Introduction

Service discovery registers a service and publishes its connectivity information so that other services are aware of how to connect to the service. As applications move toward microservices and service-oriented architectures, service discovery has become an integral part of any distributed system, increasing the operational complexity of these environments.

Docker Enterprise Edition includes service discovery and load balancing capabilities to aid the devops initiatives across any organization. Service discovery and load balancing make it easy for developers to create applications that can dynamically discover each other. Also, these features simplify the scaling of applications by operations engineers.

Docker uses a concept called **services** to deploy applications. Services consist of containers created from the same image. Each service consists of tasks that execute on worker nodes and define the state of the application. When deploying a service, a service definition is included upon service creation. The service definition consists of information that includes, among other things, the containers that comprise the service, which ports are published, which networks are attached, and the number of replicas. All of these tasks together make up the desired state of the service. If a node fails a health check or if a specific service task defined in a service definition fails a health check, then the cluster reconciles the service state to another healthy node. Docker Enterprise includes service discovery, load balancing, scaling, and reconciliation events so that this orchestration works seamlessly.

For Kubernetes-based service discovery and load-balancing, please refer to this reference architecture ([https://success.docker.com/article/ucp-service-discovery-k8s](https://success.docker.com/article/ucp-service-discovery-k8s)).

What You Will Learn

This reference architecture covers the solutions that Docker Enterprise provides in the topic areas of service discovery and load balancing for swarm mode workloads. In swarm mode, Docker uses DNS for service discovery as services are created, and different routing meshes are built into Docker to ensure your applications remain highly available. The release of UCP 3.0 introduced a versatile and an enhanced version of application layer (Layer 7) routing mesh called the **Interlock Proxy** that routes HTTP traffic based on DNS hostname. After reading this document, you will have a good understanding of how Interlock Proxy works and how it integrates with the other service discovery and load balancing features native to Docker.

Additionally, UCP 3.0 introduced enterprise support for using Kubernetes as the orchestrator of your application workloads. The document linked below will provide a good understanding of how to use Kubernetes resource objects within Docker Enterprise to deploy, run, and manage application workloads.

Service Discovery with DNS

Docker uses embedded DNS to provide service discovery for containers running on a single Docker engine and **tasks** running in a Docker swarm. Docker engine has an internal DNS server that provides name resolution to all of the containers on the host in user-defined bridge, overlay, and MACVLAN networks. Each Docker container (or **task** in swarm mode) has a DNS resolver that forwards DNS queries to the Docker engine, which acts as a DNS server. The Docker engine then checks if the DNS query belongs to a container or **service** on each network that the requesting container belongs to. If it does, then the Docker engine looks up the IP address that matches the name of a container, **task**, or **service** in its key-value store and returns that IP or **service** Virtual IP (VIP) back to the requester.

Service discovery is **network-scoped**, meaning only containers or tasks that are on the same network can use the embedded DNS functionality. Containers not on the same network cannot resolve each others' addresses. Additionally, only the nodes that have containers or tasks on a particular network store that network's DNS entries. This promotes security and performance.
If the destination container or service and the source container are not on the same network, the Docker engine forwards the DNS query to the default DNS server.

In this example, there is a service of two containers called myservice. A second service (client) exists on the same network. The client executes two curl operations for docker.com and myservice. These are the resulting actions:

- DNS queries are initiated by client for docker.com and myservice.
- The container's built-in resolver intercepts the DNS queries on 127.0.0.11:53 and sends them to Docker Engine's DNS server.
- myservice resolves to the Virtual IP (VIP) of that service which is internally load balanced to the individual task IP addresses. Container names are resolved as well, albeit directly to their IP addresses.
- docker.com does not exist as a service name in the mynet network, so the request is forwarded to the configured default DNS server.

**Internal Load Balancing**

When services are created in a Docker swarm cluster, they are automatically assigned a Virtual IP (VIP) that is part of the service's network. The VIP is returned when resolving the service's name. Traffic to the VIP is automatically sent to all healthy tasks of that service across the overlay network. This approach avoids any client-side load balancing because only a single IP is returned to the client. Docker takes care of routing and equally distributes the traffic across the healthy service tasks.

