Experimenting with serf *and* etcd

Bhihe, C. K., Flamand, B.

Facultat d'Informàtica de Barcelona, Universitat Politècnica de Catalunya, Barcelona 08034

Abstract

Microservices is one of the significant trends in distributed computing, whereby applications scale horizontally, leveraging computing resources across multiple nodes. To manage nodes organized in clusters, and the services depending on them, one requirement is node orchestration. Orchestration minimally includes health monitoring (failure detection) of cluster nodes, membership and service discovery, as well as configuration. In this work, we set out to scrutinize two technologies, instrumental in making modern node orchestration possible; serf, a part of Consul developed by Hashicorp and etcd, based on the RAFT consensus algorithm and developed by CoreOS. We first describe those two technologies and compare them. Next we describe in detail the experimental setup used in running containers containing either one serf agent (node) or one etcd clients (node). We aimed at measuring communication overhead in both cases, a goal never fulfilled due to persistent technical difficulties in running clusters of containerized nodes on either AWS instances or on local physical machines. We close the report with a perspective on future work based on our limited state of progress.

Note: This report is supported by learning material (tutorials), prepared to help future students overcome the many difficulties we encountered.

1. Introduction

One of today's salient computing trends consists in striking a balance between a full microservice-oriented horizontal scaling approach and the more traditional monolithic application-centric vertical scaling model. That is the goal pursued by application designers, interested in (I) achieving faster devops cycles, including testing, (ii) building composed services more easily, (iii) isolating and remedying failures more efficiently, (iv) deploying apps and rolling out updates without compromising service availability and process liveness.

In the following we will scrutinize two technologies instrumental in orchestrating cluster nodes for distributed applications: etcd and serf. In sections 2 and 3, we concentrate on a description and a comparison of serf and etcd. Section 4 delimits the goals of this work and section 5 describes how we set out to reach them, via experiments involving virtualization and the isolation of cluster nodes inside Docker containers. Section 6 concludes by reasserting the initial objectives of this work and by proposing new avenues for testing.

2. Cluster node management in the context of distributed applications

2.1 – *The trend toward and from microservices*

On one hand microservices [1] are based on the idea that each unit-functionality in an application should be coded as an independent application component. Those components are maintained separately and relate to one another by means of a communication protocol over the network. This guarantees an acceptable level of availability and fault-tolerance through distributed replication. However a full blown microservice-based architecture incurs an overhead due to what is generically referred to as n-tier inter-service communication requirements.

On the other hand a large monolithic application will require a large number of APIs, designed to render multiple application functions. Those APIs may support a variety of backends (e.g. RDBMS, noSQL, cache), and possibly different presentation models, such as desktop and mobile clients. This leads to a significant degree of added complexity, overhead for developers to update a large code base and for IT Operations to deploy any new version.

In the case of microservices scaling *out* is the rule, while in the latter case, scaling *up* becomes the *de facto* paradigm with its obvious physical limits and drawbacks in terms of fault tolerance and server centralization. The two approaches represent the two opposite ends in a continuous application design spectrum and each shows design-specific performance penalties. *Service oriented Architecture* (SoA) aims at resolving that application design dilemma. It is based on three main principles:

- to make each microservice as autonomous as possible,
- to define and limit the scope of each microservice, (perhaps also to balance their scope)
- to couple microservices loosely to maximize fault-tolerance and liveness without compromising correctness.

Together those three principles serve the common goals of:

- achieving faster devops cycles, including testing,
- building composed services more easily,
- isolating failure and remedying it more efficiently,
- being able to balance scale-out (horizontal scaling) and scale-up (vertical scaling).

2.2 – *Service discovery, health monitoring, configuration and orchestration*

The challenges associated with SoA are :

- discovering services and their corresponding hosts or network nodes,
- real-time monitoring of the health status of participating hosts,
- configuring services or re-configuring them rapidly and reliably as a reaction to change,
- orchestrating services according to different events in different locations.

At its core, *service discovery* is about (i) knowing which process in a cluster listen on a TCP or UDP port, (ii) being able to look up and connect to that port by name. In practice, *discovery* could be dispensed with if hard-coding addresses were an option. Unfortunately hard-coding addresses is prone to failure and hard to scale in case more than one host needs to be pointed at. For multiple service-providing hosts, a load balancer may handle *discovery* and *health monitoring* at once introducing a level of indirection between the client(s) and the service-providing host. However a load balancer introduces a fundamental weakness as it constitutes a single point of failure. *DNS-SD* [2] is a well known example of examples of how to achieve service discovery and *ZooKeeper*[i] [3],[4] a distributed platform's coordinator that also provides *service discovery*. Still, those management tools tend to operate as somewhat isolated data receptacles, in that they do not interact with other services on a time scale compatible with the execution of most distributed system services.

