> A newer version of this document is available at **Reference Architecture** Universal Control Plane 2.0 Service Discovery and Load Balancing (https://success.docker.com/api/asset/.%2Frefarch%2Fucp-1-1-service-discovery%2F%2Farticle%2FDocker_Reference_Architecture-_Universal_Control_Plane_2.0_Service_Discovery_and_Load_Balancing).

Introduction

When developing applications, developers focus on functionality, speed, robustness, and quality of the application itself more than the ongoing operations. However, the shift to DevOps in application deployment practices has forced developers to own not only the application's development but also its deployment operations (developers are no longer pager-duty-free!). This shift also encouraged the operations teams to provide a common, scalable, and secure infrastructure that multiple developer teams can use to build, test, stage and deploy their applications.

With this shift, DevOps teams want to ensure that their applications are scalable. This means that these applications need to be broken up into, and advertised as smaller, decoupled microservices that can be easily scaled across large compute clusters. The microservices approach emphasized two key architectural considerations: **service discovery** and **load balancing**. This means that as developers build their applications to scale, they need to consider and design how each component (service) is being discovered by other services within or from outside the cluster. Additionally, as these services scale horizontally across the cluster, they should be equally utilized for maximum load distribution.

Docker Universal Control Plane (UCP) was built with this operational shift in mind. Docker UCP is available as part of Docker Datacenter to address both the developers' requirement for a seamless path from development to production and IT Operations' requirement for building a secure and scalable Docker infrastructure. Docker Datacenter includes UCP, Trusted Registry and Commercially Supported Docker Engines. As an integrated platform, Docker Datacenter empowers application teams to build a Containers as a Service (CaaS) environment either on-premises or in the cloud

This reference architecture is designed to provide guidance towards a supported high availability configuration of UCP with dynamic service discovery and load balancing. Docker provides official support for Docker products as governed by the Docker Datacenter end-user service agreement. For this reference architecture, Docker will provide support for Interlock (per the service levels provided in the customer contract for Docker Datacenter Subscription) as well as best effort support for 3rd party software, although official support for such software must be obtained through those 3rd party vendors.

Goal

In this reference architecture, you will use a sample application to learn how to setup a highly-available Universal Control Plane (UCP) cluster to enable dynamic service discovery and load balancing.

The end goal is to:

- Deploy the application on UCP
- Ensure that all of its services are discoverable within the cluster
- Ensure two of its services are accessible and load balanced from outside the cluster with pre-determined DNS names

Sample Application

We will use a sample Dockerized application (Voting App) as a reference application for the exercise. The sample application is composed of five (5) microservices:

- **voting-app**: A Python webapp which lets you vote between two options. (External DNS: vote.example.com)
- **result-app**: A Node.js webapp which shows the results of the voting in real time (External DNS: results.example.com)
- **redis**: A Redis queue which collects new votes
- **worker**: A .Net worker which consumes votes and stores them in...
- **db**: A Postgres database backed by a Docker volume

Sample Application Docker Compose File:

```
version: "2"

services:
  voting-app:
    image: nicolaka/voting-app:latest
    ports:
      - "80"
    networks:
      voteapp:
  result-app:
    image: nicolaka/result-app:latest
    ports:
      - "80"
    networks:
      voteapp:
  worker:
    image: nicolaka/worker:latest
    networks:
      voteapp:
  redis:
    image: redis
    ports:
      - "6379"
    networks:
      voteapp:
    container_name: redis

  db:
    image: postgres:9.4
    volumes:
      - "db-data:/var/lib/postgresql/data"
    networks:
      voteapp:
    container_name: db
volumes:
  db-data:

networks:
  voteapp:
```

Assumptions

It is assumed that you already have a working understanding of the Docker Datacenter, in particular the following components: Docker Universal Control Plane, Docker Swarm, and Docker Compose.

If you are not familiar with them, please refer to the following resources:

- Basic understanding of Docker UCP (https://docs.docker.com/ucp)
- Basic understanding of Docker Swarm (https://docs.docker.com/swarm)
- Basic understanding of Docker Compose (https://docs.docker.com/compose)
- Basic understanding of NGINX (https://www.nginx.com/)

