas: <disabled>
==> Log data will now stream in as it occurs:

[INFO] raft: Node at 192.168.2.14:8300 [Follower] entering Follower state
----------------------------
----------------------------
----------------------------
[INFO] agent: Joining cluster...
[INFO] agent: (LAN) joining: [192.168.1.11:8301 192.168.1.12:8301 192.168.1.13:8301 19
2.168.1.14:8301 192.168.1.15:8301]
----------------------------
[INFO] agent: Join completed. Synced with 4 initial agents
[INFO] agent: Synced service 'consul'
```

Now start the server5.example.com.

Removing Servers

It is better to first add the new nodes and then remove the old nodes.

Once you have verified the existing servers are healthy, and that the cluster can handle a node leaving, the actual process is simple. You simply issue a leave command to the server.

In our example there are 5 servers in the cluster which can tolerate 2 server failures of the cluster as per the formula. We will remove the 4th server server4.example.com (192.168.1.14).

server4.example.com

```
root@server4:~# consul leave
Graceful leave complete
```

The logs should show something like:

```
...
[INFO] consul: server starting leave
...
[INFO] consul: removing LAN server ConsulServer4 (Addr: 192.168.1.14:8300) (DC: dc1)
[INFO] raft: Removed ourself, transitioning to follower
[INFO] agent: requesting shutdown
[INFO] consul: shutting down server
[INFO] agent: shutdown complete
```

The leader logs will show something like:

```
...
[INFO] consul: removing LAN server ConsulServer4 (Addr: 192.168.1.14:8300) (DC: dc1)
[INFO] raft: Removed peer 192.168.1.14:8300, stopping replication (Index: 3742)
[INFO] consul: removed server 'ConsulServer4' as peer
[INFO] consul: member 'ConsulServer4' left, deregistering
[INFO] raft: aborting pipeline replication to peer 192.168.1.14:8300
...
```

At this point the node has been gracefully removed from the cluster, and will shut down.

Note that IPs of nodes which have left gracefully (i.e. removed using a command) will be deleted from the alive nodes list - the /var/consul/raft/peers.json file. The contents of the peers.json in our case will be:

```
root@server1:~# cat /var/consul/raft/peers.json
["192.168.1.12:8300","192.168.1.13:8300","192.168.1.11:8300","192.168.1.15:8300"]
```

Now we have 4 nodes in the cluster without any failure or downtime.

```
root@server1:~# consul members
```

Node	Address	Status	Type	Build	Protocol	DC
ConsulServer1	192.168.1.11:8301	alive	server	0.6.4	2	dc1
ConsulServer2	192.168.1.12:8301	alive	server	0.6.4	2	dc1
ConsulServer3	192.168.1.13:8301	alive	server	0.6.4	2	dc1
ConsulServer5	192.168.1.15:8301	alive	server	0.6.4	2	dc1

Forced Removal

In some cases, it may not be possible to gracefully remove a server. For example, if the server hardware fails, then there is no way to issue a leave command. Instead, the cluster will detect the failure and replication will continuously retry.

If the server can be recovered, it is best to bring it back online and then make it gracefully leave the cluster.

If this is not a possibility, then the force-leave command can be used to force removal of the server. This can be done by invoking the command with the name of the failed node. At this point, the cluster leader will mark the node as having left the cluster and it will stop attempting to replicate.

Verifying Newly Added Servers

Run `consul info` on the leader. In this example the leader is server1.example.com, it may be different in your case.

```
root@server1:~# consul info
...
raft:
    applied_index = 2830
    commit_index = 2830
    fsm_pending = 0
    last_contact = never
    last_log_index = 2830
    last_log_term = 7729
    last_snapshot_index = 0
    last_snapshot_term = 0
    num_peers = 3
    state = Leader
    term = 7729
...
```

This shows various information about the state of Raft. In particular the `last_log_index` shows the last log that is on disk.

The same info command can be run on the newly added server to see how far behind it is. Eventually the server will catch up, and the values should match.