Orthogonal to service discovery is the issue of *configuration*. Configuration becomes difficult when more than a few machines find themselves in slightly different states. Beyond the use of static configuration files, dedicated management tools abstract resources and address configuration (as well as provisioning and app deployment) in different ways:

- Chef [5] and Puppet [6], are based on Erlang/Ruby and Ruby respectively and use a server/client agent model distributed among cluster nodes. Only Puppet is open-source.
- Ansible [7] and Salt [8], are two open-source, command-line-interface (cli) configuration tools, based on Python and Yaml respectively. They resort to SSH for communication between nodes.

Every solution above monitors the *health-status* of participating nodes, based on a push model, whereby a heartbeat is sent from each nodes to a listening super-node at regular intervals, followed by a wait by the listening node. In each case, either ease of configuration, fault tolerance, speed or scalability are problematic to varying extents.

Orchestration comes into play when for instance updates are needed. Cached sessions need to be purged, session encryption keys rotated and sessions or prefixes invalidated in a way suitable to a cluster of distributed hosts. *Parallel SSH* and again *ZooKeeper* may offer solutions (on different scales), but again not entirely satisfactory for the reasons already cited above.

We now broadly review a few technological responses to the above challenges. We highlight some pros and cons in each case.

i ZooKeeper is the open-source equivalent of Google's *Chubby*. It spun out of the Apache Hadoop project and became the de facto standard lock-server in the open-source world. It is now a dependency to many major projects that require *distributed coordination*, including Storm, Mesos, Kafka, and others.

3. A comparison between s*erf* and *etcd*

3.1 – *Serf*

Designed by HashiCorp, *Serf* is a basic component of *Consul* [9], a distributed highly-available, and multi-datacenter aware tool for service discovery, configuration, and orchestration for clusters. Consul enables rapid deployment, configuration, and maintenance of large scale service-oriented architectures. *Serf* [10] with a fully decentralized architecture, provides Consul with a *gossip* [11] protocol designed for fast communication and failure detection. Gossip relies on a *pull model* from one node to a randomized set of neighboring nodes, whereby the sending node issues ssh requests to nodes in P2P mode. The alternative is the more traditional *push model*. Gossip is an epidemic transmission model, which imposes a constant load on the cluster irrespectively of how many nodes are in it. It provides a relatively small feature set, including membership, failure detection, and propagation of user driven events. Events are usually remote execution instructions, such as update deployments, rolling restarts, and so on. They can be propagated to any designated set or subset of nodes inside any number of clusters. Target nodes are identified by the service(s) they run, and events consist of a name and a payload. Each receiving node is equipped with an ad-hoc handler (typically a shell script) that will then act upon receiving event notification. Designed to operate under conditions of network partition*s*, *Serf* embraces eventual consistency. It is friendly to deploy for system administrators [12] and to use for application developers and network operators [13] alike. It is also designed to propagate information quickly. Once configured, *Serf* can be used to handle changes to the cluster and update configuration files quickly, instead of relying on convergence runs. This way, a web server can join a cluster made of several thousand nodes in mere seconds instead of minutes before its membership is consensually accepted.

In practice, a *Serf* client agent must run on every member-node of a *Serf* cluster. A cluster also runs anywhere from 3 to 5 server agents (depending on the degree of availability required), one being a transparently elected leader while others act as replicas. Server agents form a consensus pool, so their centralized view of cluster state is consistently replicated among them. Replicated server agents have identical functionality as their elected leader to avoid situations where single points of failure may interrupt service. Typically server agents receive remote procedure calls (RPCs[i]) from client agents, forward them to the leading server agent, which act on them, e.g. to contact another cluster's leader-elect server agent, perhaps for the purpose of multi-cluster coordination.

Every agent sports a set of exposed APIs to handle RPCs, HTTP and DNS traffic. Every one of them is also responsible for running local health checks, event execution, etc, and is part of at least one gossip pool(s), based on the UDP unicast protocol[ii].

Every server agent has the additional responsibility of holding centralized cluster state information, for instance about node membership. This makes leaving the cluster a straightforward proposition for any node. Recovering from a transient fault or partition becomes equally easy as a server will repeatedly send pull requests to nodes reported to be unhealthy, until a response is obtained.

