Overview

Load balancing across multiple application instances is a commonly used technique for optimizing resource utilization, maximizing throughput, reducing latency, and ensuring fault-tolerant configurations.

Watch the F5 NGINX Plus for Load Balancing and Scaling webinar on demand for a deep dive on techniques that NGINX users employ to build large-scale, highly available web services.

NGINX and NGINX Plus can be used in different deployment scenarios as a very efficient HTTP load balancer.

Proxying HTTP Traffic to a Group of Servers

To start using NGINX Plus or NGINX Open Source to load balance HTTP traffic to a group of servers, first you need to define the group with the `upstream` directive. The directive is placed in the `http` context.

Servers in the group are configured using the `server` directive (not to be confused with the `server` block that defines a virtual server running on NGINX). For example, the following configuration defines a group named **backend** and consists of three server configurations (which may resolve in more than three actual servers):

Copy

```
http {
    upstream backend {
        server backend1.example.com weight=5;
        server backend2.example.com;
        server 192.0.0.1 backup;
    }
}
```

To pass requests to a server group, the name of the group is specified in the `proxy_pass` directive (or the `fastcgi_pass`, `memcached_pass`, `scgi_pass`, or `uwsgi_pass` directives for those protocols.) In the next example, a virtual server running on NGINX passes all requests to the **backend** upstream group defined in the previous example:

Copy

```
server {
    location / {
        proxy_pass http://backend;
    }
}
```

The following example combines the two snippets above and shows how to proxy HTTP requests to the **backend** server group. The group consists of three servers, two of them running instances of the same application while the third is a backup server. Because no load-balancing algorithm is specified in the `upstream` block, NGINX uses the default algorithm, Round Robin:

Copy

```
http {
    upstream backend {
        server backend1.example.com;
        server backend2.example.com;
        server 192.0.0.1 backup;
    }

    server {
        location / {
            proxy_pass http://backend;
        }
    }
}
```

Choosing a Load-Balancing Method

NGINX Open Source supports four load-balancing methods, and NGINX Plus adds two more methods:

1. Round Robin – Requests are distributed evenly across the servers, with server weights taken into consideration. This method is used by default (there is no directive for enabling it):

Copy

```
upstream backend {
   # no load balancing method is specified for Round Robin
   server backend1.example.com;
   server backend2.example.com;
}
```

2. Least Connections – A request is sent to the server with the least number of active connections, again with server weights taken into consideration:

Copy

```
upstream backend {
    least_conn;
    server backend1.example.com;
    server backend2.example.com;
}
```

3. IP Hash – The server to which a request is sent is determined from the client IP address. In this case, either the first three octets of the IPv4 address or the whole IPv6 address are used to calculate the hash value. The method guarantees that requests from the same address get to the same server unless it is not available.

Copy

```
upstream backend {
    ip_hash;
    server backend1.example.com;
    server backend2.example.com;
}
```

If one of the servers needs to be temporarily removed from the load-balancing rotation, it can be marked with the down parameter in order to preserve the current hashing of client IP addresses. Requests that were to be processed by this server are automatically sent to the next server in the group:

Copy

```
upstream backend {
    server backend1.example.com;
    server backend2.example.com;
    server backend3.example.com down;
}
```

4. Generic Hash – The server to which a request is sent is determined from a user-defined key which can be a text string, variable, or a combination. For example, the key may be a paired source IP address and port, or a URI as in this example:

Copy

```
upstream backend {
    hash $request_uri consistent;
    server backend1.example.com;
    server backend2.example.com;
}
```

The optional consistent parameter to the `hash` directive enables ketama consistent-hash load balancing. Requests are evenly distributed across all upstream servers based on the user-defined hashed key value. If an upstream server is added to or removed from an upstream group, only a few keys are remapped which minimizes cache misses in the case of load-balancing cache servers or other applications that accumulate state.