```
root@server4:~# consul info
...
raft:
    applied_index = 2830
    commit_index = 2830
    fsm_pending = 0
    last_contact = 91.32694ms
    last_log_index = 2830
    last_log_term = 7729
    last_snapshot_index = 0
    last_snapshot_term = 0
    num_peers = 3
    state = Follower
    term = 7729
...
```

The same thing can be done on the 5th server node.

You can see the members of the cluster by asking Consul for its members on any of the machines:

```
root@server4:~# consul members
```

Node	Address	Status	Type	Build	Protocol	DC
ConsulServer1	192.168.1.11:8301	alive	server	0.6.4	2	dc1
ConsulServer2	192.168.1.12:8301	alive	server	0.6.4	2	dc1
ConsulServer3	192.168.1.13:8301	alive	server	0.6.4	2	dc1
ConsulServer4	192.168.1.14:8301	alive	server	0.6.4	2	dc1
ConsulServer5	192.168.1.15:8301	alive	server	0.6.4	2	dc1

Network Environments

For running Consul we should have at least one network adapter besides the loopback interface. Consul prefers private IP addresses (as defined in RFC 1918) for communication by default.

If your machine has only a loopback interface, Consul will start but it will not be a part of any cluster.

Based on the available network adapters and the type of network (public/private) that they are connected to, Consul behaviour will vary and the appropriate configuration in each case is different.

The possible cases are

- Single IP Address, private range
- Single IP Address, public range
- Multiple IP Addresses, 0 private, 2 or more public
- Multiple IP Addresses, 1 private, 1 or more public
- Multiple IP Addresses, 2 private, others (or none) public

A summary of Consul's behavior in these situations follows

- Consul prefers private IP addresses by default, if there is only one, no configuration is required.
- In the case of multiple private IP addresses, Consul won't start as it does not choose any of them by itself, and specific configuration is required.
- In case the machine has public IP addresses only, specific configuration is required.
- The relevant configuration parameters are `bind_addr` & `advertise_addr`

We'll cover all the different combinations in this chapter.

Single IP Address, private range

RFC 1918 Private IP Addresses (In our example - 192.168.1.0/24)

We have only one network adapter excluding loopback on each server.

Let us assume below initial configuration is same for all in the networking environments.

```
root@server1:~# cat /etc/consul.d/server/config.json
{
  "datacenter": "dc1",
  "data_dir": "/var/consul",
  "encrypt": "EXz7LFN8hpQ4id8EDYiFoQ==",
  "log_level": "INFO",
  "enable_syslog": true,
  "enable_debug": true,
  "node_name": "ConsulServer1",
  "server": true,
  "leave_on_terminate": false,
  "skip_leave_on_interrupt": true,
  "rejoin_after_leave": true
}
```

Start Consul

```
root@server1:~# consul agent -config-dir /etc/consul.d/server/
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
```

Consul is running without specifying the bind address parameter because by default Consul will use the available private IPv4 address.

Since we have one private IP, Consul chose it.

Single IP Address, public range

In our examples, we use 173.28.1.0/24

We have only one network adapter excluding loopback.

Start Consul

```
root@server1:~# consul agent -config-dir /etc/consul.d/server/
==> WARNING: Expect Mode enabled, expecting 3 servers
==> Starting Consul agent...
==> Error starting agent: Failed to get advertise address: No private IP address found
root@server1:~#
```

Consul does not bind to or advertise on a public IP by default for security reasons.

We have to provide `"bind_addr": "173.28.1.11"` in the configuration to override the default behavior and force it to bind to a public address.

```
root@server1:~# consul agent -config-dir /etc/consul.d/server/
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
```

Multiple IP Addresses, 0 private, 2 or more public

We have 2 network adapters excluding loopback.

In our example, the IPs are in the following subnets:

1st IP - 173.28.1.0/24

2nd IP - 173.28.2.0/24.

Start Consul

```
root@server1:~# consul agent -config-dir /etc/consul.d/server/
==> WARNING: Expect Mode enabled, expecting 3 servers
==> Starting Consul agent...
==> Error starting agent: Failed to get advertise address: No private IP address found
root@server1:~#
```

Consul does not bind or advertise to a public IP by default for security reasons.

We can bind any one of the IP addresses to Consul.

Provide the `"bind_addr": "173.28.1.11"` key-value in the configuration to override the default behavior and force it to bind to a public address.