As a lightweight process that maintains memberships and guarantees communication between nodes, *Serf* makes it possible to run as many agents in one node as the node has memberships in different co-existing *Serf* clusters. For example a node may be a member of a cluster that maintain web server membership information for a load balancer from another Serf cluster. Simultaneously it can be a member of a different gossip cluster network with different time constraints. Multi-cluster topology may overlap without problems. Inter-cluster communication behaves according to the same gossip protocol already described for intra-cluster communication between client-agents. In the case of inter-cluster communication however, communicating nodes are preferably the leaders-elect of every partaking cluster or their replicas in the event of a leader-elect's failure.

i An RPC is a client-server based protocol used by a routine (client) to request a service from a program (server) located in another host and reachable via a network infrastructure. Several RPC models and distributed computing implementations exists. RPC server programs typically use dynamic port mappings and end-point-mapping (EPM) to avoid conflicts with programs and protocols registered in the range of well-known TCP ports. It is usual for certain port numbers to be associated with given services, e.g.: UDP/TCP 80, 443, 593 for HTTP; UDP/TCP 445 for named pipes; UDP/TCP 135 for EPM.

ii DNS primarily uses UDP on port 53 to serve requests. DNS queries consist of a single UDP request from the client followed by a single UDP reply from the server. TCP is used, for instance when the response data size exceeds 512 bytes, or for tasks such as zone transfers, i.e. DNS database replication. Some resolver implementations use TCP for all queries.

3.2 – *Etcd*

ZooKeeper and etcd [14] are somewhat similar from the point of view of their client/server (or slave/master) architecture. By contrast they are markedly different from Serf and somewhat more complex to operate. Spun off the Hadoop project in 2010 , ZooKeeper exhibits high availability and reliability. Its distributed system coordination semantics is subtly different from that of the Paxos algorithm [15]–[17] in Chubby [18]. Being a consistent and highly available key-value store, ZooKeeper is sometimes described as a cluster state (or cluster configuration) store and a service directory. But it can also be complex to implement and rather unwieldy in its deployment. Largely motivated by that fact, a simpler alternative to Paxos, called Raft [19], was announced in 2014. Built around Raft, etcd was soon introduced by CoreOS [20], as a daemon running on every node of a cluster, to solve the uncoordinated upgrade availability issue of cluster-nodes. It does so by relying on a reboot-lock service. The reboot-lock service relies in turn on a shared dynamic registry, allowing cluster configuration data to be reliably distributed and replicated among cluster nodes. It addresses the issue of persistently storing cluster state (i.e. up-to-date slaves' memberships and process status information) in a manner that is distributed, fault-tolerant, available and simpler than that proposed by ZooKeeper. Etcd also handles leader election at times of either network partition or current leader's failure. Written in Go, relies on JSON / HTTP for communication via a RESTful API.

Interestingly Consul, already mentioned in the previous section on Serf, was built based on some of the principles of etcd. Both Consul (serf) and etcd advertise themselves as service discovery solutions. Consequently etcd offers a solution to service discovery, which comprise (a) a consistent, highly available service directory, (b) a mechanism to register services and monitor service *health, (c) a* mechanism to look up and connect to services.

Although related libraries of etcd primitives exist as part of an API [21] for common instruction patterns, application developers may still have to build the specific primitives they need to implement ad-hoc mechanisms. In that case the required work scales linearly with the number of nodes and places demands on a fixed number of servers, i.e. a further departure from the lightweight P2P communication paradigm championed by *Serf.*

Furthermore most failure detection schemes built on etcd also have intrinsic scalability issues. Health checks depend on heartbeats, which, as already mentioned, use periodic updates and timeouts. As a result the failure detection window is at least as long as the timeout, meaning that in many cases failures may be confused with a liveness issue and may go undiagnosed for a significant lengths of time.

Serf and either Z*ooKeeper* or *etcd* are not mutually exclusive. The latter two are strong consistency-based orchestrators, which can be complemented by *Serf* to create systems that are fully replicated, more scalable, highly available and watchable. High availability is the upshot of fault-tolerance, and can be interpreted as resilience to single points of failure and to network partition. In the case of *etcd* it must be understood as "high availability for read operations".

The overarching goal is to design a mechanism where events (e.g. updates, roll-outs) can be reliably pushed from the Key-Value (KV) store to distributed applications, in a timely manner, and without sacrificing features.

In the rest of this work, we restrict our study to the comparison of *serf* (v0.8) and *etcd* (v3.0) in the context of a cluster of containers running either serf agents [22] or etcd clients [23].

4. Goals of this work

Three related goals drove our effort in relation with the study of serf and etcd cluster dynamics.

4.1 – *Assessment of communication performance between nodes*
- time for event propagation between two specific nodes with varying number of hops in the cluster mesh,
- time for generalized event propagation (time to full completion of propagation), for varying cluster connectivity values (number of nearest neighbors averaged over all the cluster nodes)
- time for generalized event propagation (time to full completion of propagation),
- repeat the above in the event of node failure (unannounced communication break-off of random nodes),

- repeat the above in the event of node graceful leave (announced communication break-off of random node),
- probe intra-cluster and inter-cluster communication performance for key-value of varying length (16B or 256kB) and in situations of congestion

4.2 – *Implement clusters of standalone versions of serf agents and etcd clients in Docker containers on AWS*
Containers wrap up a piece of software (application) in an RTE (run time environment) that contains the minimum set of required resources needed to run: code, RTE, system tools, system libraries. This guarantees that the application will always run the same, regardless of the environment of the platform on which the container executes. Containers are generated from a simple text recipe that can be easily passed on and versioned. They leverage current operating system capabilities for virtualizing processes and name spaces with little overhead. Container technology allows us to quickly turn recipes into runnable applications, and then deploy them anywhere.