To get the VIP of a service, run the `docker service inspect myservice` command like so:

```
# Create an overlay network called mynet
$ docker network create -d overlay mynet a59umzkdj2r0ua7x8jxd84dhr

# Create myservice with 2 replicas as part of that network
$ docker service create --network mynet --name myservice --replicas 2 busybox ping localhost 8t5r8cr0f0h6k2c3k7ih4l6f5

# Get the VIP that was created for that service
$ docker service inspect myservice
...

"VirtualIPs": [
  {
    "NetworkID": "a59umzkdj2r0ua7x8jxd84dhr",
    "Addr": "10.0.0.3/24"
  },
]
```

**Note:** DNS round robin (DNS RR) load balancing is another load balancing option for services (configured with `--endpoint-mode`). In DNS RR mode, a VIP is not created for each service. The Docker DNS server resolves a service name to individual container IPs in round robin fashion.
External Load Balancing (Swarm Mode Routing Mesh)

You can expose services externally by using the `--publish` flag when creating or updating the service. Publishing ports in Docker swarm mode means that every node in your cluster is listening on that port, but what happens if the service's task isn't on the node that is listening on that port?

This is where routing mesh comes into play. Routing mesh combines `ipvs` and `iptables` to create a powerful cluster-wide transport-layer (L4) load balancer. It allows all the swarm nodes to accept connections on the services published ports. When any swarm node receives traffic destined to the published TCP/UDP port of a running service, it forwards the traffic to the service's VIP using a pre-defined overlay network called `ingress`. The `ingress` network behaves similarly to other overlay networks, but its sole purpose is to transport mesh routing traffic from external clients to cluster services. It uses the same VIP-based internal load balancing as described in the previous section.

Once you launch services, you can create an external DNS record for your applications and map it to any or all Docker swarm nodes. You do not need to worry about where your container is running as all nodes in your cluster look as one with the routing mesh routing feature.

```bash
# Create an overlay network called appnet
$ docker network create -d overlay appnet
# Create a service with two replicas and export port 8000 on the cluster
$ docker service create --name app --replicas 2 --network appnet --publish 8000:80 nginx
```

This diagram illustrates how the routing mesh works.

- A service is created with two replicas, and it is port mapped externally to port 8000.
- The routing mesh exposes port 8000 on each host in the cluster.
- Traffic destined for the app can enter on any host. In this case the external LB sends the traffic to a host without a service replica.
- The kernel's IPVS load balancer redirects traffic on the `ingress` overlay network to a healthy service replica.

The Swarm Layer 7 Routing (Interlock Proxy)

The swarm mode routing mesh is great for transport-layer routing. It routes to services using the service's published ports. But what if you wanted to route traffic to services based on hostname instead? The "Swarm Layer 7 Routing (Interlock)" is a feature that enables service discovery on the application layer (L7). This Layer 7 Routing extends upon the swarm mode routing mesh by adding application layer capabilities such as inspecting the HTTP header. Interlock and swarm mode routing meshes are used together for flexible and robust service delivery. The addition of Interlock allows for each service to be accessible via a DNS label passed to the service. As the service scales horizontally and more replicas are added, the service uses round-robin load balancing as well.

The Interlock Proxy works by using the HTTP/1.1 header field definition. Every HTTP/1.1 TCP request contains a Host: header. A HTTP request header can be viewed using `curl`:
When using Interlock with HTTP requests, both the swarm mode routing mesh and Interlock are used in tandem. When a service is created using the `com.docker.lb.hosts` label, the Interlock configuration is updated to route all HTTP requests that contain the Host: header specified in the `com.docker.lb.hosts` label to route to the VIP of the newly created services. Since Interlock runs as a service, it is accessible on any node in the cluster using the configured published port.

The following is an overview diagram to show how the swarm mode routing mesh and Interlock work together:

- The traffic comes in from the external load balancer into the swarm mode routing mesh.
- The `ucp-interlock-proxy` service is configured to listen on port 8080 and 8443, so any request to port 8080 or 8443 on the UCP cluster will hit this service first.
- All services attached to a network that is enabled for "Hostname-based routing" can utilize the Interlock proxy to have traffic routed based on the HTTP Host: header.

The following graphic represents a closer look of the previous diagram. You can see how Interlock works under the hood.