5. Least Time (NGINX Plus only) – For each request, NGINX Plus selects the server with the lowest average latency and the lowest number of active connections, where the lowest average latency is calculated based on which of the following parameters to the `least_time` directive is included:

   - `header` – Time to receive the first byte from the server
   - `last_byte` – Time to receive the full response from the server
   - `last_byte inflight` – Time to receive the full response from the server, taking into account incomplete requests

Copy

```
upstream backend {
    least_time header;
    server backend1.example.com;
    server backend2.example.com;
}
```

6. Random – Each request will be passed to a randomly selected server. If the two parameter is specified, first, NGINX randomly selects two servers taking into account server weights, and then chooses one of these servers using the specified method:
   - least_conn – The least number of active connections
   - least_time=header (NGINX Plus) – The least average time to receive the response header from the server ($upstream_header_time)
   - least_time=last_byte (NGINX Plus) – The least average time to receive the full response from the server ($upstream_response_time)

   Copy

```
upstream backend {
    random two least_time=last_byte;
    server backend1.example.com;
    server backend2.example.com;
    server backend3.example.com;
    server backend4.example.com;
}
```

   The **Random** load balancing method should be used for distributed environments where multiple load balancers are passing requests to the same set of backends. For environments where the load balancer has a full view of all requests, use other load balancing methods, such as round robin, least connections and least time.

Note: When configuring any method other than Round Robin, put the corresponding directive (hash, ip_hash, least_conn, least_time, or random) above the list of server directives in the upstream {} block.

Server Weights

By default, NGINX distributes requests among the servers in the group according to their weights using the Round Robin method. The weight parameter to the server directive sets the weight of a server; the default is 1:

Copy

```
upstream backend {
    server backend1.example.com weight=5;
    server backend2.example.com;
    server 192.0.0.1 backup;
}
```

In the example, **backend1.example.com** has weight 5; the other two servers have the default weight (1), but the one with IP address 192.0.0.1 is marked as a backup server and does not receive requests unless both of the other servers are unavailable. With this configuration of weights, out of every 6 requests, 5 are sent to **backend1.example.com** and 1 to **backend2.example.com**.

Server Slow-Start

The server slow-start feature prevents a recently recovered server from being overwhelmed by connections, which may time out and cause the server to be marked as failed again.

In NGINX Plus, slow-start allows an upstream server to gradually recover its weight from 0 to its nominal value after it has been recovered or became available. This can be done with the slow_start parameter to the server directive:

Copy

```
upstream backend {
    server backend1.example.com slow_start=30s;
    server backend2.example.com;
    server 192.0.0.1 backup;
}
```

The time value (here, 30 seconds) sets the time during which NGINX Plus ramps up the number of connections to the server to the full value.

Note that if there is only a single server in a group, the max_fails, fail_timeout, and slow_start parameters to the server directive are ignored and the server is never considered unavailable.

Enabling Session Persistence

Session persistence means that NGINX Plus identifies user sessions and routes all requests in a given session to the same upstream server.

NGINX Plus supports three session persistence methods. The methods are set with the sticky directive. (For session persistence with NGINX Open Source, use the hash or ip_hash directive as described above.)

- Sticky cookie – NGINX Plus adds a session cookie to the first response from the upstream group and identifies the server that sent the response. The client's next request contains the cookie value and NGINX Plus route the request to the upstream server that responded to the first request:

  Copy

  ```
  upstream backend {
      server backend1.example.com;
      server backend2.example.com;
      sticky cookie srv_id expires=1h domain=.example.com path=/;
  }
  ```

  In the example, the `srv_id` parameter sets the name of the cookie. The optional `expires` parameter sets the time for the browser to keep the cookie (here, `1` hour). The optional `domain` parameter defines the domain for which the cookie is set, and the optional `path` parameter defines the path for which the cookie is set. This is the simplest session persistence method.

- Sticky route – NGINX Plus assigns a "route" to the client when it receives the first request. All subsequent requests are compared to the `route` parameter of the `server` directive to identify the server to which the request is proxied. The route information is taken from either a cookie or the request URI.

  Copy

  ```
  upstream backend {
      server backend1.example.com route=a;
      server backend2.example.com route=b;
      sticky route $route_cookie $route_uri;
  }
  ```

- Sticky learn method – NGINX Plus first finds session identifiers by inspecting requests and responses. Then NGINX Plus "learns" which upstream server corresponds to which session identifier. Generally, these identifiers are passed in a HTTP cookie. If a request contains a session identifier already "learned", NGINX Plus forwards the request to the corresponding server:

  Copy

  ```
  upstream backend {
     server backend1.example.com;
     server backend2.example.com;
     sticky learn
         create=$upstream_cookie_examplecookie
         lookup=$cookie_examplecookie
         zone=client_sessions:1m
         timeout=1h;
  }
  ```

  In the example, one of the upstream servers creates a session by setting the cookie `EXAMPLECOOKIE` in the response.