Requirements

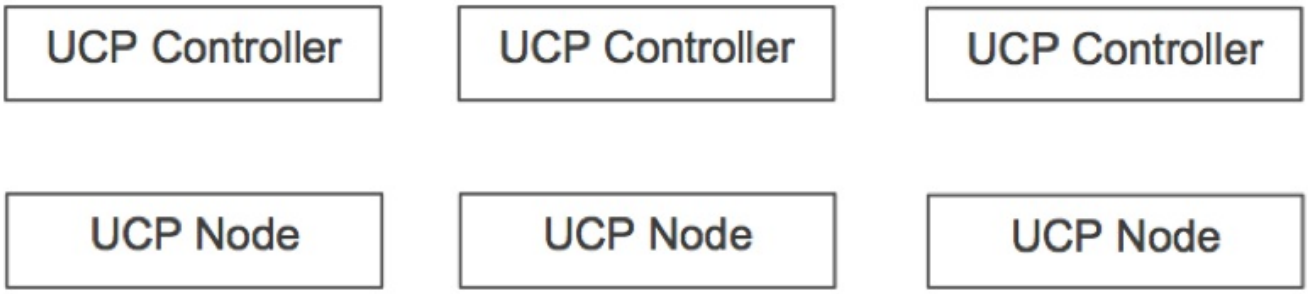
There are software version requirements for this reference architecture. Other variations have not been tested or validated. For more details on software compatibility and interoperability please go to the Compatibility Matrix (https://success.docker.com/Policies/Compatibility_Matrix).

- Docker UCP 1.0.0
- Docker Compose 1.6.1
- Commercially-supported Docker Engine 1.10

Prerequisites

To follow this reference architecture, you will need the following working environment:

- At least 3 UCP Controllers Nodes
- At least 3 UCP Cluster Nodes
- A designated DNS record for UCP (e.g. ucp.example.com)
- Unrestricted network between the nodes
- Network reachability to the UCP controller nodes on TCP port 80/443



Note: You do not need to have UCP installed. You will install it at a later step. However, you can review the installation requirements (https://docs.docker.com/ucp/plan-production-install/).

Note: You do not need to have UCP installed. You will install it at a later step.

Design Considerations

There are multiple considerations for designing a production-ready infrastructure using Docker Datacenter. From an operational point of view, it is important to ensure that UCP itself is highly available so that any failure in one or more UCP controllers won't result in an inability to access the UCP controller. Additionally, providing a scalable, secure, and stateless load-balancing service for all applications is important so that as the application scales. Load balancing can dynamically ensure that traffic is equally distributed across all of the containers providing these services.

From a developer's perspective, it is important to ensure that any design should integrate with the established developer workflow. For example, if developers use Docker Compose to build their applications locally during development, the new design should ensure Compose files can be used to deploy to production. Either directly by the developers or through a coordinated sign-off process to the deployment operations team. Additionally, it is important to ensure that each service deployed on Docker Datacenter is easily discoverable and reachable by other services that are part of the same application, regardless of where the containers providing these services are deployed in the cluster. This means that developers can assume that moving their apps from local development to production cluster will not break the application. Finally, it is crucial to ensure that the developers' apps are easily discoverable and accessible from outside the cluster regardless of which cluster or cluster node they are deployed to. This means that as the app moves from one cluster to another, developers should not worry about losing access to their applications.

In summary, there are three key design considerations that need to be addressed to ensure the developers and IT operations teams requirements are met:

- UCP High Availability
- Internal Service Discovery + Load Distribution
- External Service Discovery + Load Distribution

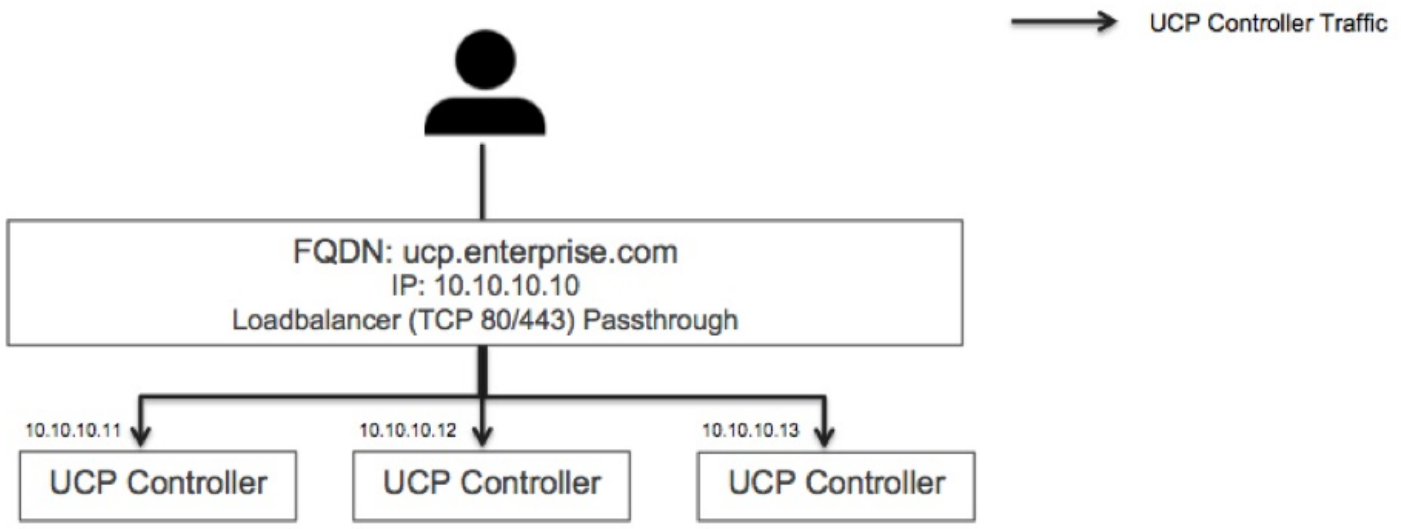
Solution Overview

In the following sections, we will go through each of the three design considerations and provide a solution to address them. We will start by building a highly-available UCP cluster that can withstand controller failure without impacting deployment operations. We will then focus on addressing the concern of intra-cluster services discovery. Finally, we will go through designing a highly-scalable load-balancing infrastructure that uses industry standards.

