NGINX LOAD BALANCING – TCP AND UDP LOAD BALANCER

This chapter describes how to use NGINX Plus and NGINX Open Source to proxy and load balance TCP and UDP traffic.

Table of Contents

Introduction

Load balancing refers to efficiently distributing network traffic across multiple backend servers.

In Release 5 and later, NGINX can proxy and load balance TCP traffic. TCP (Transmission Control Protocol) is the protocol for many popular applications and services, such as LDAP, MySQL, and RTMP.

In Release 9 and later, NGINX can proxy and load balance UDP traffic. UDP (User Datagram Protocol) is the protocol for many popular nontransactional applications, such as DNS, syslog, and RADIUS.

To load balance HTTP traffic, refer to the HTTP Load Balancing article.

Prerequisites

- Latest NGINX Open Source built with the `--with-stream` configuration flag, or latest NGINX Plus (no extra build steps required)
- An application, database, or service that communicates over TCP or UDP
- Upstream servers, each running the same instance of the application, database, or service

Configuring Reverse Proxying

First, you will need to configure *reverse proxying* so that NGINX can forward TCP connections or UDP datagrams from clients to an upstream group or a proxied server.

Open the NGINX configuration file and perform the following steps:

1. Create a top-level `stream {}` block:

```
stream {
...
}
```

2. Define one or more `server {}` configuration blocks for each virtual server in the top-level `stream {}` context.
3. Within the `server {}` configuration block for each server, include the `listen` directive to define the *IP address* and/or *port* on which the server listens. For UDP traffic, also include the `udp` parameter. TCP is the default protocol for the `stream` context, so there is no `tcp` parameter to the `listen` directive:

```
stream {
    server {
        listen 12345;
        ...
    }
    server {
        listen 53 udp;
        ...
    }
    ...
}
```

4. Include the `proxy_pass` directive to define the proxied server or an upstream group to which the server forwards traffic:

```
stream {
    server {
        listen     12345;

        #TCP traffic will be proxied to the "stream_backend" upstream group
        proxy_pass stream_backend;
    }

    server {
        listen     12346;

        #TCP traffic will be proxied a proxied server
        proxy_pass backend.example.com:12346;
    }

    server {
        listen     53 udp;

        #UDP traffic will be proxied to the "dns_servers" upstream group
        proxy_pass dns_servers;
    }
    ...
}
```

5. Optionally, if your proxy server has several network interfaces, you can configure NGINX to choose a source IP address for connecting to an upstream server. This may be useful if a proxied server behind NGINX is configured to accept connections from particular IP networks or IP address ranges.

   Specify the `proxy_bind` directive and the IP address of the necessary network interface:

```
stream {
    ...
    server {
        listen     127.0.0.1:12345;
        proxy_pass backend.example.com:12345;
        proxy_bind 127.0.0.1:12345;
    }
}
```

6. Optionally, you can tune the size of two in-memory buffers where NGINX can put data from both the client and upstream connections. If there is a small volume of data, the buffers can be reduced which may save memory resources. If there is a large volume of data, the buffer size can be increased to reduce the number of socket read/write operations. As soon as data is received on one connection, NGINX reads it and forwards it over the other connection. The buffers are controlled with the `proxy_buffer_size` directive:

```
stream {
    ...
    server {
        listen            127.0.0.1:12345;
        proxy_pass        backend.example.com:12345;
        proxy_buffer_size 16k;
    }
}
```

Configuring TCP or UDP Load Balancing

To configure load balancing:

1. Create a group of servers, or an *upstream group* whose traffic will be load balanced. Define one or more `upstream {}` configuration blocks in the top-level `stream {}` context and set the name for the upstream group, for example, `stream_backend` for TCP servers and `dns_servers` for UDP servers:

```
stream {
    upstream stream_backend {
        ...
    }

    upstream dns_servers {
        ...
    }
    ...
}
```

Make sure that the name of the upstream group is referenced in the `proxy_pass` directive configured earlier.