Our choice to build a cluster of either serf or etcd in Docker containers was motivated by the popularity of Docker, CoreOS, and related systems in enterprise business applications, as a response to demand for lightweight and portable container solutions.

5. Experimental setup and first trials

What follows departs voluntarily from what an article meant for publication would contain in that we provide details clearly unnecessary for publication.

We aim at launching clusters of either etcd or serf nodes. Each node is isolated in its Docker container, and represent a containerized service in a cluster, i.e. a pool of compute, storage, and networking resources. We decided to deploy Docker on virtual machines by leveraging AWS-EC2 cloud resources. We availed the free tier conditions offered by Amazon from the AWS Educate Starter Account 75[i].

The AWS EC2 instances were t2.micro virtual servers based on either Ubuntu 14.04 or 16.04 (64bits), in the cloud compute-region *us-west-2*. Those instances are caracterized by 1 CPU and 1 GB of RAM each and caps on iops. A running instance utilizes approximately 350MB of RAM, the rest being available to run docker containers as serf agent nodes or etcd client nodes. They qualify for the *AWS Free Tier*, and provide a small amount of consistent CPU resources, They permit the increase of CPU capacity in short bursts, when load spikes must be absorbed up to 200% CPU capacity. They are well suited for lower throughput applications, which don't use the full CPU often or consistently, but occasionally need to absorb bursts of activity. The detailed requirements and limitations of t2-micro EC2-PVC instances are well documented on AWS[ii].

5.1 – *Leveraging AWS -EC2 from an AWS Educate Account*

After registering[iii] for an AWS Educate Account, run the “AWS Educate Starter Account 75” lab session. In doing so, note that you obtain an *Access Key ID* and a *Secret Access Key*, both displayed under the “Open Console” button. Those keys will be needed later to manage instances’ environment. Do not close the corresponding tab and make a note of the key-values. Hit the “Open Console” button to access the AWS Services Console home page.

On the AWS Console we had a choice between 2 services: ECS or EC2 . ECS (EC2 Container Service) essentially offer a pre-installed Docker engine running on an EC2 instance of your choice, while EC2 (Elastic Cloud Computing) is the bare Virtual Private Server (VPS). ECS is advertized by AWS as an easier to run instance. However because we ran into difficulties trying to make several instances in the same AWS region launch and communicate with specific Security Group (IPTables), we opted for EC2 instances on which we installed the latest version of the Docker engine.

We fully automated the launch and remote SSH connection to AWS-EC2 instances by writing a script that gathers connection and AWS session meta-data as well as basic metrics on launch time and time to reachability (network

i https://awseducate.qwiklabs.com: Review by AWS of applications for a student account takes 48 to 96 hours to complete. Registration is only possible with a an academic email address.
ii t2-micro instance characteristics and requirements: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-instances.html
iii Never end the lab session, unless you re done experimenting. It would make the specific lab session expire forever.

bootstrap time) of the instance. The script (*`initInst.sh`*) is fully available on GitHub[i] as well as in the accompanying documentation.

Our instances consisted of a custom made HVM-AMI[ii], based on:

- t2.micro Ubuntu 14.04.5 VPS
- Docker Engine v17.05 (Docker's old version numbering scheme stopped at v1.12)
- permission configurations for user "ubuntu" and group "docker"
- installation of two standard Debian communication packages: bridge-utils, iputils-ping
- simplified custom security group rules allowing SSH over TCP from 0.0.0.0/0 over port 22 and incoming ICMP traffic from 0.0.0.0/0. This was later expanded with additional rule to allow incoming TCP traffic from specific IPs over all ports.
- an additional script (*`initDC.sh`*) designed to automate the launch of containers with either serf or etcd agents from within a running EC2 instance. As before that script is made available on GitHub.

5.2 – *Launching containers with serf agents or etcd clients*

The aforementioned script designed to automate the launch of container images accepts any DockerHub or custom-made local container image. We proceeded with trials using two container images for serf: (i) *harisekhon/serf* based on serf (v0.8), Alpine (v3.5) but devoid of command shell and therefore not meant to be run in interactive mode, and (ii) cbhihe/serf-alpine-bash, based on serf (v0.8.1), Alpine (v3.5) and bash (v4.3), which can be executed in interactive mode. The 2 Dockerfiles used to build those images are publicly available on DockerHub. The two built images weigh respectively 18 and 20 MB on disk and need less than 10MB allocation during in RAM execution with live nodes. The difference in weights is primarily due to the bash libraries mounted in the second custom image. Thus we planned to be able to run between 15 and 30 such containers in a single t2.micro VPS instance. Readers may test-run and download the freely available Dockerfiles and accompanying instructions for the two above mentioned images from Docker Hub.