1. High-Availability UCP Cluster

Docker UCP supports high availability (HA) by replicating the UCP controller along with the underlying Swarm Manager and key-value store containers within your cluster. When you deploy UCP, you start by deploying the first UCP controller followed by the replicas. Functionally, all controllers are the same. HA requires at least three (3) controllers, a primary and two replicas, to be configured on three separate nodes. It is not recommended to run a cluster with only the primary controller and a single replica as this results in a split-brain scenario (e.g. each controller thinks it is the master controller in the cluster). Additionally, HA mode requires an odd number of controllers to achieve simple majority (quorum).

Failure tolerance for UCP HA deployments can be summarized as follows:
Number Of Deployed ControllersFailure Tolerance10315273
UCP controllers are stateless by design. All UCP controllers accept requests and then forward them to the underlying Swarm Manager. Any controller failure when UCP is deployed in HA will not have any impact on your UCP cluster, both from UCP web access (UI) or Docker client requests using the CLI. However, if you're statically mapping a DNS record to a primary UCP controller's IP address and that controller goes down, you will not be able to reach UCP. For that reason, it is recommended to deploy a UCP controller load balancer. An upstream load balancer can distribute all UCP requests to all three controllers behind it. Similarly, if you're deploying UCP in a public cloud, you can create a load balancer directly from the cloud provider (For example, AWS ELB or Azure Load Balancer)



As a sample reference, an HAProxy load balancer config file is as follows:

```
global
  maxconn 256
defaults
  mode tcp
  timeout connect 5000ms
  timeout client 50000ms
  timeout server 50000ms

frontend public
  option tcplog
  bind *:80
  redirect scheme https code 301 if !{ ssl_fc }
  bind *:443
  default_backend servers

backend servers
  mode tcp
  balance roundrobin
  server ucp1 10.10.10.11:443 check
  server ucp2 10.10.10.12:443 check
  server ucp3 10.10.10.13:443 check
```

Here are some recommended UCP controller and load balancer configurations:

- **Health Checks:** The load balancer can use UCP's API endpoint /_ping to ensure that each of the controllers is healthy. A 200 OK response means that the controller is healthy and can receive traffic.
- **Listeners:** The load balancer should be configured to load balance using TCP ports 80 and 443. The load balancer should not terminate/reestablish HTTPS connections due to mutual TLS connection requirement in order to use Docker Client with UCP.
- **DNS:** A DNS record should be mapped to the load balancer itself (e.g. VIP) and not to any individual controller.
- **IPs:** The load balancer can load balance to the controller's private or public IPs as long as the load balancer can reach each of the controllers.

- **SSL Certificates:** When you install the UCP controllers, ensure that you use the Fully Qualified Domain Name (FQDN) of the UCP when asked for additional Subject Alternative Names (SAN). You need to do this on ALL UCP controllers (including the replicas). The SANs are used by the UCP Certificate Authority to sign the SSL certificates. If you would like to use your own CA to sign UCP's certificate, please follow these directions (https://docs.docker.com/ucp/production-install/#step-5-customize-the-ca-used-optional).

Should any controller fail, the UCP Controller load balancer will ensure that UCP can be reached and that all Docker deployment workflows are run without impact.

Now that you have the full requirements for UCP HA deployment, you can easily deploy UCP following these directions (https://docs.docker.com/ucp/production-install/).

2. Internal Service Discovery + Load Distribution

For different microservice containers to communicate, they must first discover each other. The introduction of multi-host networking in Docker 1.9 enabled multiple services belonging to a single application to be connected via an Overlay Network that spans multiple nodes. Docker 1.10 added an embedded DNS-based for hostname lookups for a more reliable and scalable service discovery.

Note: Read Overlay Networks (https://docs.docker.com/engine/userguide/networking/get-started-overlay/) for more details.

Containers deployed using Docker 1.10 can now use DNS to resolve the IP addresses of other containers on the same network. This behavior works out of the box for user-defined bridge and overlay networks.

Docker 1.10 also introduced the concept of *network alias*. A network alias abstracts multiple containers under a single alias. This means that scaled services (e.g `docker-compose scale service=number`) can be grouped under and resolved by a single alias. Docker will resolve the alias to a healthy container that belongs to that alias. This is extremely helpful for stateless services where any container can be used to provide a service. Docker uses round-robin DNS resolution to load-balance requests across all healthy containers that are part of an alias.