  The mandatory `create` parameter specifies a variable that indicates how a new session is created. In the example, new sessions are created from the cookie `EXAMPLECOOKIE` sent by the upstream server.

  The mandatory `lookup` parameter specifies how to search for existing sessions. In our example, existing sessions are searched in the cookie `EXAMPLECOOKIE` sent by the client.

  The mandatory `zone` parameter specifies a shared memory zone where all information about sticky sessions is kept. In our example, the zone is named **client_sessions** and is `1` megabyte in size.

  This is a more sophisticated session persistence method than the previous two as it does not require keeping any cookies on the client side: all info is kept server-side in the shared memory zone.

  If there are several NGINX instances in a cluster that use the "sticky learn" method, it is possible to sync the contents of their shared memory zones on conditions that:

  - the zones have the same name
  - the `zone_sync` functionality is configured on each instance
  - the `sync` parameter is specified

  Copy

  ```
     sticky learn
         create=$upstream_cookie_examplecookie
         lookup=$cookie_examplecookie
         zone=client_sessions:1m
         timeout=1h
         sync;
  }
  ```

See Runtime State Sharing in a Cluster for details.

Limiting the Number of Connections

With NGINX Plus, it is possible to limit the number of active connections to an upstream server by specifying the maximum number with the `max_conns` parameter.

If the `max_conns` limit has been reached, the request is placed in a queue for further processing, provided that the `queue` directive is also included to set the maximum number of requests that can be simultaneously in the queue:

Copy

```
upstream backend {
    server backend1.example.com max_conns=3;
    server backend2.example.com;
    queue 100 timeout=70;
}
```

If the queue is filled up with requests or the upstream server cannot be selected during the timeout specified by the optional `timeout` parameter, the client receives an error.

Note that the `max_conns` limit is ignored if there are idle `keepalive` connections opened in other `worker processes`. As a result, the total number of connections to the server might exceed the `max_conns` value in a configuration where the memory is shared with multiple worker processes.

Configuring Health Checks

NGINX can continually test your HTTP upstream servers, avoid the servers that have failed, and gracefully add the recovered servers into the load-balanced group.

See HTTP Health Checks for instructions how to configure health checks for HTTP.

Sharing Data with Multiple Worker Processes

If an `upstream` block does not include the `zone` directive, each worker process keeps its own copy of the server group configuration and maintains its own set of related counters. The counters include the current number of connections to each server in the group and the number of failed attempts to pass a request to a server. As a result, the server group configuration cannot be modified dynamically.

When the `zone` directive is included in an `upstream` block, the configuration of the upstream group is kept in a memory area shared among all worker processes. This scenario is dynamically configurable, because the worker processes access the same copy of the group configuration and utilize the same related counters.

The `zone` directive is mandatory for active health checks and dynamic reconfiguration of the upstream group. However, other features of upstream groups can benefit from the use of this directive as well.

For example, if the configuration of a group is not shared, each worker process maintains its own counter for failed attempts to pass a request to a server (set by the max_fails parameter). In this case, each request gets to only one worker process. When the worker process that is selected to process a request fails to transmit the request to a server, other worker processes don't know anything about it. While some worker process can consider a server unavailable, others might still send requests to this server. For a server to be definitively considered unavailable, the number of failed attempts during the timeframe set by the `fail_timeout` parameter must equal `max_fails` multiplied by the number of worker processes. On the other hand, the `zone` directive guarantees the expected behavior.

Similarly, the Least Connections load-balancing method might not work as expected without the `zone` directive, at least under low load. This method passes a request to the server with the smallest number of active connections. If the configuration of the group is not shared, each worker process uses its own counter for the number of connections and might send a request to the same server that another worker process just sent a request to. However, you can increase the number of requests to reduce this effect. Under high load requests are distributed among worker processes evenly, and the `Least Connections` method works as expected.

Setting the Zone Size

It is not possible to recommend an ideal memory-zone size, because usage patterns vary widely. The required amount of memory is determined by which features (such as session persistence, health checks, or DNS re-resolving) are enabled and how the upstream servers are identified.

As an example, with the `sticky_route` session persistence method and a single health check enabled, a 256-KB zone can accommodate information about the indicated number of upstream servers:

- 128 servers (each defined as an IP-address:port pair)
- 88 servers (each defined as hostname:port pair where the hostname resolves to a single IP address)
- 12 servers (each defined as hostname:port pair where the hostname resolves to multiple IP addresses)

Configuring HTTP Load Balancing Using DNS

The configuration of a server group can be modified at runtime using DNS.