The two scripts mentioned earlier are "Docker image-agnostic" in that they do not depend on the content of any chosen Docker image. They can be used to launch either serf or etcd nodes in strictly the same manner.

Only configuration files to setup a cluster will differ according to the primitives made available by both the serf API and the etcd API.

5.3 – *Trials*

A description of the launch trials for etcd clients in containers cannot be reported at this time (info unavailable).

For cluster of serf nodes, a very large number of attempts were made over the last 45 days. Launching any number of serf nodes in the following configuration was successful:

- Lone serf nodes outside or inside a Docker container, on an AWS instance or on a physical (local) host. Several agents on as many Docker containers could be made to run isolated from one another on a given host.

- Passing commands to live serf agents was functional, either from the host's environment, outside the node's container :

```
        ubuntu@ip-172-31-1-190:~$ docker –version
Docker version 17.05.0-ce, build 89658be

        ubuntu@ip-172-31-1-190:~$ docker exec container-id serf cmd [args]
```

or from inside the node's container:

```
        ubuntu@ip-172-31-1-190:~$ docker attach container-id
        /root/$ serf cmd [args] [&]
```

However we failed at building a cluster of nodes.
Below is a representative example with accompanying error messages.

i https://github.com/Cbhihe/serfDocker.git
ii HVM (Hardware Virtual Machine) – AMI (Amazon Machine Image): For more information see https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html

First, using i*`nitDC.sh`*, run 3 serf nodes in as many docker containers on one host (VPS on AWS in this example case):

```
        ubuntu@ip-172-31-1-190:~$ bash initDC.sh cbhihe/serf-alpine-bash 3
[...output…]

        ubuntu@ip-172-31-1-190:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND            CREATED          STATUS        PORTS
NAMES

28418296fb4d        serf-alpine-bash    "/bin/bash"             9 seconds ago       Up 8 seconds
0.0.0.0:32838->7373/tcp, 0.0.0.0:7003->7946/tcp, 0.0.0.0:32837->8443/tcp   serfDC3
6353e7f6134d        serf-alpine-bash    "/bin/bash"             3 minutes ago         Up 3 minutes
0.0.0.0:32823->7373/tcp, 0.0.0.0:7002->7946/tcp, 0.0.0.0:32821->8443/tcp   serfDC2
d4fd90692e18        serf-alpine-bash    "/bin/bash"             3 minutes ago         Up 3 minutes
0.0.0.0:32814->7373/tcp, 0.0.0.0:7001->7946/tcp, 0.0.0.0:32812->8443/tcp   serfDC1
```

Next visualize meta-data collected in file *`./setup.db`* by our script *`initDC.sh`*, in particular on virtual network interfaces for each containers.

```
        ubuntu@ip-172-31-1-190:~$ more setup.db
Date: 20170603-222017
AWS EC2 instance host name: 166.red-193-152-248.dynamicip.rima-tde.net.
Private instance IP: 172-31-1-190
Public instance IP: 35.163.173.104
DC-#    DC-id         DC-privIP    DC-veth      Node-id
1       28418296fb4d  172.17.0.2   vethfff73f9  28418296fb4d4b1c7ce642baecf94c1a7bdb428e15c7
6b5297a45b6768a29e56
2       6353e7f6134d  172.17.0.3   vethf2ece87  6353e7f6134dd56c40d032fcbd9a8f31544f7d91b2db
8364e21bf5c8d80a1d34
3       d4fd90692e18  172.17.0.4   veth198b30c  d4fd90692e18eb08b6b4eee3fbafb259a6878c034cb9
141f9bafaea60a8d0942
```

Veth interfaces are way to listen to inter-container traffic on the overlay bridge network *`docker0`*. For more information, refer to the accompanying documentation at section "Docker networks". For example to listen to pings from container serfDC2 to container serfDC3:

```
        ubuntu@ip-172-31-1-190:~$ docker exec 28418296fb4d ping 172.17.0.3 &> /dev/null &

        ubuntu@ip-172-31-1-190:~$ sudo tcpdump -i vethfff73f9 icmp
```

or equivalently:

```
        ubuntu@ip-172-31-1-190:~$ sudo tcpdump -ni docker0 host 172.17.0.2 and host 172.17.0.3
```

Note: The above listening technique was meant to be used in conjunction with raw traffic monitoring between serf agents or between etcd clients running in different containers. Mapping the virtual interfaces of Docker containers to their IP addresses and container IDs is not trivial. We describe one technique of discovery in script *`initDC.sh`*.

