Part 1: Nginx Load Balancing

Setting up Load Balancing

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

HTTP load balancer – Distributing HTTP requests across a group of servers based on a choice of algorithms, with passive and proactive checking of upstream server health and runtime modification of the load-balancing configuration.
It is possible to use nginx as a very efficient HTTP load balancer to distribute traffic to several application servers and to improve performance, scalability and reliability of web applications with nginx. We first configure the servers.

On the back end web servers, run the following commands to install nginx:

```
sudo apt-get install -y nginx
uname -n | sudo tee /usr/share/nginx/html/index.html
```

On the load balancer, run the following commands:

```
sudo apt-get install -y nginx
```

You need 2 modules which are built into the nginx core: Proxy, which forwards requests to another location, and Upstream, which defines the other location(s). They should be available by default.

Use the following as the contents of `/etc/nginx/sites-available/default`:

```
upstream web_backend {
    # Uncomment for the IP Hashing load balancing method:
    # ip_hash;
    
    # Uncomment for the Least Connected load balancing method:
    # least_conn;
    
    # Replace the IP addresses with the IP addresses (or host names) of your back end web servers.
    # Examples:
    # server www1.example.com:8080;
    # server 192.168.1.100;
    server backend1.example.com;
    server backend2.example.com;
    server backend3.example.com;
}
server {
    listen 80;
    location / {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_pass http://web_backend;
    }
}
```

Make nginx read the new configuration by running the following command:
sudo service nginx reload

In the example above, there are 3 instances of the same application running on srv.1-srv.3. When the load balancing method is not specifically configured, it defaults to round-robin. All requests are proxied to the server group web_backend.

If you require nginx to apply HTTP load balancing to distribute the requests, you can just put the above upstream web_backend into the http{} as (similar for https{})

```nginx
http{
    upstream web_backend {
        ...
    }
}
```

If one of the servers needs to be temporarily removed, 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:

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

By default, NGINX distributes requests among the servers in the group according to their weights using the round robin algorithm. The weight parameter of the server directive sets the weight of a server, by default, it is 1:

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

In the example, the first server has weight 5, the other two servers have the default weight (=1), but one of them is marked as a backup server and does not normally receive any requests. So of every six requests, five requests will be sent to the first server and one request will be sent to the second server.
Choosing a Load Balancing Method

Nginx supports four load balancing methods:

- The **round-robin** method: requests are distributed evenly across the servers with server weights taken into consideration. This method is used by default:

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

- The **least_conn** method: a request is sent to the server with the least number of active connections with server weights taken into consideration:

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

- The **ip_hash** method: the server to which a request is sent is determined from the client IP address. In this case, either the first three octets of 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.

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

- The **generic hash** method: the server to which a request is sent is determined from a user-defined key which may be a text, variable, or their combination. For example, the key may be a source IP and port, or URI:

  ```
  upstream backend {
  ```
The optional `consistent` parameter of the `hash` directive enables ketama consistent hash load balancing. Requests will be 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 few keys will be remapped which will minimize cache misses in case of load balancing cache servers and other applications that accumulate state.

### Online Resources

Please check the following URL for more detailed information

Part 2: Nginx Logging and Error Checking

This part describes how to configure logging of errors and processed requests, which mainly used for logging errors and requests (If some functions are only available in NGINX Plus, a paid version of the software, students are not expected to use them.).

Setting up the Error Log

Nginx writes information about encountered issues of different severity levels to the error log. The `error_log` directive sets up logging to a particular file, stderr, or syslog and specifies the minimal severity level of messages to log. By default, the error log is located at `logs/error.log`, and messages from all severity levels above the one specified are logged.

The configuration below changes the minimal severity level of error messages to log from error to warn:

```plaintext
error_log logs/error.log warn;
```

In this case, messages of warn, error, crit, alert, and emerg levels will be logged.

Setting up the Access Log

Nginx writes information about client requests in the `logs/access.log` after processing the request. By default, the access log is located at `logs/access.log`, and the information is written to the log in the predefined combined format. To override the default setting, use `log_format` directive to configure a format of logged messages, and `access_log` directive to specify the location of the log and its format.