- The core service is called `ucp-interlock`. This listens to the Docker remote API for events and configures an upstream service that is accessed by another service called `ucp-interlock-extension`.
- The extension service is called `ucp-interlock-extension`. This service queries the core `ucp-interlock` service and uses the response information from that service to generate the configuration file appropriate for the proxy service called `ucp-interlock-proxy`. The configuration file is generated in the form of a Docker config object which will be used by the proxy service.
- The proxy service is called `ucp-interlock-proxy`. This is the reverse proxy and is responsible for handling the actual requests for application services. The proxy uses the configuration generated by the corresponding extension service `ucp-interlock-extension`.
- The `ucp-interlock-proxy` service receives the TCP packet and inspects the HTTP header.
  - Services that contain the label `com.docker.lb.hosts` are checked to see if they match the HTTP Host: header.
  - If a Host: header and service label match, then the value of the label `com.docker.lb.port` is queried for that service. This instructs what port the `ucp-interlock-proxy` should use to access the application service.
  - Traffic is routed to the service's VIP on its port using the swarm mode routing mesh (L4).
  - If a service contains multiple replicas, then each replica container is load balanced via round-robin using
the internal L4 routing mesh.

Differences Between the Interlock Proxy and Swarm Mode Routing Mesh

The main difference between the Interlock Proxy and swarm mode routing mesh is that the Interlock Proxy is intended to be used only for HTTP traffic at the application layer, while the swarm mode routing mesh works at a lower level on the transport layer.

Deciding which to use depends on the application. If the application is intended to be publicly accessible and is an HTTP service, then the Interlock Proxy could be a good fit. If mutual TLS is required for the backend application, then using the transport layer would probably be preferred.

Another advantage of using the Interlock Proxy is that less configuration is required for traffic to be routed to the service. Often times only a DNS record is needed along with setting the label on the service. If a wildcard DNS entry is used, then no configuration outside of setting the service label is necessary. In many organizations, access to load balancers and DNS is restricted. Being able to control requests to applications by just a service label can empower developers to quickly iterate over changes. With the swarm mode routing mesh, any frontend load balancer can be configured to send traffic to the service's published port.

The following diagram shows an example with wildcard DNS:

![Diagram showing wildcard DNS]

Enabling the Swarm Layer 7 Routing (Interlock Proxy)

The Interlock Proxy can be enabled from the UCP web console. To enable it:

1. Log into the UCP web console.
2. Navigate to Admin Settings > Layer 7 Routing.
3. Check Enable Layer 7 Routing under the section titled Swarm Layer 7 Routing (Interlock).
4. Configure the ports for Interlock Proxy to listen on, with the defaults being 8080 and 8443. The HTTPS port defaults to 8443 so that it doesn't interfere with the default UCP management port (443).

Once enabled, UCP creates three services on the swarm cluster: `ucp-interlock`, `ucp-interlock-extension`, and `ucp-interlock-proxy`. The `ucp-interlock-proxy` service is responsible for routing traffic to the specified container based on the HTTP Host: header. Since the Interlock Proxy service is a swarm mode service, every node in the UCP cluster can route traffic to it by receiving traffic from ports 8080 and 8443. By default the Interlock Proxy service exposes ports 8080 and 8443 cluster-wide, and any requests on ports 8080 and 8443 to any node in the cluster are sent to the Interlock Proxy service.

Networks and Access Control

The Interlock Proxy uses one or more overlay networks to communicate with the backend application services. To allow the Interlock Proxy to communicate with and consequently forward requests to application frontend services, it needs to share a network with that service. This is accomplished by setting a label `com.docker.lb.network` to a value which is the name of the network the Interlock Proxy service should attach to for upstream connectivity. As such this action does not require administrator level access within UCP to be performed.

This configuration also allows the isolation between frontend services using the Interlock Proxy since the exposed application services do not share a common network with other similar exposed services.
Swarm Layer 7 Routing (Interlock Proxy) Requirements

There are three requirements that services must satisfy to use the Interlock Proxy:

1. The service must be connected to a network which is also defined as the value for the service label `com.docker.lb.network`.
2. The service must listen on a port. This port need not be exposed to the outside, but it should be configured as the value of the label `com.docker.lb.port` in the service.
3. The service must define a service label `com.docker.lb.hosts` to specify the host (or FQDN) served by the service. Multiple hosts can optionally be configured using a comma-separated list.