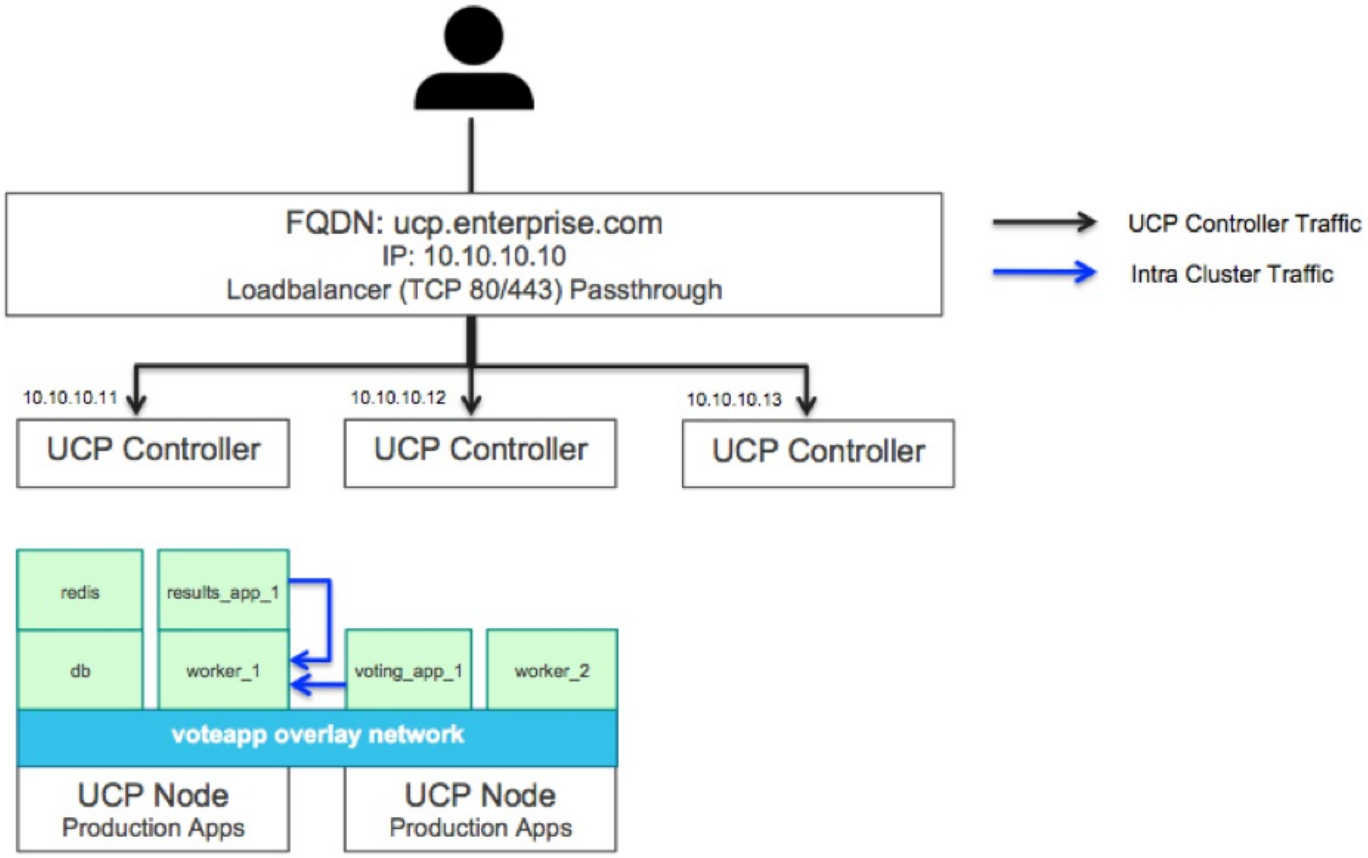
Example In our sample Voting App, the Java `worker` service can belong to the alias `workers`. If additional workers are needed, you can scale the `worker` service using Compose. All the `worker` services can belong to a single alias called `workers`. If other services need to connect with any of these services, Docker will resolve the `workers` alias to a healthy container. We can add network aliases for the `worker` service in the Compose file as follows:

```
<snippet>

  worker:
    image: nicolaka/worker:latest
    networks:
      voteapp:
       aliases:
        - workers

<snippet>
```

We can now deploy the app on UCP using Docker Compose. The `worker` service is then scaled such that there are two containers running (`worker_1` and `worker_2`). Other services can reach either worker containers by using the container name `worker_1` or the alias name (`workers`). In the case that `worker_1` goes down, Docker will automatically resolve the `workers` alias to `worker_2`.