2. Populate the upstream group with *upstream servers*. Within the `upstream {}` block, add a `server` directive for each upstream server, specifying its IP address or hostname (which can resolve to multiple IP addresses) and an *obligatory* port number. Note that you do not define the protocol for each server, because that is defined for the entire upstream group by the parameter you include on the `listen` directive in the `server` block, which you have created earlier.

```
stream {
    upstream stream_backend {
        server backend1.example.com:12345;
```

```
            server backend2.example.com:12345;
            server backend3.example.com:12346;
            ...
        }
        upstream dns_servers {
            server 192.168.136.130:53;
            server 192.168.136.131:53;
            ...
        }
        ...
    }
```

3. Configure a *load balancing method* used by the upstream group. You can specify one of the following methods:

    - `round-robin` – By default, NGINX uses the round-robin algorithm to load balance traffic, directing it sequentially to the servers in the configured upstream group. Because it is the default method, there is no `round-robin` directive; simply create an `upstream` configuration block in the top-level `stream` context and add the `server` directives as described in the previous step.
    - `least_conn` – NGINX selects the server with the smaller number of current active connections.
    - `least_time` – NGINX selects the server with the lowest average latency and the least number of active connections. The lowest average latency is calculated based on which of the following parameters is included on the `least_time` directive:

        - `connect` – Time to connect to the upstream server
        - `first_byte` – Time to receive the first byte of data
        - `last_byte` – Time to receive the full response from the server

    ```
    upstream stream_backend {
        least_time first_byte;

        server backend1.example.com:12345;
        server backend2.example.com:12345;
        server backend3.example.com:12346;
    }
    ```

    - `hash` – NGINX selects the server based on a user-defined key, for example, a source IP address (`$remote_addr`):

    ```
    upstream stream_backend {
        hash $remote_addr;

        server backend1.example.com:12345;
        server backend2.example.com:12345;
        server backend3.example.com:12346;
    }
    ```

The hash load balancing method is also used to configure *session persistence*. As the hash function is based on client IP address, connections from a given client are always passed to the same server unless the server is down or otherwise unavailable. Specify an optional `consistent` parameter to apply the *ketama* consistent hashing method:

```
hash $remote_addr consistent;
```

4. Optionally, for each upstream server specify server-specific parameters including maximum number of connections, server weight, and so on:

```
upstream stream_backend {
    hash   $remote_addr consistent;
    server backend1.example.com:12345 weight=5;
    server backend2.example.com:12345;
    server backend3.example.com:12346 max_conns=3;
}

    upstream dns_servers {
        least_conn;
        server 192.168.136.130:53;
        server 192.168.136.131:53;
        ...
    }
```

An alternative approach is to proxy traffic to a single server instead of an upstream group. If you identify the server by hostname, and configure the hostname to resolve to multiple IP addresses, then NGINX load balances traffic across the IP addresses using the round-robin algorithm. In this case, you *must* specify the server's port number in the `proxy_pass` directive and *must not* specify the protocol before IP address or hostname:

```
stream {
    ...
    server {
        listen     12345;
        proxy_pass backend.example.com:12345;
    }
}
```

Passive Health Monitoring

If an attempt to connect to an upstream server times out or results in an error, NGINX Open Source or NGINX Plus can mark the server as unavailable and stop sending requests to it for a defined amount of time. To define the conditions under which NGINX considers an upstream server unavailable, include the following parameters to the `server` directive

- `fail_timeout` – The amount of time within which a specified number of connection attempts must fail for the server to be considered unavailable. Also, the amount of time that NGINX considers the server unavailable after marking it so.
- `max_fails` – The number of failed attempts that happen during the specified time for NGINX to consider the server unavailable.

The default values are **10** seconds and **1** attempt. So if a connection attempt times out or fails at least once in a 10-second period, NGINX marks the server as unavailable for 10 seconds. The example shows how to set these parameters to 2 failures within 30 seconds:

```
upstream stream_backend {
    server backend1.example.com:12345 weight=5;
    server backend2.example.com:12345 max_fails=2 fail_timeout=30s;
    server backend3.example.com:12346 max_conns=3;
}
```

Active Health Monitoring

Health checks can be configured to test a wide range of failure types. For example, NGINX Plus can continually test upstream servers for responsiveness and avoid servers that have failed.