Connecting Docker containers on multiple hosts is in principle only slightly more complex as reported in a slightly dated blog post elsewhere [24]. Our first priority however is not to connect containers on distant hosts but to connect serf agents in different containers running on the same host.

Before going on to show where we failed to join serf agents in a cluster, we need to check that all 3 previous containers belonged to the same bridge network.

```
        ubuntu@ip-172-31-1-190:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
08d7c8cb69da        bridge              bridge              local
f52f6fa2db46        host                host                local
08413a11e83b        none                null                local

         ubuntu@ip-172-31-1-190:~$ docker network inspect bridge
[... check that all 3 containers serfDC{1,2,3} are listed in output JSON …]
```

Next, bind the serf agent in container serfDC1 to the port 7946 :

```
       ubuntu@ip-172-31-1-190:~$ docker exec serfDC1 serf agent -node=Node1 -bind=0.0.0.0:7946
==> Starting Serf agent…
==> Starting Serf agent RPC...
==> Serf agent running!
         Node name: 'd4fd90692e18'
         Bind addr: '0.0.0.0:7946'
          RPC addr: '127.0.0.1:7373'
         Encrypted: false
          Snapshot: false
           Profile: lan
==> Log data will now stream in as it occurs:
    2017/06/04 00:01:10 [INFO] agent: Serf agent starting
    2017/06/04 00:01:10 [INFO] serf: EventMemberJoin: d4fd90692e18 127.0.0.1
    2017/06/04 00:01:11 [INFO] agent: Received event: member-join
       ^C

       ubuntu@ip-172-31-1-190:~$ docker exec serfDC2 serf agent -node=Node2 -join=172.17.0.4
==> Starting Serf agent...
==> Starting Serf agent RPC...
==> Serf agent running!
         Node name: '6353e7f6134d'
         Bind addr: '0.0.0.0:7946'
          RPC addr: '127.0.0.1:7373'
         Encrypted: false
          Snapshot: false
           Profile: lan
==> Joining cluster...(replay: false)
    Join completed. Synced with 1 initial agents
==> Log data will now stream in as it occurs:
    2017/06/04 00:18:35 [INFO] agent: Serf agent starting
    2017/06/04 00:18:35 [INFO] serf: EventMemberJoin: 6353e7f6134d 127.0.0.1
    2017/06/04 00:18:35 [INFO] agent: joining: [172.17.0.4] replay: false
    2017/06/04 00:18:35 [INFO] serf: EventMemberJoin: d4fd90692e18 127.0.0.1
    2017/06/04 00:18:35 [INFO] agent: joined: 1 nodes
    2017/06/04 00:18:36 [WARN] memberlist: Got ping for unexpected node 'd4fd90692e18'
from=127.0.0.1:7946
    2017/06/04 00:18:36 [INFO] agent: Received event: member-join
    2017/06/04 00:18:37 [WARN] memberlist: Got ping for unexpected node d4fd90692e18
from=127.0.0.1:34876
    2017/06/04 00:18:37 [ERR] memberlist: Failed TCP fallback ping: EOF
    2017/06/04 00:18:37 [INFO] memberlist: Suspect d4fd90692e18 has failed, no acks received
    2017/06/04 00:18:38 [WARN] memberlist: Got ping for unexpected node 'd4fd90692e18'
from=127.0.0.1:7946
    2017/06/04 00:18:39 [WARN] memberlist: Got ping for unexpected node d4fd90692e18
from=127.0.0.1:34879
    2017/06/04 00:18:39 [ERR] memberlist: Failed TCP fallback ping: EOF
    2017/06/04 00:18:40 [INFO] memberlist: Suspect d4fd90692e18 has failed, no acks received
    2017/06/04 00:18:41 [WARN] memberlist: Got ping for unexpected node 'd4fd90692e18'
from=127.0.0.1:7946
    2017/06/04 00:18:42 [WARN] memberlist: Got ping for unexpected node d4fd90692e18
from=127.0.0.1:34881
    2017/06/04 00:18:42 [ERR] memberlist: Failed TCP fallback ping: EOF
    2017/06/04 00:18:42 [INFO] memberlist: Marking d4fd90692e18 as failed, suspect timeout reached
(0 peer confirmations)
    2017/06/04 00:18:42 [INFO] serf: EventMemberFailed: d4fd90692e18 127.0.0.1
    2017/06/04 00:18:43 [INFO] agent: Received event: member-failed
    2017/06/04 00:18:44 [INFO] memberlist: Suspect d4fd90692e18 has failed, no acks received
    2017/06/04 00:19:05 [INFO] serf: attempting reconnect to d4fd90692e18 127.0.0.1:7946
   ^C

        ubuntu@ip-172-31-1-190:~$ docker exec serfDC2 serf members
6353e7f6134d  127.0.0.1:7946  alive
d4fd90692e18  127.0.0.1:7946  failed
        ubuntu@ip-172-31-1-190:~$ docker exec serfDC1 serf members
d4fd90692e18  127.0.0.1:7946  alive
6353e7f6134d  127.0.0.1:7946  failed
```

In the above trial, the MEMBER-JOIN request from serf agent Node2 located in container “serfDC2” reaches serf agent Node1 located in container “serfDC1“. However Node1 does not reply with an ACK to that request. This is caused by serf nodes not announcing themselves correctly over a *routable* IP address. Instead each serf node announces itself to the other as `"localhost"`, since each containers harbors its own “`localhost"`, resulting in confusion and failure to join.