3. External Service Discovery + Load Distribution

Some services are designed to be accessible from outside the UCP cluster (typically by a DNS name) either as services that need to be accessed by other services in a different cluster or by external public users/services. To access these services, you typically need to create a DNS record for each service and map it to the exact node that that service is running on. If you also need to balance the load across multiple containers, you need to add a load balancer and reconfigure it every time a container comes up/goes down. This process is tedious and not scalable.

An easier, more scalable, and automated solution to enable external service discovery and load balancing is to use an event-driven service registrator that automatically updates a load-balancer's config as containers go up or down in your UCP cluster. This can be achieved by combining [Interlock](https://success.docker.com/api/asset/.%2Frefarch%2Fucp-1-1-service-discovery%2Fgithub.com%2Fehazlett%2Finterlock) with a loadbalancer (NGINX).

Interlock is a containerized, event-driven tool that connects to the UCP controllers and watches for events. In this case, events can be containers being spun up or going down. Interlock also looks for certain metadata that these containers have. These can be hostnames or labels configured for the container. It then uses the metadata to register/de-register these containers to a load balancing backend. The load balancer uses updated backend configs to direct incoming requests to healthy containers. Both Interlock and the load balancer containers are stateless, and hence can be scaled horizontally across multiple nodes to provide a highly-available load balancing services for all deployed applications.

How it Works

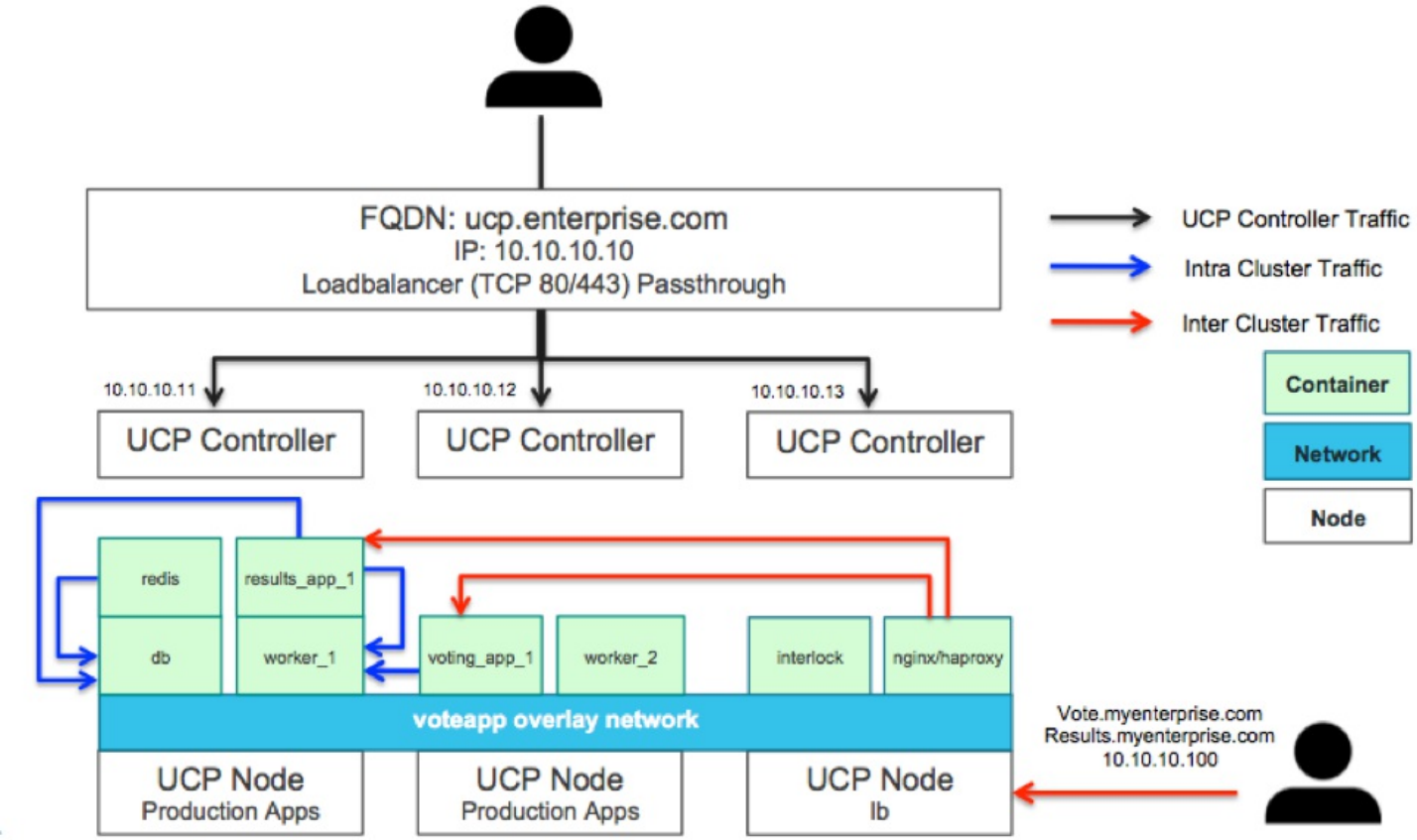
First, you need to deploy Interlock and the load balancer containers on a regular UCP node(or multiple UCP nodes). It is recommended that you dedicate some UCP cluster nodes for external connectivity and load balancing services. These nodes need to have externally routable IP addresses reachable by the services that

need to access your application. The other nodes running your services do not have to have externally routable IP addresses. In this example, we will one of the three UCP nodes (we will call it `lb`) to deploy Interlock and the load balancer using Docker Compose.