```
root@server1:~# consul agent -config-dir /etc/consul.d/server/
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
```

Multiple IP Addresses, 1 private, 1 or more public

In our example, the public IPs are in the following subnets:

1st IP - 173.28.1.0/24

2nd IP - 173.28.2.0/24

and the private IP is in the following:

3rd IP - 192.168.1.0/24

Start Consul

```
root@server1:~# consul agent -config-dir /etc/consul.d/server/
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
```

Consul started without any problem because we have one private IP.

If we want to bind to one of the public IPs we can provide the `bind_addr` key by specifying it in the configuration.

Multiple IP Addresses, 2 private, others (or none) public

In our example, the public IPs are in the following subnets:

1st IP - 173.28.1.0/24

2nd IP - 173.28.2.0/24

and the private ones are in the following:

3rd IP - 192.168.1.0/24

4th IP - 192.168.2.0/24

Start Consul

```
root@server1:~# consul agent -config-dir /etc/consul.d/server/```
==> Starting Consul agent...
==> Error starting agent: Failed to get advertise address: Multiple private IPs found.
 Please configure one.
root@server1:~#
```

Consul won't start as it does not choose one of the private IPs by itself. As before we have to specify the `bind_addr` key.

Consul with Containerized Microservices

This section describes how Registrator is used in conjunction with Consul in an environment running microservices inside Docker containers.

Deploying Consul in Docker Containers

The first thing we will do is start our Consul cluster. We are going to use a 3 node cluster in the subnet range of 172.16.20.0.

We'll run the following on the first node (Node1):

```
$export HOST_IP=$(ifconfig eth0 | grep 'inet ' |  cut -d: -f2 | awk '{ print $1 }')

$export DOCKER_BRIDGE_IP=$(ifconfig docker0 | grep 'inet ' |  cut -d: -f2 | awk '{ pri
nt $1 }')

$docker run -d -h $HOSTNAME -v /mnt:/data \
-p $HOST_IP:8300:8300 \
-p $HOST_IP:8301:8301 \
-p $HOST_IP:8301:8301/udp \
-p $HOST_IP:8302:8302 \
-p $HOST_IP:8302:8302/udp \
-p $HOST_IP:8400:8400 \
-p $HOST_IP:8500:8500 \
-p $DOCKER_BRIDGE_IP:53:53/udp \
progrium/consul -server -advertise $HOST_IP -bootstrap-expect 3
```

As part of this docker run command, we are binding all Consul's internal ports to the private IP address of our first host, except for the DNS port (53) which is exposed only on the docker0 interface (172.17.0.1 by default). The reason we use the docker bridge interface for the DNS server is that we want all the containers running on the same host to query this DNS interface, but we don't want anyone from outside doing the same. Since each host will be running a Consul agent, each container will query its own host. We also added the -advertise flag to tell Consul that it should use the host's IP instead of the docker container's IP.

On the second host (Node2), we'll run the the same thing but with the additional parameter -join pointing to the first node's IP:

```
$export HOST_IP=$(ifconfig eth0 | grep 'inet ' |  cut -d: -f2 | awk '{ print $1 }')

$export DOCKER_BRIDGE_IP=$(ifconfig docker0 | grep 'inet ' |  cut -d: -f2 | awk '{ pri
nt $1 }')

$docker run -d -h $HOSTNAME -v /mnt:/data \
-p $HOST_IP:8300:8300 \
-p $HOST_IP:8301:8301 \
-p $HOST_IP:8301:8301/udp \
-p $HOST_IP:8302:8302 \
-p $HOST_IP:8302:8302/udp \
-p $HOST_IP:8400:8400 \
-p $HOST_IP:8500:8500 \
-p $DOCKER_BRIDGE_IP:53:53/udp \
progrium/consul -server -advertise $HOST_IP -join 172.16.20.132
```

Note: 172.16.20.132 is the first node's IP address.

We'll do the same for the third node (Node3):

```
$export HOST_IP=$(ifconfig eth0 | grep 'inet ' |  cut -d: -f2 | awk '{ print $1 }')

$export DOCKER_BRIDGE_IP=$(ifconfig docker0 | grep 'inet ' |  cut -d: -f2 | awk '{ pri
nt $1 }')

$docker run -d -h $HOSTNAME -v /mnt:/data \
-p $HOST_IP:8300:8300 \
-p $HOST_IP:8301:8301 \
-p $HOST_IP:8301:8301/udp \
-p $HOST_IP:8302:8302 \
-p $HOST_IP:8302:8302/udp \
-p $HOST_IP:8400:8400 \
-p $HOST_IP:8500:8500 \
-p $DOCKER_BRIDGE_IP:53:53/udp \
progrium/consul -server -advertise $HOST_IP -join 172.16.20.132
```

Registrator Deployment

Registrator is a host-level service that can be run as a Docker container. It watches for new containers, inspects them for service information, and registers them with Consul. It also deregisters them when the container dies. Once our Consul cluster is up and running we can start the Registrator container.

We'll take an example where we deploy a dummy microservice in a container. Registrator needs to be deployed on the same machine (bare metal or virtual) where the microservices are going to be run.

If Registrator needs to run on a separate machine rather than where the Consul container is running then we need to do the following