Configuring DNS with the Swarm Layer 7 Routing (Interlock Proxy)

This section covers how to configure DNS for services using the Interlock Proxy. To use the Interlock Proxy, a DNS record for the service needs to point to the UCP cluster. This can be accomplished through a variety of different ways because of the flexibility that the swarm mode routing mesh provides.

If a service needs to be publicly accessible for requests to `foo.example.com`, then the DNS record for that service can be configured in one of the following ways:

1. Configure DNS to point to any single node on the UCP cluster. All requests for `foo.example.com` will get routed through that node to the Interlock Proxy.
2. Configure round-robin DNS to point to multiple nodes on the UCP cluster. Any node that receives a request for `foo.example.com` will get routed through the Interlock Proxy.
3. Or, the best solution for high availability, is to configure an external HA load balancer to reside in front of the UCP cluster. There are some considerations to keep in mind when using an external HA load balancer:
   - Set the DNS record for `foo.example.com` to point to the external load balancer.
   - The external load balancer should point to multiple UCP nodes that reside in different availability zones for increased resiliency.
   - Configure the external load balancer to perform a TCP health check on the Interlock Proxy service's configured exposed port(s) so that traffic will route through healthy UCP nodes.

Which nodes should be used to route traffic, managers, or workers? There are a few ways to approach that question.

1. Routing through the manager nodes is fine for smaller deployments since managers are generally more static in nature.
   - **Advantage**: Manager nodes generally do not shift around (new hosts, new IPs, etc.) often, and it is easier to keep load balancers pointing to the same nodes.
   - **Disadvantage**: Manager nodes are responsible for the control plane traffic. If application traffic is large, you don't want to saturate traffic to these nodes and cause adverse affects to your cluster.
2. Routing through the worker nodes.
   - **Advantage**: Worker nodes do not manage the entire cluster, so there's less additional networking overhead.
   - **Disadvantage**: Worker nodes are treated like "pets" instances, which can make infrastructure less repeatable and less reliable. Any automation built around destroying and building nodes needs to take this into account if load balancers are pointing to worker nodes.
3. Routing through specific worker nodes (Host mode). This setup is also referred to as **Service clusters**. In
this setup, specific worker nodes will route traffic for a predetermined set of applications. Consequently, another set of worker nodes can be setup to route traffic for a different set of applications, potentially using a different configuration or extension.

- **Advantages:**
  - Application traffic is completely segregated from one another.
  - Flexibility exists when applying different extensions / configurations for each application or application cluster.
  - Higher control exists over which hosts will route traffic to specific sets of applications or services. For instance, this feature can be leveraged to constrain the Interlock Proxy on hosts that are in different regions to offer high availability.
  - Updates or changes to the Interlock Proxy can be better planned such that it does not impact all applications / services. In other words, an outage to a service cluster will be localized to a failure of applications only within that service cluster.

- **Disadvantages:** Due to managing the proxy hosts as fixed (host mode, without ingress), there is a cost around slight reduction in reliability for greater flexibility. If using ingress, then you will need to manage the ports corresponding to each service cluster. In both the cases, there is a small overhead of managing the configurations in this mode of routing.

Regardless of which type of instance your frontend load balancer is directing traffic to, it's important to make sure the instances have an adequate network connection.

**Interlock Proxy Usage**

The following sections cover various use cases and the deployment syntax for the Interlock Proxy for HTTP routing, logging, monitoring, and setting up of secure application clusters (also known as service clusters).

**Routing in Interlock Proxy**

For services to be published using Interlock Proxy, they must contain, among other labels, at least two labels where the keys are `com.docker.lb.hosts` and `com.docker.lb.port`.

- The value of the label `com.docker.lb.hosts` should be the host that the service should serve. Optionally it can also be a comma-separated list of the hosts that the service should serve.
- The value of the label `com.docker.lb.port` should contain the port to use for the internal upstream communication with the service. Note that this port on the service need not be published externally.
- Optionally, the label `com.docker.lb.network` can be set to point to the name of the network that Interlock Proxy service needs to attach to for upstream connectivity. This label is required only if the service to be published using Interlock Proxy is attached to multiple overlay networks.