Second, you need to create a DNS record that represents your application's domain name and map it to the IP address(s) of `lb`.If you intend to deploy multiple `lb` nodes, you can register the A-record for the VIP of these instances.

Finally, you need to add specific metadata in the form of container labels when deploying your application. The labels are then used by Interlock to register the container against the load balancer.

These steps provide the necessary service registration and load balancing solution that can be used by any developer when deploying their applications on UCP. Follow these step-by-step procedures to configure your UCP cluster based on your preferred industry-standard load balancing backend (NGINX). The following diagram shows the load balancing solution.



The following steps provide a guideline to configuring the load-balancing solution on a dedicated UCP node using Interlock + NGINX/NGINX+:

1. On the dedicated UCP node (**lb**), install Docker Compose (https://docs.docker.com/compose/install/). Then ensure that docker-compose is installed:

```
$ docker-compose version
docker-compose version 1.6.2, build 4d72027
docker-py version: 1.7.2
CPython version: 2.7.6
OpenSSL version: OpenSSL 1.0.1f 6 Jan 2014
```

2. On the dedicated UCP node (**lb**) , create a new Docker Compose file called `docker-compose.yml` with the below content. **Note:** In this example, we're using the standard NGINX Docker image. However, you can use your own NGINX+ image. All you need to do is change the image for the `nginx` service in the Docker Compose file and repeat step #1 with the `NginxPlusEnabled = true` option.

```
interlock:
    image: ehazlett/interlock:master
    command: -D run
    tty: true
    ports:
        - 8080
    environment:
        INTERLOCK_CONFIG: |
            ListenAddr = ":8080"
            DockerURL = "${SWARM_HOST}"
            TLSCACert = "/certs/ca.pem"
            TLSCert = "/certs/cert.pem"
            TLSKey = "/certs/key.pem"
            PollInterval = "10s"

            [[Extensions]]
            Name = "nginx"
            ConfigPath = "/etc/nginx/nginx.conf"
            PidPath = "/etc/nginx/nginx.pid"
            MaxConn = 1024
            Port = 80
    volumes:
        - ucp-node-certs:/certs
    restart: always

nginx:
    image: nginx:latest
    entrypoint: nginx
    command: -g "daemon off;" -c /etc/nginx/nginx.conf
    ports:
        - 80:80
    labels:
        - "interlock.ext.name=nginx"
    restart: always
```

3. On the dedicated UCP node (**lb**), export an environment variable called **SWARM_HOST**. This variable should be the FQDN/IP address+ Swarm manager port of UCP Controller. The Swarm port is `2376` by default. You can check it by doing a `docker ps` and checking the host mounter port of the `ucp-swarm-manager` container on the UCP controller node. If you use a FQDN that is assigned to the UCP loadbalancer then please ensure that you're also forwarding on port access. For example, if you're using a private AWS ELB to loadbalance across the UCP Controllers, it needs to forward TCP port `2376`. Alternatively, you can use the private IP address of any of the UCP controllers.

```
$ export SWARM_HOST=tcp://<private_IP_of_ANY_UCP_controller>:2376
```

4. On the dedicated UCP node (**lb**), deploy Interlock+NGINX using the following docker-compose command:

```
$ docker-compose up -d
Creating interlock_nginx_1
Creating interlock_interlock_1
```

5. Confirm that Interlock is connected to the Swarm event stream:

```
 $ docker-compose logs
 Attaching to interlock_interlock_1, interlock_nginx_1
 interlock_1  | INFO[0000] interlock 1.2.0-master
(https://success.docker.com/api/asset/.%2Frefarch%2Fucp-1-1-service-discovery%2F2fd9af6)
 interlock_1  | DEBU[0000] loading config from environment
 interlock_1  | DEBU[0000] using tls for communication with docker
 interlock_1  | DEBU[0000] docker client: url=tcp://192.168.3.100:2376
 interlock_1  | DEBU[0000] loading extension: name=nginx
 interlock_1  | DEBU[0000] using internal configuration template         ext=lb
 interlock_1  | INFO[0000] interlock node:
id=eed2837eb4807518b9ff55b49ec29ad99415816b49eb7ea6176f3953d1842db1  ext=lb
 interlock_1  | DEBU[0000] starting event handling
 interlock_1  | INFO[0000] using polling for container updates: interval=10s
```

Application Deployment Configuration

Now that Interlock+LB are up and configured to listen on Swarm events. You can start deploying your applications on the UCP cluster. Interlock expects specific container metadata via labels. A complete list of all Interlock options can be found here (https://github.com/ehazlett/interlock/blob/ng/docs/interlock_data.md).