How It Works

NGINX Plus sends special health-check requests to each upstream server and checks for a response that satisfies certain conditions. If a connection to the server cannot be established, the health check fails, and the server is considered unhealthy. NGINX Plus does not proxy client connections to unhealthy servers. If several health checks are defined for a group of servers, the failure of any one check is enough for the corresponding server be considered unhealthy.

Prerequisites

- You have configured an upstream group of servers in the `stream` context, for example:

```
stream {
    upstream stream_backend {

    server backend1.example.com:12345;
    server backend2.example.com:12345;
    server backend3.example.com:12345;
   }
}
```

- You have configured a server that passes traffic (in this case, TCP connections) to the server group:

```
server {
    listen     12345;
```

```
        proxy_pass stream_backend;
    }
```

Basic Configuration

1. Specify a *shared memory zone* – a special area where the NGINX Plus worker processes share state information about counters and connections. Add the `zone` directive to the upstream server group and specify the *zone name* and the *amount of memory*:

```
stream {
    upstream stream_backend {

        zone   stream_backend 64k;
        server backend1.example.com:12345;
        server backend2.example.com:12345;
        server backend3.example.com:12345;
   }
}
```

2. Enable health checks for servers in the upstream group. Add the `health_check` and `health_check_timeout` directives to the server that proxies connections to the upstream group:

```
server {
    listen       12345;
    proxy_pass    stream_backend;
    health_check;
    health_check_timeout 5s;
}
```

The **`health_check`** directive enables the health check functionality, while **`health_check_timeout`** overrides the `proxy_timeout` value for health checks, as for health checks this timeout needs to be significantly shorter.

To enable health checks for UDP traffic, in the `health_check` directive specify the `udp` parameter that enables health checks for UDP, and the `match=` parameter with a name of a corresponding **`match`** block that contains tests for verifying server responses (see Fine-Tuning UDP Health Checks):

```
server {
    listen       5053;
    proxy_pass   dns_servers;
    health_check udp match=dns;
    health_check_timeout 5s;
}
```

Fine-Tuning Health Checks

By default, NGINX Plus tries to connect to each server in an upstream server group every 5 seconds. If the connection cannot be established, NGINX Plus considers the health check failed, marks the server as unhealthy, and stops forwarding client connections to the server.

To change the default behavior, include parameters to the `health_check` directive:

- `interval` – How often (in seconds) NGINX Plus sends health check requests (default is 5 seconds)
- `passes` – Number of consecutive health checks the server must respond to to be considered healthy (default is 1)
- `fails` – Number of consecutive health checks the server must fail to respond to to be considered unhealthy (default is 1)

```
server {
    listen       12345;
    proxy_pass   stream_backend;
    health_check interval=10 passes=2 fails=3;
}
```

In the example, the time between TCP health checks is increased to `10` seconds, the server is considered unhealthy after `3` consecutive failed health checks, and the server needs to pass `2` consecutive checks to be considered healthy again.

By default, NGINX Plus sends health check messages to the port specified by the `server` directive in the `upstream` block. You can specify another port for health checks, which is particularly helpful when monitoring the health of many services on the same host. To override the port, specify the `port` parameter of the `health_check` directive:

```
server {
    listen       12345;
    proxy_pass   stream_backend;
    health_check port=8080;
}
```

Fine-Tuning Health Checks with the match Configuration Block

You can verify server responses to health checks by configuring a number of tests. These tests are defined with the `match {}` configuration block placed in the `stream {}` context. Specify the `match {}` block and set its name, for example, `tcp_test`:

```
stream {
    ...
    match  tcp_test {
        ...
    }
}
```

Then refer to the block from the `health_check` directive by including the `match` parameter and the name of the `match` block:

```
stream {
    server {
        listen       12345;
        health_check match=tcp_test;
        proxy_pass   stream_backend;
    }
}
```

The conditions or tests under which a health check succeeds are set with `send` and `expect` parameters:

- `send` – The text string or hexadecimal literals ("/x" followed by two hex digits) to send to the server
- `expect` – Literal string or regular expression that the data returned by the server needs to match

These parameters can be used in different combinations, but no more than one `send` and one `expect` parameter can be specified at a time. The parameters configuration also depends on the protocol used (TCP or UDP).