```
$export CONSUL_IP=172.16.20.132

$docker run -d -v /var/run/docker.sock:/tmp/docker.sock --name registrator -h $HOSTNAM
E  gliderlabs/registrator:latest consul://$CONSUL_IP:8500
```

If Registrator is on the same machine as where the Consul container is, then we need to do the following:

```
$docker run -d -v /var/run/docker.sock:/tmp/docker.sock --name registrator -h $HOSTNAM
E  gliderlabs/registrator:latest consul://$HOST_IP:8500
```

Here we are mounting our “/var/run/docker.sock” file to the container. This file is a Unix socket where the docker daemon listens for events. This is actually how the docker client and the docker daemon communicate, through a REST API accessible from this socket. The important thing to know is that by listening on the same port as Docker, Registrator is able to know everything that happens with Docker on that host.

If you check the logs of the “registrator” container you’ll see a bunch of stuff and a message in the end indicating that it is waiting for new events. You should run the same commands on the other 2 nodes(VM) to start Registrator on those.

```
$docker logs registrator
2016/05/13 14:35:15 Starting registrator v7 ...
2016/05/13 14:35:15 Using consul adapter: consul://172.16.20.132:8500
2016/05/13 14:35:15 Connecting to backend (0/0)
2016/05/13 14:35:15 consul: current leader  172.16.20.132:8300
2016/05/13 14:35:15 Listening for Docker events ...
```

To summarize what we have done so far, we have 3 different hosts running a Consul agent and a Registrator container each. The Registrator instance on each host watches for changes in docker containers on that host and talks to the local or remote Consul agent as the case may be.

Deploying the Microservice in a Container

We will first run our dummy microservice container on one of the hosts:

```
$docker run -d --name service1 -P yadavsms/python-micro-service
```

Lets see what happened in our Registrator container on the same host:

```
$docker logs 5ad2c811770b
2016/05/13 14:35:15 Starting registrator v7 ...
2016/05/13 14:35:15 Using consul adapter: consul://172.16.20.132:8500
2016/05/13 14:35:15 Connecting to backend (0/0)
2016/05/13 14:35:15 consul: current leader  172.16.20.132:8300
2016/05/13 14:35:15 Listening for Docker events ...
2016/05/13 14:35:15 Syncing services on 2 containers
2016/05/13 14:35:15 ignored: 5ad2c811770b no published ports
2016/05/13 15:32:46 added: c09f762d3283 registrator:service1:5000
2016/05/13 15:43:48 added: d628843108c9 registrator:service2:5000
```

Registrator saw that a new container (service1) was started, exposing port 5000 and it automatically registered it with our Consul cluster. We'll query our cluster now to see if the service was added there:

```
$curl $HOST_IP:8500/v1/catalog/services | python -mjson.tool
{
    "consul": [],
    "consul-53": [
        "udp"
    ],
    "consul-8300": [],
    "consul-8301": [
        "udp"
    ],
    "consul-8302": [
        "udp"
    ],
    "consul-8400": [],
    "consul-8500": [],
    "python-micro-service": []
}
```

Let's do the same for Node2 also.

Whenever we have a Consul cluster running we can query any node (client or server) and the response should always be the same. Since we are running our microservice containers in Node1 and Node2, let's query Consul on Node3:

```
$curl $HOST_IP:8500/v1/catalog/service/python-micro-service | python -mjson.tool
    {
        "Address": "172.16.20.94",
        "Node": "node3",
        "ServiceAddress": "",
        "ServiceID": "registrator:service2:5000",
        "ServiceName": "python-micro-service",
        "ServicePort": 32768,
        "ServiceTags": null
    },
    {
        "Address": "172.16.20.132",
        "Node": "node1",
        "ServiceAddress": "",
        "ServiceID": "registrator:service2:5000",
        "ServiceName": "python-micro-service",
        "ServicePort": 32770,
        "ServiceTags": null
    },
    {
        "Address": "172.16.20.184",
        "Node": "node2",
        "ServiceAddress": "",
        "ServiceID": "registrator:service2:5000",
        "ServiceName": "python-micro-service",
        "ServicePort": 32769,
        "ServiceTags": null
    }
```

Verifying the Microservice with Consul's DNS

Let's try one more thing - use Consul's DNS interface from a different container to ping our service. We'll run a simple Ubuntu container on Node3:

```
$docker run --dns 172.17.0.1 --dns 8.8.8.8 --dns-search service.consul --rm --name pin
g_test -it ubuntu
```

The "--dns" parameter allows us to use a custom DNS server for our container. By default the container will use the same DNS servers as its host. In our case we want it to use the docker bridge interface (172.17.0.1) first and then, if it can not find the host, then to go to 8.8.8.8. Finally, the "dns-search" option makes it easier to query our services. For example, instead of querying for "python-micro-service.service.consul" we can just query for "python-micro-service". Let's try to ping our service from the new Ubuntu container:

```
root@00a70eff5a2e:/# ping -qc 1 python-micro-service
PING python-micro-service.service.consul (172.16.20.184) 56(84) bytes of data.

--- python-micro-service.service.consul ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.430/0.430/0.430/0.000 ms

root@00a70eff5a2e:/# ping -qc 1 python-micro-service
PING python-micro-service.service.consul (172.16.20.94) 56(84) bytes of data.

--- python-micro-service.service.consul ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.056/0.056/0.056/0.000 ms
root@00a70eff5a2e:/# ping -qc 1 python-micro-service
PING python-micro-service.service.consul (172.16.20.132) 56(84) bytes of data.

--- python-micro-service.service.consul ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.481/0.481/0.481/0.000 ms
```

Removing Services

If we stop a running container, Registrator will notice it and also remove the service from Consul. We can see this by stopping the container running in Node1:

```
$docker stop d628843108c9
```

From the logs on Registrator we can see that it will remove the service from Consul

```
$docker logs 5ad2c811770b
2016/05/13 14:35:15 Starting registrator v7 ...
2016/05/13 14:35:15 Using consul adapter: consul://172.16.20.132:8500
2016/05/13 14:35:15 Connecting to backend (0/0)
2016/05/13 14:35:15 consul: current leader  172.16.20.132:8300
2016/05/13 14:35:15 Listening for Docker events ...
2016/05/13 14:35:15 Syncing services on 2 containers
2016/05/13 14:35:15 ignored: 5ad2c811770b no published ports
2016/05/13 15:32:46 added: c09f762d3283 registrator:service1:5000
2016/05/13 15:53:08 removed: d628843108c9 registrator:service2:5000
```

Now we are left with two instance of the service.

```
$curl $HOST_IP:8500/v1/catalog/service/python-micro-service | python -mjson.tool
    {
        "Address": "172.16.20.94",
        "Node": "node3",
        "ServiceAddress": "",
        "ServiceID": "registrator:service2:5000",
        "ServiceName": "python-micro-service",
        "ServicePort": 32768,
        "ServiceTags": null
    },
    {
        "Address": "172.16.20.184",
        "Node": "node2",
        "ServiceAddress": "",
        "ServiceID": "registrator:service2:5000",
        "ServiceName": "python-micro-service",
        "ServicePort": 32769,
        "ServiceTags": null
    }
```

Outage Recovery

The sub-sections describes various outage scenarios and how each can be managed to minimize downtime.

What is an Outage?

An outage is a loss of quorum for Consul, i.e., the nodes are not able to elect a leader. This might be caused by a majority of nodes going offline or not being able to communicate due to network partitions. We'll deal primarily with nodes going offline here.

For a quorum to be healthy it is necessary for every node to know its neighbors. The list of neighbors is kept in the Raft peers list in the consul-data-directory/raft/peers.json file, where consul-data-directory is specified when Consul starts (see below). There should be at least 2 nodes there in order to form a quorum and elect a leader on startup.

Outage recovery might sometimes involve manual editing of the peers list where the machines are not recoverable.

Key players:

- Raft manages leadership and its consistency.
- Serf manages membership like node joining/leaving.
- The "peer set" is the set of all members participating in log replication. For Consul's purposes, all server nodes are in the peer set of the local datacenter.

Data Directory

This is where each Consul agent stores state. It is required for all agents. The directory should be durable across reboots. This is especially critical for agents that are running in server mode as they persist cluster state in this directory.

The data directory contains raft/peers.json, serf/local.snapshot & serf/remote.snapshot. The raft directory contains a list of known servers of the cluster and serf contains the list of connected servers and clients like alive/not-alive state.

During an outage you can find out the state of your cluster is in by looking through your Serf and Raft peers lists.

raft/peers.json contains currently known Raft peers set (nodes that take part in the quorum and consensus). For a 3-node cluster the known raft/peers.json will look like:

```
root@server1:~# cat /var/consul/raft/peers.json
["192.168.1.12:8300","192.168.1.11:8300","192.168.1.13:8300"]
```

serf/*.snapshot: contains record of serf protocol progress and also which events took place over time.

Parameters

Parameters that affect the shutdown behavior of nodes:

You can either allow a Consul node to auto-leave the quorum on shutdown, or you can manually remove the node from the quorum with the leave or force-leave commands (which have to be run on the node you wish to remove).

There are two parameters that affect the behavior of nodes in a cluster on shutdown and restart - both of which have “leave” in them.

```
“leave_on_terminate” - false by default. Setting this to true will remove the node fro
m the cluster when it is terminated.
“skip_leave_on_interrupt” - false by default. Setting this to true will not remove the
 node from the cluster when it is interrupted.(see more on signals).
```

A Consul node can be stopped via Unix signals. The behavior depends on how the nodes are notified i.e., which signal is passed to Consul.

Consul Responses to Signals

```
SIGINT - Graceful leave by default.
SIGTERM - Non graceful leave by default.
```

Common kill signals and their impact on a running Consul process

Signal name	Signal value	Effect
SIGHUP	1	Reloads Consul’s service configuration (some services like Checks, Services, Watches, HTTP Client Address and Log Levels)
SIGINT	2	Usually generated by a Ctrl-c, graceful shutdown by default
SIGKILL	9	Non-graceful shutdown by default
SIGTERM	15	Non-graceful shutdown by default

We’ll explore combinations of signals and configurations and see how Consul behaves:

leave_on_terminate false & skip_leave_on_interrupt false.

- On SIGTERM signal known peers list will not be affected.
- On SIGINT(ctrl+c) signal known peers list will become null and the node will be deregistered from remaining nodes.

leave_on_terminate true & skip_leave_on_interrupt to false.

- On SIGTERM signal known peers list will become null and the node will be deregistered from remaining nodes.
- On SIGINT(ctrl+c) signal known peers list will become null and the node will be deregistered from remaining nodes.

leave_on_terminate true & skip_leave_on_interrupt to true.

- On SIGTERM signal known peers list will become null and the node will be deregistered from remaining nodes.
- On SIGINT(ctrl+c) signal known peers list will not be affected.

leave_on_terminate true & skip_leave_on_interrupt to true.

- On SIGTERM signal known peers list will not be affected.
- On SIGINT(ctrl+c) signal known peers list will not be affected.

Note that in case of a node failure (power failure or Consul kill signal) the known peers list will not affect.

Failure & Recovery Simulation of a Node in a Multi-server Cluster

We'll simulate the failure of a single node via various signals and its recovery, and its impact on the cluster state.

These examples assume the following config.json file in each of the nodes (3 nodes in this example)

```
{
"advertise_addr": "192.168.1.11",
"datacenter": "dc1",
"data_dir": "/var/consul",
"encrypt": "EXz7LFN8hpQ4id8EDYiFoQ==",
"node_name": "ConsulServer1",
"server": true,
"bootstrap_expect": 3,
"leave_on_terminate": true,
"skip_leave_on_interrupt": false,
"rejoin_after_leave": true,
"retry_join": [
    "192.168.1.11:8301",
    "192.168.1.12:8301",
    "192.168.1.13:8301"
 ]
}
```

Let's verify that we have a quorum and the leader present

```
root@server1:~#curl -v http://127.0.0.1:8500/v1/status/leader
* Trying 127.0.0.1…
* Connected to 127.0.0.1 (127.0.0.1) port 8500 (#0)
> GET /v1/status/leader HTTP/1.1
> Host: 127.0.0.1:8500
> User-Agent: curl/7.47.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< Date: Mon, 24 May 2016 11:08:22 GMT
< Content-Length: 19
< * Connection #0 to host 127.0.0.1 left intact
"192.168.1.11:8300"root@server1:~#
```

We also verify that they see each other by checking the Serf peers list

```
root@server1:~# consul members -detailed
```

Node	Address	Status	
ConsulServer1	192.168.1.11:8301	alive	build=0.6.4:26a0ef8c,dc=dc1,expect=3
ConsulServer2	192.168.1.12:8301	alive	build=0.6.4:26a0ef8c,dc=dc1,expect=3
ConsulServer3	192.168.1.13:8301	alive	build=0.6.4:26a0ef8c,dc=dc1,expect=3

Let's also verify that the Raft peers contains the right set of nodes across every Consul server that should form the quorum, i.e. no left, failed or unreachable servers are present.