**Monitoring**

To monitor the Interlock Proxy from a frontend load balancer, set the load balancer to monitor the exposed Interlock Proxy ports on the cluster using a TCP health check. If Interlock Proxy is configured to listen on the default ports of 8080 and 8443, then the frontend load balancer would need to simply perform a TCP health check on all nodes that are in its pool.

**Interlock Proxy HA Considerations**

This section discusses a few usage considerations with regards to Interlock Proxy running in high-availability mode.
The ucp-interlock-proxy can be scaled up to have more replicas, and those replicas can be constrained to only those nodes that have high performance network interfaces. The additional benefit of this architecture is that it improves security by avoiding all application traffic from routing through the managers.

The following steps needed to accomplish this design:

1. Update the high performance nodes by using node labels to identify them:

   ```sh
docker node update --label-add nodetype=loadbalancer <node>
   ```

2. Constrain the Interlock Proxy service tasks to only run on the high performance nodes using the node labels. This is done by updating the ucp-interlock service configuration to deploy the interlock proxy service with the updated constraints in the ProxyConstraints array as explained below:

   - Retrieve the configuration that is currently being used for the ucp-interlock service and save it to a file (config.toml for example):

     ```sh
docker service inspect --format \\
ucp-interlock

docker config inspect --format '{%
Spec.Data %}' \\
$CURRENT_CONFIG_NAME > config.toml
   ```

   - Update the ProxyConstraints array in the config.toml file as shown below:

     ```toml
[Extensions]
[Extensions.default]
  ProxyConstraints = ["node.labels.com.docker.ucp.orchestrator.swarm==true", 
  "node.platform.os==linux", "node.labels.nodetype==loadbalancer"]
   ```

   - Create a new Docker config from the file that was just edited:

     ```sh
docker config create $NEW_CONFIG_NAME config.toml
   ```

   - Update the ucp-interlock service to start using the new configuration:

     ```sh
docker service update \\
--config-rm $CURRENT_CONFIG_NAME \\
--config-add source=$NEW_CONFIG_NAME,target=/config.toml \\
ucp-interlock
   ```

3. Configure the upstream load balancer to direct requests to only those high performance nodes on the Interlock Proxy ports. This ensures all traffic is directed to only these nodes.

For more information and steps see:

- Configure Layer 7 routing for production (https://docs.docker.com/ee/ucp/interlock/deploy/production/)
- Configure Layer 7 routing service (https://docs.docker.com/ee/ucp/interlock/config/)

## Interlock Proxy Usage Examples

This section explains the following types of applications, using all of the available networking modes for Interlock Proxy:

- HTTP Routing
To run through these examples showcasing service discovery and load balancing, the following are required:

1. A Docker client that has the UCP client bundle loaded and communicating with the UCP cluster.
2. DNS pointing to a load balancer sitting in front of your UCP cluster. If no load balancer can be used, then direct entries in your local hosts’ file to a host in your UCP cluster. If connecting directly to a host in your UCP cluster, connect over the published Interlock Proxy ports (8080 and 8443 by default).

---

**Note:** The repository for the sample application can be found on [GitHub](https://github.com/dockersuccess/counter-demo/blob/master/interlock-docker-compose.yml).

---

### Interlock Proxy Routing Example

Consider an example, standard 3-tier application that showcases service discovery and load balancing in Docker EE.

To deploy the application stack, run these commands with the UCP client bundle loaded:

```bash
$ wget https://raw.githubusercontent.com/dockersuccess/counter-demo/master/interlock-docker-compose.yml
$ DOMAIN=<domain-to-route> docker stack deploy -c interlock-docker-compose.yml http-example
```

Then access the example application at `http://<domain-to-route>/`.

### Websockets

The example also demonstrates support for websockets using the label `com.docker.lb.websocket_endpoints` with its value set to `/total` as shown in this section. The value can also be a comma-separated list of endpoints to configure to be upgraded for websockets.