For servers in an upstream group that are identified with a domain name in the `server` directive, NGINX Plus can monitor changes to the list of IP addresses in the corresponding DNS record, and automatically apply the changes to load balancing for the upstream group, without requiring a restart. This can be done by including the `resolver` directive in the `http` block along with the `resolve` parameter to the `server` directive:

Copy

```
http {
    resolver 10.0.0.1 valid=300s ipv6=off;
    resolver_timeout 10s;
    server {
        location / {
            proxy_pass http://backend;
        }
    }
    upstream backend {
        zone backend 32k;
        least_conn;
        # ...
        server backend1.example.com resolve;
        server backend2.example.com resolve;
    }
}
```

In the example, the `resolve` parameter to the `server` directive tells NGINX Plus to periodically re-resolve the **backend1.example.com** and **backend2.example.com** domain names into IP addresses.

The `resolver` directive defines the IP address of the DNS server to which NGINX Plus sends requests (here, `10.0.0.1`). By default, NGINX Plus re-resolves DNS records at the frequency specified by time-to-live (TTL) in the record, but you can override the TTL value with the `valid` parameter; in the example it is `300` seconds, or `5` minutes.

The optional `ipv6=off` parameter means only IPv4 addresses are used for load balancing, though resolving of both IPv4 and IPv6 addresses is supported by default.

If a domain name resolves to several IP addresses, the addresses are saved to the upstream configuration and load balanced. In our example, the servers are load balanced according to the Least Connections load-balancing method. If the list of IP addresses for a server has changed, NGINX Plus immediately starts load balancing across the new set of addresses.

Load Balancing of Microsoft Exchange Servers

In NGINX Plus Release 7 and later, NGINX Plus can proxy Microsoft Exchange traffic to a server or a group of servers and load balance it.

To set up load balancing of Microsoft Exchange servers:

1. In a `location` block, configure proxying to the upstream group of Microsoft Exchange servers with the `proxy_pass` directive:

   Copy

   ```
   location / {
       proxy_pass https://exchange;
       # ...
   }
   ```

2. In order for Microsoft Exchange connections to pass to the upstream servers, in the `location` block set the `proxy_http_version` directive value to `1.1`, and the `proxy_set_header` directive to `Connection ""`, just like for a keepalive connection:

   Copy

   ```
   location / {
       # ...
       proxy_http_version 1.1;
       proxy_set_header   Connection "";
       # ...
   }
   ```

3. In the `http` block, configure a upstream group of Microsoft Exchange servers with an `upstream` block named the same as the upstream group specified with the `proxy_pass` directive in Step 1. Then specify the `ntlm` directive to allow the servers in the group to accept requests with NTLM authentication:

   Copy

```
http {
    # ...
    upstream exchange {
        zone exchange 64k;
        ntlm;
        # ...
    }
}
```

4. Add Microsoft Exchange servers to the upstream group and optionally specify a load-balancing method:

Copy

```
http {
    # ...
    upstream exchange {
        zone exchange 64k;
        ntlm;
        server exchange1.example.com;
        server exchange2.example.com;
        # ...
    }
}
```

Complete NTLM Example

Copy

```
http {
    # ...
    upstream exchange {
        zone exchange 64k;
        ntlm;
        server exchange1.example.com;
        server exchange2.example.com;
    }

    server {
        listen              443 ssl;
        ssl_certificate     /etc/nginx/ssl/company.com.crt;
        ssl_certificate_key /etc/nginx/ssl/company.com.key;
        ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;

        location / {
            proxy_pass         https://exchange;
            proxy_http_version 1.1;
            proxy_set_header   Connection "";
        }
    }
}
```

For more information about configuring Microsoft Exchange and NGINX Plus, see the Load Balancing Microsoft Exchange Servers with NGINX Plus deployment guide.

Dynamic Configuration Using the NGINX Plus API

With NGINX Plus, the configuration of an upstream server group can be modified dynamically using the NGINX Plus API. A configuration command can be used to view all servers or a particular server in a group, modify parameter for a particular server, and add or remove servers. For more information and instructions, see Configuring Dynamic Load Balancing with the NGINX Plus API.