Issue

Intermittent connection timeouts or networking errors on Docker swarm multi-host overlay networks due to IPVS connection tracking time out. Symptoms of this issue may include:

- timeout exception from java based application gateway
- proxy errors

An example of a community issue with this issue is moby/moby#31208 (https://github.com/moby/moby/issues/31208).

This issue typically affects application gateways or proxies. It occurs when a TCP client consuming a swarm service VIP is completely idle for longer than 15 minutes. The problem state is evidenced by an open TCP socket on the client without a corresponding IPVS connection tracking entry.

Root Cause

Swarm service VIPs and ports published via the swarm ingress network use the IPVS L4 load balancer built into the Linux kernel to distribute requests across swarm service tasks. In order to maintain a clean connection tracking table IPVS, removes idle connections after 15 minutes. You can check the timers for a particular network namespace using `IPVSadm`. The defaults are as follows (units in seconds):

```
$ ipvsadm -l --timeout
Timeout (tcp tcpfin udp): 900 120 300
```

IPVS timeouts are configurable on a per-network-namespace basis. The Docker Engine does not currently have the ability to modify the settings from the default.

Resolution

Docker-internal enhancement request FIELD-232 has been opened to make the IPVS timeout configurable for swarm overlay networking. If you are a Docker Enterprise customer interested in this enhancement please contact your account team or open a support case to express interest.

There are several operational accommodations for this issue, outlined below. Options that do not require application modification are listed first.

Resolution without Application Change -- Service Endpoint Mode dnsrr

You can avoid this issue by reconfiguring swarm routing to stop using IPVS. This is accomplished on a per-service basis by changing the service endpoint mode to `dnsrr`. Since the IPVS load balancer is on the client side, this change only strictly needs to be performed on affected TCP *server* services.

Syntax:

- Compose syntax for endpoint mode dnsrr (https://docs.docker.com/engine/reference/commandline/service_create/#attach-a-service-to-an-existing-network---network)
- Service create syntax: `docker service create --endpoint-mode [...]`

Caveats:

- Services that publish ports on the ingress network cannot be set to endpoint mode dnsrr.
- TCP client applications that cache DNS interact poorly with dnsrr mode because they don't learn when a server task goes down until the name cache expires. If possible, disable DNS caching on TCP client applications, and test for expected behavior during TCP server service update before deploying this change to production.
- Endpoint mode cannot be changed on an existing service. The simplest way to make this change on an existing service is to remove the service and redeploy it with point mode dnsrr, incurring a brief interruption to service availability. To redeploy with no downtime:
  1. Deploy a copy of the service with endpoint mode dnsrr and a network alias set to the original service name.
     - Compose syntax for network alias (https://docs.docker.com/compose/compose-file/#aliases)
     - Service create syntax for network alias (https://docs.docker.com/engine/reference/commandline/service_create/#attach-a-service-to-an-existing-network---network)
  2. Remove the original service.
  3. Redeploy the original service with endpoint mode dnsrr.
  4. Remove the copy.

Resolution by Application Modification

1. Enable keepalive at the TCP or application network layers:
   - **TCP Keepalive.** In Linux TCP keepalive parameters can be set when opening the socket. The default TCP keepalive settings (sysctls) are:
     - `net.ipv4.tcp_keepalive_time = **7200**`: seconds idle before sending the first probe
     - `net.ipv4.tcp_keepalive_intvl = 75`: seconds between probes
     - `net.ipv4.tcp_keepalive_probes = 9`: number of probes to fail before considering the session failed

     The following are the required socket options:

     ```
      * `SO_KEEPALIVE = 1` -- enable keepalive.  Keep alive *must* be enabled by the application upon socket creation. There is no OS-level configuration to globally enable TCP keepalive in Linux.
      * `TCP_KEEPIDLE = 600` -- send first probe after 10 minutes idle.  This can be set on the host via sysctl `ipv4.tcp_keepalive_time`.  However, containers on kernel 4.13 and later will not inherit this sysctl from the host, and as of EE Engine 18.09 setting sysctls with swarm mode services is not yet supported.
     ```

   - **Application keepalive.** Reconfigure or modify the application to send some data periodically (heartbeat) over otherwise quiet TCP sessions.
2. Reconfigure or modify affected TCP client applications to close idle TCP sessions prior to the 15 minute timeout.

Diagnostic Steps

On Docker EE Engine 17.06* you can use the following commands to check if this problem that exists on a particular tcp client container. Issue the commands from a bash shell on the container host.

* Load balancing was moved to a dedicated namespace on EE Engine 18.09.

1. Set a shell variable to the name or ID of the affected TCP client container (replace <^>client<^^> with the name or ID of the container):

```
CONTAINER_NAME=<^>client<^^>
```

2. Set a shell variable to the currently running UCP version:

```
UCP_VERSION=$(docker container inspect ucp-proxy --format '{{index (split  .Config.Image ":") 1}}')
```

3. We will use `docker run --network container:[...]` to start a container network diagnostic tools in the network namespace of the client container. Define a shell alias to shorten the `docker run` command:

```
alias client="docker container run -i --rm --cap-add NET_ADMIN --network
container:${CONTAINER_NAME} docker/ucp-dsinfo:${UCP_VERSION}"
```

4. (optional) List the local and remote address for all established TCP sockets using `ss`:

```
client ss -nt state established | awk 'NR > 1 {print $3,$4}'
```

5. (optional) List all `ESTABLISHED` conntrack sessions in the same format as `ss` (local then remote address):

```
client bash << "EOCOMMAND"
  conntrack -L -p tcp --state ESTABLISHED 2>/dev/null |
    awk '/^tcp.*ESTABLISHED/ {
         split($5,src,"=");
         split($6,dst,"=");
         split($7,sport,"=");
         split($8,dport,"=");
         print src[2]":"sport[2],dst[2]":"dport[2]}'
EOCOMMAND
```

6. Compare active sessions with conntrack state using `comm` to list established TCP sockets that are not in conntrack.
   If there is any output, it will indicate TCP sessions affected by this issue.

```
client bash << "EOCOMMAND"
  comm -2 -3 \
    <( ss -nt state established |
          awk 'NR > 1 {print $3,$4}' |
          sort ) \
    <( conntrack -L -p tcp --state ESTABLISHED 2>/dev/null |
         awk '/^tcp.*ESTABLISHED/ {
              split($5,src,"=");
              split($6,dst,"=");
              split($7,sport,"=");
              split($8,dport,"=");
              print src[2]":"sport[2],dst[2]":"dport[2]}' |
         sort )
EOCOMMAND
```