This is the contents of the compose file if you just want to copy/paste into the UCP UI instead:
version: "3.3"

services:
  web:
    image: dockersuccess/webserver:latest
    environment:
      app_url: app:8080
    deploy:
      replicas: 2
      labels:
        com.docker.lb.hosts: ${DOMAIN:-app.dockerdemos.com}
        com.docker.lb.port: 2015
        com.docker.lb.websocket_endpoints: /total
    networks:
      - frontend

  app:
    image: dockersuccess/counter-demo:latest
    environment:
      ENVIRONMENT: ${env:-PRODUCTION}
    deploy:
      replicas: 5
      endpoint_mode: dnsrr
    networks:
      - frontend
      - backend

  db:
    image: redis:latest
    volumes:
      - data:/data
    networks:
      backend:

networks:
  frontend:
    driver: overlay
  backend:
    driver: overlay

volumes:
  data:

It is also possible to deploy through the UCP UI by going to Shared Resources -> Stacks -> Create Stack. Name the stack, change the Mode to Swarm Services, and copy/paste the above compose file into the open text field. Be sure to replace ${DOMAIN:-app.dockerdemos.com} with the correct domain name when deploying through the UI. Click on the Create button to deploy the stack.

Interlock Proxy Service Deployment Breakdown

The Interlock Proxy polls the Docker API for changes every 3 seconds (default), so once an application is deployed, Interlock Proxy polls for the new service and finds it at http://<domain-to-route>.
When the application stack is deployed using the compose file used as an example in this section, the following happens:

1. Three services are created — a service called web that is the frontend running a Caddy webserver, a service called app which contains the application logic, and another service db running a Redis database to store data.

2. Multiple overlay networks specific to the application stack are created called `<stack-name>_frontend` and `<stack-name>_backend`. The web and app services share the `<stack-name>_frontend` network, and the app and db services share the `<stack-name>_backend` network. In other words the web service cannot connect to the db service directly; it needs to connect to the app service, which is the only service that can connect to the db service.

3. The app service creates a DNS A Record of app on the `<stack-name>_frontend` network. This DNS record directs to the IP address(es) of the app containers.

   - The web service uses an environment variable app_url, the value of which is set to app:8080. This value points to the app service and does not need to change regardless of the stack name.
   - Similarly, the Redis task creates a DNS A Record of db on the `<stack-name>_backend` network.
   - The app service does not need to change for every stack when accessing Redis DB. It will always connect to db, and this can be hardcoded inside the app service. This is also independent of the stack name.

4. The frontend service web contains the labels com.docker.lb.hosts and com.docker.lb.port. The value for the label com.docker.lb.hosts is set to the domain(s) where the application needs to be made available. This can be conveniently set using the $DOMAIN environment variable. The value for the label com.docker.lb.port is set to the port where the web service is running, which is 2015 in this example. Because the web service is connected to a single overlay network `<stack-name>_frontend`, Interlock Proxy was able to attach itself to the `<stack-name>_frontend` network.

5. These labels above act as triggers for integration with Interlock Proxy. Once this service is available, Interlock Proxy will detect it and publish it. Any requests to the configured domain on the Interlock Proxy port (80 by default) will be forwarded to one of the web service replicas on the configured port.

6. The declared healthy state for the web service is 2 replicas, so 2 replica tasks are created. The two web service replicas are configured as upstreams. Interlock Proxy is responsible for balancing traffic across all published service replicas.

7. Interlock Proxy creates an entry so that all 5 frontend replicas are load balanced backed on the $DOMAIN environment variable that was passed for the stack deploy.

8. By doing a refresh of http://$DOMAIN in a web browser, the hit counter should increment with every request. It is load balancing across all of the frontend web service replicas.

9. Interlock Proxy polls every 3 seconds for Docker events, and it picks up the com.docker.lb.* labels on any newly created or updated services.

**Interlock Proxy Sticky Session Example**

**Cookie-Based Session Persistence**

Interlock Proxy has the ability to route to a specific backend service based on a named cookie. For example, if your application uses a cookie named JSESSIONID as the session cookie, you can persist connections to a specific service replica task by setting the value of the label com.docker.lb.sticky_session_cookie to JSESSIONID.