In our sample app, we want to expose two services externally. These services are `voting-app` and `results-app`. For interlock to register these services as backends for the load balancer, we need to provide additional container labels. In our case, we want `voting-app` to be registered as `vote.example.com` and `results-app` as `results.example.com`. The DNS records need to be mapped to the IP addrress of (**lb**). You may also map a wildcard DNS record to the **lb**. To configure and deploy the app, follow the below steps:

1. From your local machine, download a UCP client bundle. Instructions can be found here (https://docs.docker.com/ucp/deploy-application/#step-2-get-the-client-bundle-and-configure-a-shell).

2. Ensure that you have Docker Compose installed on your local environment.

```
$ docker-compose version
docker-compose version 1.6.2, build 4d72027
docker-py version: 1.7.2
CPython version: 2.7.6
OpenSSL version: OpenSSL 1.0.1f 6 Jan 2014
```

3. Ensure that you're pointing your local Docker client to the UCP controller:

```
$ cd /path/to/ucp-bundle-admin
ucp-bundle-admin$ source env.sh
```

   Followed by:

```
ucp-bundle-admin$ docker version
Client:
 Version:      1.10.1
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   9e83765
 Built:        Thu Feb 11 19:27:08 2016
 OS/Arch:      linux/amd64

Server:
 Version:      ucp/1.0.0
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   5c4f6d8
 Built:
 OS/Arch:      linux/amd64
```

4. We need to adjust the app's Compose file to add the necessary Interlock labels.

   For `voting-app` we add the following:

   ```
   labels:
    interlock.hostname: "vote"
    interlock.domain:   "example.com"
   ```

   For `results-app` we add the following:

   ```
   labels:
    interlock.hostname: "results"
    interlock.domain:   "example.com"
   ```

   The complete docker-compose file now should like this

```
version: "2"

services:
  voting-app:
    image: nicolaka/voting-app:latest
    ports:
      - "80"
    networks:
      voteapp:
    labels:
         interlock.hostname: "vote"
         interlock.domain:   "example.com"
  result-app:
    image: nicolaka/result-app:latest
    ports:
      - "80"
    networks:
      voteapp:
    labels:
         interlock.hostname: "results"
         interlock.domain:   "example.com"
  worker:
    image: nicolaka/worker:latest
    networks:
      voteapp:
       aliases:
        - workers
  redis:
    image: redis
    ports:
      - "6379"
    networks:
      voteapp:
    container_name: redis

  db:
    image: postgres:9.4
    volumes:
      - "db-data:/var/lib/postgresql/data"
    networks:
      voteapp:
    container_name: db
volumes:
  db-data:

networks:
  voteapp:
```

5. Deploy the app on UCP using Docker Compose:

```
$ docker-compose up -d
Creating network "examplevotingapp_voteapp" with the default driver
Creating examplevotingapp_worker_1
Creating db
Creating redis
Creating examplevotingapp_voting-app_1
Creating examplevotingapp_result-app_1
$ docker-compose ps
            Name                           Command               State            Ports
-------------------------------------------------------------------------------------------------
db                              /docker-entrypoint.sh postgres   Up      5432/tcp
examplevotingapp_result-app_1   node server.js                   Up      0.0.0.0:32808->80/tcp
examplevotingapp_voting-app_1   /bin/sh -c dotnet Worker.dll     Up      0.0.0.0:32809->80/tcp
examplevotingapp_worker_1       /bin/sh -c dotnet Worker.dll     Up
redis                           docker-entrypoint.sh redis ...   Up      0.0.0.0:32807->6379/tcp
```