```
root@server1:~# cat /var/consul/raft/peers.json
["192.168.1.12:8300","192.168.1.11:8300","192.168.1.13:8300"]
```

SIGINT: Now let us shut down any of the node. I am shutting down the existing leader, in my case it is 192.168.1.11 (server1.example.com).

Ctrl+c to the running Consul

Server1.example.com

```
==> Caught signal: interrupt
==> Gracefully shutting down agent…
[INFO] consul: server     starting leave
[INFO] raft: Removed peer 192.168.1.12:8300, stopping replication (Index: 739)
[INFO] raft: Removed peer 192.168.1.13:8300, stopping replication (Index: 739)
[INFO] raft: Removed ourself, transitioning to follower
[INFO] raft: Node at 192.168.1.11:8300 [Follower] entering Follower state
[INFO] consul: cluster leadership lost
[INFO] raft: aborting pipeline replication to peer 192.168.1.12:8300
[INFO] raft: aborting pipeline replication to peer 192.168.1.13:8300
[INFO] serf: EventMemberLeave: ConsulServer1.dc1 192.168.1.11
[INFO] consul: removing WAN server ConsulServer1.dc1 (Addr: 192.168.1.11:8300) (DC: dc
1)
[INFO] serf: EventMemberLeave: ConsulServer1 192.168.1.11
[INFO] consul: removing LAN server ConsulServer1 (Addr: 192.168.1.11:8300) (DC: dc1)
[INFO] agent: requesting shutdown
[INFO] consul: shutting down server
[INFO] agent: shutdown complete
```

Now the peers.json file will become null in server1.example.com

```
root@server1:~# cat /var/consul/raft/peers.json
null
```

192.168.1.11 should be deregistered from server2.example.com and server3.example.com root@server2:~# cat /var/consul/raft/peers.json ["192.168.1.12:8300","192.168.1.13:8300"]

```
root@server3:~# cat /var/consul/raft/peers.json
["192.168.1.12:8300","192.168.1.13:8300"]
```

Now the new leader will be elected from the running server2.example.com(192.168.1.12) & server3.example.com(192.168.1.13)

In my case server2.example.com(192.168.1.12) is elected as the new leader.

```
[INFO] consul: removing LAN server ConsulServer1 (Addr: 192.168.1.11:8300) (DC: dc1)
[ERR] agent: failed to sync remote state: No cluster leader
[WARN] raft: Heartbeat timeout reached, starting election
[INFO] raft: Node at 192.168.1.12:8300 [Candidate] entering Candidate state
[INFO] raft: Duplicate RequestVote for same term: 36
[WARN] raft: Election timeout reached, restarting election
[INFO] raft: Node at 192.168.1.12:8300 [Candidate] entering Candidate state
[INFO] raft: Election won. Tally: 2
[INFO] raft: Node at 192.168.1.12:8300 [Leader] entering Leader state
[INFO] consul: cluster leadership acquired
[INFO] consul: New leader elected: ConsulServer2
[INFO] raft: pipelining replication to peer 192.168.1.13:8300
[INFO] consul: member 'ConsulServer1' left, deregistering
```

Let’s verify the Consul members

```
root@server2:~# consul members
```

Node	Address	Status	Type	Build	Protocol	DC
ConsulServer1	192.168.1.11:8301	left	server	0.6.4	2	dc1
ConsulServer2	192.168.1.12:8301	alive	server	0.6.4	2	dc1
ConsulServer3	192.168.1.13:8301	alive	server	0.6.4	2	dc1

If the node which left the cluster can be brought up again, we can just restart it and have it rejoin the cluster, returning the cluster to a fully healthy state.

```
root@server1:~# consul agent -config-dir /etc/consul.d/config
```

If any of the node's managed to perform a graceful leave, you may need to have then rejoin the cluster using the join command (whether they restarted or not).