Why would cookie-based persistence need to be used? It can reduce load on the load balancer. The load balancer picks a certain instance in the backend pool and maintains the connection instead of having to re-route on new requests. Another use case could be for rolling deployments. When you bring in a new application
server into the load balancer pool you can avoid the "thundering herd" of new instances. Instead, it eases connections to the new instances into load balancing as sessions expire.

In general, sticky sessions are better suited for improving cache performance and lessening the load on certain aspects of the system. If you need to hit the same backend every time because your application is not using distributed storage, then you can run into more problems down the road when swarm mode reschedules your tasks. It's important to keep this in mind while using application cookie-based persistence.

To deploy the example service for sticky sessions, run these commands with the UCP client bundle loaded. Note the label `com.docker.lb.sticky_session_cookie` used to indicate the cookie to use to enable sticky sessions.

```bash
# Create an overlay network so that service traffic is isolated and secure
docker network create -d overlay demo

# Next create the service with the cookie to use for sticky sessions. Replace <domain-to-route> with a valid domain.
docker service create \ 
  --name demo \ 
  --network demo \ 
  --detach=false \ 
  --replicas=5 \ 
  --label com.docker.lb.hosts=<domain-to-route> \ 
  --label com.docker.lb.sticky_session_cookie=session \ 
  --label com.docker.lb.port=8080 \ 
  --env METADATA="demo-sticky" \ 
dockersuccess/docker-demo
```

Access the example application at `http://<domain-to-route>` in a browser tab or use `curl` as shown below:

```bash
Note: To test using a browser you need a DNS entry to `<domain-to-route>` with a valid domain pointing to a load balancer sitting in front of your UCP cluster. If no load balancer can be used, then direct entries in your local hosts' file to a host in your UCP cluster. If connecting directly to a host in your UCP cluster, you need to connect over the published Interlock Proxy ports (8080 and 8443 by default). The following examples use `demo.local` in lieu of `<domain-to-route>`.
```
In the output of `curl` above, the `Set-Cookie` attribute from the application is sent with subsequent requests, which are pinned to the same instance. The same `x-upstream-addr` is used for new requests.

**IP Hashing-Based Sticky Sessions**

Interlock Proxy also supports IP Hashing. In this mode a unique hash key is generated using the IP addresses of the source and destination. This hash is used by the load balancer (Interlock Proxy in our case) to allocate clients to a particular backend server.

Below is an example that uses IP hashing to enable sticky sessions. The label to use in this case is `com.docker.lb.ip_hash` with its value set to true.

Run the following commands with the UCP bundle loaded.

```bash
{{content}}gt; docker network create -d overlay demo 1se1glh749q1i4pw0kf26mf&x5

{{content}}gt; docker service create \
  --name demo \
  --network demo \
  --detach=false \
  --replicas=5 \
  --label com.docker.lb.hosts=demo.local \
  --label com.docker.lb.port=8080 \
  --label com.docker.lb.ip_hash=true \
  --env METADATA="demo-sticky" \
  dockersuccess/docker-demo
```

**Interlock Proxy HTTPS Example**
Interlock Proxy has support for routing using HTTPS / SSL. Both "SSL Termination" and "SSL Passthrough" can be setup to provide different configurations of load balancing encrypted web traffic.

In SSL Termination, SSL encrypted requests are decrypted by Interlock Proxy (the Load Balancer layer), and the unencrypted request is sent to the backend servers.

In SSL Passthrough, the SSL requests are sent as-is directly to the backend servers. The requests remain encrypted, and the backend servers become responsible for the decryption. This also implies that the backend servers have the necessary certificates and libraries to perform the decryption.

Secrets and certificates are almost always involved when dealing with encrypted communications. Before deploying the example application, generate the necessary certificates. You can also use one of several Certificate Authorities like Let's Encrypt to generate the certificates.

Here are some helpful commands to generate self-signed certificates to use with the application using `openssl`.

```bash
openssl req 
new 
-newkey rsa:4096 
-days 3650 
-nodes 
x509
-subj "/C=US/ST=SomeState/L=SomeCity/O=Interlock/CN=demo.local" 
-keyout demo.local.key 
-out demo.local.cert
```

Now that the certificates are generated, you can use the resulting files as input to create two Docker secrets using the following commands:

```bash
docker secret create demo.local.cert demo.local.cert ywn8ykn16cnnq4iz64um1pj7s
docker secret create demo.local.key demo.local.key e2xo036ukhfapip05c0sizf5w
```

The secrets are now encrypted in the cluster-wide key value store. The secrets are encrypted at rest and using TLS while in motion to nodes that need the secret. Secrets can only be viewed by the application that needs to use them.