6. On the **lb**, confirm that Interlock registered the apps with the load balancer by looking at its logs. You should see the "restarted proxy container" message if Interlock registered the container successfully.

```
 $ docker-compose -f nginx-docker-compose.yml logs -f
 Attaching to interlock_nginx_1, interlock_interlock_1
 interlock_1  | INFO[0000] interlock 1.2.0-master
(https://success.docker.com/api/asset/.%2Frefarch%2Fucp-1-1-service-discovery%2F2fd9af6)
 interlock_1  | DEBU[0000] loading config from environment
 interlock_1  | DEBU[0000] using tls for communication with docker
 interlock_1  | DEBU[0000] docker client: url=tcp://192.168.23.35:2376
 interlock_1  | DEBU[0000] loading extension: name=nginx
 interlock_1  | DEBU[0000] using internal configuration template         ext=lb
 interlock_1  | INFO[0000] interlock node:
id=476e5c33b0d37c51979eb9378fde5a29f001e7397863c32d856eb4a890b17d35  ext=lb
 interlock_1  | DEBU[0000] starting event handling
 interlock_1  | INFO[0000] using polling for container updates: interval=10s
 interlock_1  | DEBU[0010] detected new containers; triggering reload
 interlock_1  | DEBU[0010] event received: status=interlock-restart id=1470273010924854177 type=
action=
 interlock_1  | DEBU[0010] notifying extension: lb
 interlock_1  | DEBU[0010] triggering reload                             ext=lb
 interlock_1  | DEBU[0014] reaping key: reload
 interlock_1  | DEBU[0014] triggering reload from cache                  ext=lb
 interlock_1  | DEBU[0014] checking to reload                            ext=lb
 interlock_1  | DEBU[0014] updating load balancers                       ext=lb
 interlock_1  | DEBU[0014] generating proxy config                       ext=lb
 interlock_1  | DEBU[0014] websocket endpoints: []                       ext=nginx
 interlock_1  | DEBU[0014] alias domains: []                             ext=nginx
 interlock_1  | INFO[0014] result.example.com: upstream=192.168.24.18:32808  ext=nginx
 interlock_1  | DEBU[0014] websocket endpoints: []                       ext=nginx
 interlock_1  | DEBU[0014] alias domains: []                             ext=nginx
 interlock_1  | INFO[0014] vote.example.com: upstream=192.168.24.18:32809  ext=nginx
 interlock_1  | DEBU[0014] proxy config path: /etc/nginx/nginx.conf      ext=lb
 interlock_1  | DEBU[0014] detected proxy container:
id=4452bce7c1e1bd5919b51cf4e40d381705635523f20ac72a6a943c5b9b1b3d65 backend=nginx  ext=lb
 interlock_1  | DEBU[0014] proxyContainers:
[{4452bce7c1e1bd5919b51cf4e40d381705635523f20ac72a6a943c5b9b1b3d65 [/ip-192-168-24-
18/interlock_nginx_1] nginx:latest nginx -g 'daemon off;' -c /etc/nginx/nginx.conf 1470272999 Up 10
seconds [{ 443 0 tcp} {192.168.24.18 80 80 tcp}] 0 0 map[com.docker.compose.project:interlock
com.docker.compose.service:nginx com.docker.compose.version:1.8.0 interlock.ext.name:nginx
com.docker.compose.config-hash:7d2169ca9bdc4664f5665b69c1e3b4dd679821835037ecadd6718c91d16326db
com.docker.compose.container-number:1 com.docker.compose.oneoff:False] {map[bridge:{<nil> [] []
2aa1ebd461d18bf74ca94fc34a38446c12da08b88be062bbfcce02de39e0d740 172.17.0.1 172.17.0.3 16   0
02:42:ac:11:00:03}]}}]  ext=lb
 interlock_1  | DEBU[0014] saving proxy config                           ext=lb
 interlock_1  | DEBU[0014] updating proxy config:
id=4452bce7c1e1bd5919b51cf4e40d381705635523f20ac72a6a943c5b9b1b3d65  ext=lb
 interlock_1  | DEBU[0015] signaling reload                              ext=lb
 interlock_1  | DEBU[0016] reloading proxy container:
id=4452bce7c1e1bd5919b51cf4e40d381705635523f20ac72a6a943c5b9b1b3d65  ext=nginx
 interlock_1  | INFO[0016] restarted proxy container: id=4452bce7c1e1 name=/ip-192-168-24-
18/interlock_nginx_1  ext=nginx
 interlock_1  | DEBU[0016] triggering proxy network cleanup              ext=lb
 interlock_1  | INFO[0016] reload duration: 2267.36ms                    ext=lb
 interlock_1  | DEBU[0016] checking to remove proxy containers from networks  ext=lb
```

7. You can now access the app by going to http://vote.example.com (http://vote.example.com) or http://results.example.com (http://results.example.com).

8. If you need to scale the voting-app service, you can simply scale it using docker-compose. Interlock will add the newly added container to the voting-app backend. You will note that your request will be served from a different container each time you hit `vote.example.com`.

```
$ docker-compose scale voting-app=10
Creating and starting 2 ... done
Creating and starting 3 ... done
Creating and starting 4 ... done
Creating and starting 5 ... done
Creating and starting 6 ... done
Creating and starting 7 ... done
Creating and starting 8 ... done
Creating and starting 9 ... done
Creating and starting 10 ... done
```

Summary

In this Reference Architecture, we set up a highly-available Docker Universal Control Plane (UCP) cluster and enabled dynamic built-in service discovery and load balancing by addressing three key design requirements: Universal Control Plane High-Availability, Cluster Service Discovery with Load Distribution, and External Service Discovery with Load Distribution. Additionally, sample configurations and workflows for deploying microservice applications on Universal Control Plane.

For more information on Universal Control Plane and the rest of the Docker Datacenter subscription, visit our website at www.docker.com/products/docker-datacenter.