Trouble-shooting the above issue has been both a source of frustration and extremely time-consuming. The absence of definitive answers on dedicated forums and an apparently rather small community of advanced users, capable of suggesting a solution. A query sent to Hashicorp (publisher and mantainer of serf) was left unanswered. Fellow students could not help either.

5.4 – *Solution for cluster formation*

It is only very late that we found a solution to the specific problem described above.

In our case the Docker containers’ hosting machine can be one or many of either an AWS-EC2 instance (or Virtual Private Server, VPS) or a physical (local) machine both running Ubuntu 14.04.5 and the same docker image cbhihe/serf-alpine-bash. We created 6 containers as needed on one host, named “`serfDC{1..6}`”. For the first container with IP 172.17.0.2

```
        ubuntu@ip-172-31-1-190:~$ docker exec serfDC1 serf agent -node=NODE1 -iface=eth0
```

For any subsequent containers among {serfDC2, ...serfDCN}, e.g. for serfDC5:

```
        ubuntu@ip-172-31-1-190:~$ docker exec serfDC5 serf agent -node=NODE1 -join=172.17.0.2 \
-iface=eth0
```

The result is a cluster of dockerized serf nodes. The exact same procedure can be used to build a cluster of etcd clients. Only the join primitive syntax is different.

```
        ubuntu@ip-172-31-1-190:~$ docker ps
CONTAINER ID        IMAGE                     COMMAND             CREATED             STATUS
PORTS                                                                        NAMES
e7936ba543aa        cbhihe/serf-alpine-bash   "/bin/bash"         4 hours ago         Up 4 hours
0.0.0.0:32791->7373/tcp, 0.0.0.0:32790->7946/tcp, 0.0.0.0:32789->8443/tcp   serfDC6
03dd7eef7e34        cbhihe/serf-alpine-bash   "/bin/bash"         5 hours ago         Up 5 hours
0.0.0.0:32782->7373/tcp, 0.0.0.0:32781->7946/tcp, 0.0.0.0:32780->8443/tcp   serfDC5
f474d361a499        cbhihe/serf-alpine-bash   "/bin/bash"         6 hours ago         Up 6 hours
0.0.0.0:32779->7373/tcp, 0.0.0.0:32778->7946/tcp, 0.0.0.0:32777->8443/tcp   serfDC4
a166a3b3389e        cbhihe/serf-alpine-bash   "/bin/bash"         6 hours ago         Up 6 hours
0.0.0.0:32776->7373/tcp, 0.0.0.0:32775->7946/tcp, 0.0.0.0:32774->8443/tcp   serfDC3
ced25e2f6649        cbhihe/serf-alpine-bash   "/bin/bash"         6 hours ago         Up 6 hours
0.0.0.0:32773->7373/tcp, 0.0.0.0:32772->7946/tcp, 0.0.0.0:32771->8443/tcp   serfDC2
01f5f48457b6        cbhihe/serf-alpine-bash   "/bin/bash"         6 hours ago         Up 6 hours
0.0.0.0:32770->7373/tcp, 0.0.0.0:32769->7946/tcp, 0.0.0.0:32768->8443/tcp   serfDC1

       ubuntu@ip-172-31-1-190:~$ docker exec serfDC1 serf members
NODE3  172.17.0.4:7946  alive
NODE4  172.17.0.5:7946  alive
NODE5  172.17.0.6:7946  alive
NODE6  172.17.0.7:7946  alive
NODE1  172.17.0.2:7946  alive
NODE2  172.17.0.3:7946  alive
```

6. Future work and conclusion

We experienced major difficulties in troubleshooting the communication issues that plagued this project from beginning to end. A paucity of information in the form of tutorials, similar use-cases and community forum support has compounded those problems.

For that reason we contemplate future work in this area as the continued pursuit of the goals expounded in Section 4. Further work should conducted to master various cluster topologies’ configuration. In particular building clusters whose nodes are distributed across several hosts (VPS) and geographical regions on AWS seems a logical continuation of our first step, From a pure networking technology standpoint, configuring IPTables so as to make cluster communication secure is also an issue of particular relevance in production.