```
root@server1:~# consul join <Node Address>
Successfully joined cluster by contacting 1 nodes.
```

It should be noted that any existing member can be used to issue the join command as the gossip protocol will take care of discovering the server nodes. At this point, the quorum is healthy.

In our case with the above config.json we can simply start the server1.example.com and it will rejoin the cluster because "rejoin_after_leave" was set to true and retry_join keys were provided.

Now we have a quorum with leader and 2 followers.

```
root@server2:~# consul members
```

Node	Address	Status	Type	Build	Protocol	DC
ConsulServer1	192.168.1.11:8301	alive	server	0.6.4	2	dc1
ConsulServer2	192.168.1.12:8301	alive	server	0.6.4	2	dc1
ConsulServer3	192.168.1.13:8301	alive	server	0.6.4	2	dc1

SIGTERM: Now let us kill any of the node. We choose the existing leader (kill)

```
root@server1:~# curl http://127.0.0.1:8500/v1/status/leader
"192.168.1.12:8300"
```

In our case the leader is 192.168.1.12(server2.example.com).

The peers.json file in the killed node will become null.

```
root@server2:~# cat /var/consul/raft/peers.json
null
```

A new leader is elected - ConsulServer1 in this example

```
root@server1:~# curl http://127.0.0.1:8500/v1/status/leader
"192.168.1.11:8300"
```

If the failed server2.example.com is recoverable the easiest option is to bring it back online and have it rejoin the cluster, returning the cluster to a fully healthy state.

Now let’s start Consul:

```
root@server2:~# consul agent -config-dir /etc/consul.d/config
…
[INFO] agent: Joining cluster…
…
[INFO] serf: Re-joined to previously known node: ....
…
[INFO] agent: Synced service 'consul'
```

SIGKILL: In case the node has crashed or sigkill (kill -9 pid) was used, or we are unable to bring up node, we need to rebuild a new Consul server to replace it. Keep in mind that the rebuilt node needs to have the same IP as the failed node because the killed node's IP will be in the other nodes peers.json file. Again, once this node is online, the cluster will return to a fully healthy state provided there should be a leader present that means for bringing up the failed node the other 2 nodes should be in the running state.

If building a new node with the same IP isn't an option, you need to remove the failed node's IP from raft/peers.json file on all remaining nodes.

Now we will see how the failed/unrecoverable node will be removed from the raft/peers.json file on all remaining nodes.

Stop all remaining nodes. You can attempt a graceful leave or issue a sigterm.

Go to data directory, in this example it is under /var/consul of each Consul node. We need to edit the raft/peers.json file. It should look something like:

```
root@server1:~# cat /var/consul/raft/peers.json
["192.168.1.12:8300","192.168.1.11:8300","192.168.1.13:8300"]
```

Delete the entries for all the failed nodes. We should confirm those servers have indeed failed and will not later rejoin the cluster. We also need to ensure that this file is same across all remaining nodes. It is not enough to just get rid of failed nodes, we also have to make sure that all other healthy nodes are in there. Without it a quorum will not form.

Now if the stopped nodes are successfully stopped then the healthy node's data directory raft/peers.json will be null.

```
root@server1:~# cat /var/consul/raft/peers.json
null
```

We need to remove the failed node's entry from the existing nodes raft/peers.json file before we add the new node. This is not mandatory but if it's not done the old nodes will keep attempting to reach the failed one.

In this example, the new node details are server4.example.com with IP 192.168.1.14.

Now let's start the old nodes with modified raft/peers.json files. After they are started let's start the new node and join the existing nodes.

Now all the Consul servers peers.json files will look like:

```
root@server1:~# cat /var/consul/raft/peers.json
["192.168.1.14:8300","192.168.1.12:8300","192.168.1.11:8300"]
```

At this point, the cluster should be in an operable state again. One of the nodes should claim leadership and emit the log event:

```
[INFO] consul: cluster leadership acquired
```

We should also verify at this point that one node is the leader and all the others are in the follower state. All the nodes should agree on the peer count as well. This count is (N-1), since a leader does not count itself as a peer.

Summary

- When there are no nodes in the peers.json file, the nodes fail to form a quorum.
- If we wish to remove the failed node's IP from the alive nodes, we need to restart all the alive nodes after editing the peers.json file.
- With "leave_on_terminate"=false and "skip_leave_on_interrupt"=true configurations set, the peers are not removed from the peers.json.

Reference: https://www.consul.io/docs/guides/outage.html

Deployment Architectures

Using Consul for Key Value Configuration

TBD

Using Consul for Service Discovery

TBD

Running Consul on AWS

To be done