Now create an overlay network so that service traffic is isolated and secure:
Interlock Proxy detects when the service is available and publish it. Once the tasks are running and the proxy service has been updated the application should be available via https://demo.local.

**Note:** To test using a browser you need a DNS entry to `<domain-to-route>` pointing to a load balancer sitting in front of your UCP cluster. If no load balancer can be used, then direct entries in your local hosts' file to a host in your UCP cluster. If connecting directly to a host in your UCP cluster, you would need to connect over the published Interlock Proxy ports (8080 and 8443 by default). The following examples use demo.local in lieu of `<domain-to-route>`.  

```bash
{{content}}gt; docker network create -d overlay demo
1se1glh749qIIiw0Kf26mfx5

{{content}}gt; docker service create \
--name demo \
--network demo \
--label com.docker.lb.hosts=demo.local \
--label com.docker.lb.port=8080 \
--label com.docker.lb.ssl_cert=demo.local.cert \
--label com.docker.lb.ssl_key=demo.local.key \
dockersuccess/docker-demo
6r0wiglf53bdpy6zesh1pzx
```
Since the certificate and key are stored securely within the Docker swarm, you can safely scale this service as well as the proxy service, and the Docker swarm will handle granting access to the credentials only as needed.

Interlock Optimizations for Production

Below are common optimizations for production deployments.

Use Service Clusters for Interlock Segmentation
Interlock service clusters allow Interlock to be segmented into multiple logical instances called “service clusters”, which have independently managed proxies. Application traffic only uses the proxies for a specific service cluster, allowing the full segmentation of traffic. Each service cluster only connects to the networks using that specific service cluster, which reduces the number of overlay networks to which proxies connect. Because service clusters also deploy separate proxies, this also reduces the amount of churn in LB configs when there are service updates.

**Minimizing Number of Overlay Networks**

The Interlock proxy containers will connect to the overlay network of every Swarm service. Having many networks connected to Interlock adds incremental delay when Interlock updates its load balancer configuration. Each network connected to Interlock generally adds 1-2 seconds of update delay. With many networks the Interlock update delay will cause the LB config to be out of date for too long which can cause traffic to be dropped.

Minimizing the number of overlay networks that Interlock connects to can be accomplished in two ways:

- Reduce the number of networks. If the architecture permits it, applications can be grouped together to use the same networks.
- Use Interlock Service Clusters. By segmenting Interlock, service clusters also segment which networks are connected to Interlock, reducing the number of networks that each proxy is connected to.

**Use Interlock VIP Mode**

VIP Mode can be used to reduce the impact of application updates on the Interlock proxies. It utilizes the Swarm L4 load balancing VIPs instead of individual task IPs to load balance traffic to a more stable internal endpoint. This prevents the proxy LB configs from changing for most kinds of app service updates reducing churn for Interlock. The major tradeoff is that the following features are not supported in VIP mode:

- Sticky Sessions
- Canary Deployments

The following features are still supported in VIP mode:

- Host & Context Routing
- Context Root Rewrites
- Interlock TLS Termination
- TLS Passthrough
- Service Clusters

For additional tips on optimizing interlock for applications, see Optimizing Interlock for applications ([https://docs.docker.com/ee/ucp/interlock/architecture/#optimizing-interlock-for-applications](https://docs.docker.com/ee/ucp/interlock/architecture/#optimizing-interlock-for-applications))

**Summary**

The ability to scale and discover services in Docker is now easier than ever. With the service discovery and load balancing features built into Docker, engineers can spend less time creating these types of supporting capabilities on their own and more time focusing on their applications. Instead of creating API calls to set DNS for service discovery, Docker automatically handles it for you. If an application needs to be scaled, Docker takes care of adding it to the load balancer pool. By leveraging these features, organizations can deliver highly available and resilient applications in a shorter amount of time.