Fine-Tuning TCP Health Checks

To fine-tune health checks for TCP, you can combine `send` and `expect` parameters as follows:

- If no `send` or `expect` parameters are specified, the ability to connect to the server is tested.
- If the `expect` parameter is specified, the server is expected to unconditionally send data first:

```
match pop3 {
    expect ~* "\+OK";
}
```

- If the `send` parameter is specified, it is expected that the connection will be successfully established and the specified string will be sent to the server:

```
match pop_quit {
    send QUIT;
}
```

- If both the `send` and `expect` parameters are specified, then the string from the `send` parameter must match the regular expression from the `expect` parameter:

```
stream {
    upstream   stream_backend {
        zone   upstream_backend 64k;
        server backend1.example.com:12345;
    }
```

```
        match http {
            send      "GET / HTTP/1.0\r\nHost: localhost\r\n\r\n";
            expect ~* "200 OK";
        }

        server {
        listen       12345;
        health_check match=http;
        proxy_pass   stream_backend;
        }
    }
```

The example shows that in order for a health check to pass, the HTTP request must be sent to the server, and the expected result from the server contains "200 OK" to indicate a successful HTTP response.

Fine-Tuning UDP Health Checks

To fine-tune health checks for UDP, you should specify both `send` and `expect` parameters:

Example for NTP:

```
match ntp {
   send
\xe3\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00

   expect ~* \x24;
}
```

Example for DNS:

```
match dns {
   send
\x00\x2a\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x03\x73\x74\x6c\x04\x75\x6d\x73\x6c\x03\x65\x64\x75\x00

   expect ~* "health.is.good";
}
```

On-the-Fly Configuration

Upstream server groups can be easily reconfigured on-the-fly using a simple HTTP interface. Using this interface, you can view all servers in an upstream group or a particular server, modify server parameters, and add or remove upstream servers.

To enable on-the-fly configuration:

1. Create the top-level `http {}` block or make sure it is present in your configuration:

```
http {
    ...
}
```

2. Create a location for configuration requests, for example, *upstream_conf*:

```
http {

    server {
        location /upstream_conf {
            ...
        }
    }
}
```

3. In this location specify the `upstream_conf` directive – a special handler that can be used to inspect and reconfigure upstream groups in NGINX Plus:

```
http {

    server {
        location /upstream_conf {
            upstream_conf;
            ...
        }
    }
}
```

4. Limit access to this location with `allow` and `deny` directives:

```
http {

    server {
        location /upstream_conf {
            upstream_conf;
            allow 127.0.0.1; # permit access from localhost
            deny  all;       # deny access from everywhere else
        }
    }
}
```

5. Create a *shared memory zone* for the group of upstream servers so that all worker processes can use the same configuration. To do this, in the top-level `stream {}` block, find the target

upsteam group, add the zone directive to the upstream server group and specify the *zone name* and the *amount of memory*:

```
stream {
    upstream stream_backend {
        zone backend 64k;
        ...
    }
}
```

On-the-Fly Configuration Example

```
stream {
    ...
    # Configuration of an upstream server group
    upstream appservers {
        zone appservers 64k;

        server appserv1.example.com:12345 weight=5;
        server appserv2.example.com:12345 fail_timeout=5s;

        server backup1.example.com:12345 backup;
        server backup2.example.com:12345 backup;
    }

    server {
        # Server that proxies connections to the upstream group
        proxy_pass appservers;
        health_check;
    }
}

http {
    ...
    server {
        # Location for configuration requests
        location /upstream_conf {
            upstream_conf;
            allow 127.0.0.1;
            deny  all;
        }
    }
}
```

Here, access to the location is allowed only from the `127.0.0.1` address. Access from all other IP addresses is denied.