Despite not being able to present numerical results , we nevertheless feel that the benefits derived from researching issues related to inter-container communication, serf and etcd, far outweigh the frustration of not being able to present results on communication performance in decentralized (P2P) networks (serf) and distributed networks (etcd).

As the adjoined supporting material will show, we delved deep into the practical aspects of IPTables and NAT table configuration in the context of the deployment of Docker containers. We also became familiar with the AWS eco-systems including its CLI API (`'apicli'`), security groups, limits and caps, AMI/snapshot management as well as pricing options. Learning how to build our own custom Docker container images, in view to minimize in RAM Runtime memory requirements led us to the discovery of a rich ecosystem with extensive resources to leverage and more still yet to be.

Acknowledgement:
Cédric Bhihe would like to thank Toni Pohl for his encouragements and advice on how to build proper Docker images, how to navigate GitHub and how to treat AWS instances.

Project handout: An evaluation of `Serf` and `etcd`

Serf implements a decentralized gossiping protocol. Etcd implements distributed key value.

The objective of the work is to understand the performance and limits of `Serf` and `etcd`.

Both technologies are available at:
Serf -- https://www.Serfdom.io/
etcd -- https://coreos.com/etcd/docs/latest/getting-started-with-etc.hdtml

Expected behavior: Robust behavior in dynamic situations.

Expected result: Performance characterization and comparison

References

[1] C. Richardson, “Microservices: Decomposing Applications for Deployability and Scalability,” *InfoQ*, 25-May-2014. .

[2] S. Cheshire and M. Krochmal, “RFC6763 - DNS-Based Service Discovery,” IETF, IETF Standard Track RFC6763, Feb. 2013.

[3] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “ZooKeeper: Wait-free Coordination for Internet-scale Systems,” in *Proceedings of the 2010 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2010, p. 14.

[4] B. Reed, “ZooKeeper: Wait-free Coordination for Internet-scale Systems,” presented at the 2010 USENIX Annual Technical Conference, Berkeley, CA, USA, 2010.

[5] “Chef (software),” *Wikipedia*. 03-Apr-2017.

[6] “Puppet (software),” *Wikipedia*. 27-Mar-2017.

[7] “Ansible (software),” *Wikipedia*. 09-Apr-2017.

[8] “Salt (software),” *Wikipedia*. 26-Mar-2017.

[9] A. Dagbar, “Consul: Service-oriented at Scale,” presented at the QCon, San Francisco, CA (USA), 2014.

[10] “Serf by HashiCorp.” [Online]. Available: https://www.serf.io/. [Accessed: 12-Apr-2017].

[11] “Gossip protocol,” *Wikipedia*. 08-Apr-2017.

[12] C. Thompson, “How to set up a Serf cluster on several Ubuntu VPS,” *DigitalOcean*, 23-Jan-2014. [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-set-up-a-serf-cluster-on-several-ubuntu-vps.

[13] T. Thieman, “Fast, zero-downtime deploy with Serf and Lox,” *Travis Thieman*, 20-Jul-2014. .

[14] CoreOS, “etcd v3 Getting started,” *Getting started with etcd*. [Online]. Available: https://coreos.com/etcd/docs/latest/getting-started-with-etcd.html.

[15] L. Lamport, “The Part-Time Parliament,” *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[16] L. Lamport, “Paxos Made Simple,” *ACM SIGACT News*, vol. 32, no. 4, pp. 51–58, Dec. 2001.

[17] T. Chandra, R. Griesemer, and J. Redstone, “Paxos Made Live -- An Engineering Perspective,” in *Proc. of PODC 2007*, Portland, OR (USA), 2007.

[18] M. Burrows, “The Chubby Lock Service for Loosely-Coupled Distributed Systems,” in *Proc. Seventh Symposium on Operating System Design and Implementation*, Seattle, WA, 2006, p. 16.

[19] D. Ongaro and J. Ousterhout, “In Search of an Understandable Consensus Algorithm,” in *Proc. of the 2014 Usenix Anual Tech. Conf.*, Philadelphia, PA, 2014, pp. 305–319.

[20] CoreOS, “etcd v3.1.5 Documentation,” *coreOS.com*. [Online]. Available: https://coreos.com/etcd/docs/latest/.

[21] CoreOS, “etcd v3 API Documentation.” [Online]. Available: https://github.com/coreos/etcd. [Accessed: 03-Jun-2017].

[22] SequenceIQ, “Serf on Docker containers,” *Serf on Docker*. [Online]. Available: https://github.com/sequenceiq/docker-serf. [Accessed: 03-Jun-2017].

[23] CoreOS, “Running etcd v3 under Docker.” [Online]. Available: https://coreos.com/etcd/docs/latest/v2/docker_guide.html.

[24] M. Goldmann, “Connecting Docker containers on multiple hosts,” 21-Jan-2014.