To pass a configuration command to NGINX, send an HTTP request by any method. The request must have an appropriate URI to get into the location that includes the `upstream_conf` directive. The request must include the `upstream` argument set to the name of the server group.

For example, to view all backup servers (indicated by the `backup` argument) in the server group, send:

```
http://127.0.0.1/upstream_conf?stream=&upstream=appservers&backup=
```

To add a new server to the server group, send a request with `add` and `server` arguments:

```
http://127.0.0.1/upstream_conf?
stream=&add=&upstream=appservers&server=appserv3.example.com:12345&weight=2&max_fails=3
```

To remove a server from the server group, send a request with the `remove` argument and the `id` argument identifying the server:

```
http://127.0.0.1/upstream_conf?stream=&remove=&upstream=appservers&id=2
```

To modify a parameter for a specific server, send a request with the `id` argument identifying the server and the name of the parameter:

```
http://127.0.0.1/upstream_conf?stream=&upstream=appservers&id=2&down=
```

Example of TCP and UDP Load Balancing Configuration

This is a configuration example of TCP and UDP load balancing with NGINX:

```
stream {
        upstream stream_backend {
            least_conn;
            server backend1.example.com:12345 weight=5;
            server backend2.example.com:12345 max_fails=2 fail_timeout=30s;
            server backend3.example.com:12346 max_conns=3;
        }

        upstream dns_servers {
            least_conn;
            server 192.168.136.130:53;
            server 192.168.136.131:53;
            server 192.168.136.132:53;
        }

        server {
            listen        12345;
            proxy_pass    backend;
            proxy_timeout 3s;
            proxy_connect_timeout 1s;

        }

        server {
            listen     53 udp;
            proxy_pass dns_servers;
        }

        server {
            listen     12346;
            proxy_pass backend.example.com:12346;
        }
    }
```

In this example, there are three servers, each defined in a `server` block, and two named groups of three upstream servers. Each upstream server inside a group host the same content defined in the `upstream` block. All TCP and UDP proxy-related functionality is configured inside the `stream` block just like the `http` block for HTTP requests.

The first server listens on port `12345` and proxies all TCP connections to a group of servers named `backend`. Note that the `proxy_pass` directive defined in the context of the *stream* module must not contain a protocol. Two optional timeout parameters are specified: the `proxy_connect_timeout` directive sets the timeout required for establishing a connection with a server that belongs to the `backend` group. The `proxy_timeout` directive sets a timeout used after proxying to one of the servers in the `backend` group has started.

The second server listens on port `53` and proxies all UDP datagrams (the `UDP` parameter of the `listen` directive) to an upstream group called `dns_servers`. If the parameter is not specified, the socket will listen for TCP connections.

Each upstream group consists of three servers (`backend1` – `backend3` and `192.168.136.130` – `192.168.136.133`) that host the same content. Each server name is followed by the obligatory port number. Connections are distributed among the servers according to the `least-connected` load balancing method: a connection goes to the server with the fewest number of active connections.

The third virtual server listens on port `12346` and proxies TCP connections to the `backend4` physical server, which can resolve to several IP addresses that are load balanced with the round-robin method.

See Also

- TCP Load Balancing with NGINX 1.9.0 and NGINX Plus R6
- Announcing UDP Load Balancing in NGINX and NGINX Plus
- Use case: MySQL High Availability with NGINX Plus and Galera Cluster

TRY NGINX PLUS FOR FREE

ASK US A QUESTION

Sign up for newsletter

Products

NGINX Plus
Dynamic modules
Pricing
Technical specifications
Compare versions
NGINX Amplify

Solutions

Microservices
ADC / Load balancing
Web & mobile performance
Cloud migration
Security
API gateway

Resources

Admin Guide
Blog
Library
Webinars
Events
Case studies
FAQ
Community wiki

Company

About NGINX
Careers
Leadership
Press

Partners

Certified module program

Support

NGINX Plus support
Professional Services
Training

Connect With Us

Stay in the Loop

enter your email