The following example defines the `log_format` that extends the predefined combined format with the value indicating the ratio of gzip compression of the response. The format is then applied to a virtual server that enables compression.

```plaintext
http {
  log_format compression "$remote_addr - $remote_user [$time_local] '
  ""$request" $status $body_bytes_sent '"
  "$http_referer" "$http_user_agent" "$gzip_ratio"";

  server {
    gzip on;
    access_log /spool/logs/nginx-access.log compression;
    ...
  }
}
```
Logging can be optimized by enabling the buffer for log messages and the cache of descriptors of frequently used log files whose names contain variables. To enable buffering use the buffer parameter of the `access_log` directive to specify the size of the buffer. The buffered messages are then written to the log file when the next log message does not fit into the buffer as well as in some other cases.

To enable caching of log file descriptors, use the `open_log_file_cache` directive.

## Logging to Syslog

Syslog is a standard for computer message logging and allows collecting log messages from different devices on a single syslog server. In nginx, logging to syslog is configured with the `syslog:` prefix in `error_log` and `access_log` directives.

Syslog messages can be sent to a `server=` which can be a domain name, an IP address, or a UNIX-domain socket path. A domain name or IP address can be specified with a port, by default port 514 is used. A UNIX-domain socket path can be specified after the `unix:` prefix:

```bash
error_log server=unix:/var/log/nginx.sock debug;
access_log syslog:server=[2001:db8::1]:1234,facility=local7,tag=nginx,severity=info;
```

In the example, nginx error log messages will be written to UNIX domain socket with the debug logging level, and the access log will be written to a syslog server with IPv6 address and port 1234.

## Online Resources

Please check the following URL for more detailed information


---

### Part 3: Writing Python Scripts to Send HTTP Requests to an URL

In this part, we will use `urllib2`, which is a Python module for fetching URLs. It offers a very simple interface, in the form of the `urlopen` function. This is capable of fetching URLs using a variety of different protocols. It also offers a slightly more complex
interface for handling common situations - like basic authentication, cookies, proxies and so on. These are provided by objects called handlers and openers.

**Fetching URLs**

The simplest way to use `urllib2` is as follows:

```python
import urllib2
response = urllib2.urlopen('http://python.org/)
html = response.read()
```

Many uses of `urllib2` will be that simple (note that instead of an ‘http:’ URL we could have used an URL starting with ‘ftp:’, ‘file:’, etc.).

HTTP is based on requests and responses - the client makes requests and servers send responses. `urllib2` mirrors this with a Request object which represents the HTTP request you are making. In its simplest form you create a Request object that specifies the URL you want to fetch. Calling `urlopen` with this Request object returns a response object for the URL requested. This response is a file-like object, which means you can for example call `.read()` on the response:

```python
import urllib2

req = urllib2.Request('http://www.python.org/
response = urllib2.urlopen(req)
the_page = response.read()
```

Note that `urllib2` makes use of the same Request interface to handle all URL schemes. For example, you can make an FTP request like so:

```python
req = urllib2.Request('ftp://example.com/
response = urllib2.urlopen(req)
ftp_page = response.read()
```

In the case of HTTP, there are two extra things that Request objects allow you to do: First, you can pass data to be sent to the server. Second, you can pass extra information (“metadata”) about the data or the about request itself, to the server - this information is sent as HTTP “headers”.

**Data**

Sometimes you want to send data to a URL. With HTTP, this is often done using what’s known as a POST request. This is often what your browser does when you submit a HTML form that you filled in on the web. Not all POSTs have to come from forms: you can use a POST to transmit arbitrary data to your own application. In the common case
of HTML forms, the data needs to be encoded in a standard way, and then passed to the Request object as the data argument. The encoding is done using a function from the urllib library not from urllib2.

```python
import urllib
import urllib2

url = 'http://www.pythontab.com'
values = {'name' : 'Michael Foord',
          'location' : 'pythontab',
          'language' : 'Python' }

data = urllib.urlencode(values)
req = urllib2.Request(url, data)
response = urllib2.urlopen(req)
the_page = response.read()
```

## Headers

We’ll discuss here one particular HTTP header, to illustrate how to add headers to your HTTP request.

Some websites dislike being browsed by programs, or send different versions to different browsers. By default urllib2 identifies itself as “Python-urllib/x.y” (where x and y are the major and minor version numbers of the Python release, e.g. Python-urllib/2.5), which may confuse the site, or just plain not work. The way a browser identifies itself is through the “User-Agent” header. When you create a Request object you can pass a dictionary of headers in. The following example makes the same request as above, but identifies itself as a version of Internet Explorer.

```python
import urllib
import urllib2

url = 'http://www.pythontab.com/'
user_agent = 'Mozilla/5.0 (compatible; Windows NT 6.1; Win64; x64)'
values = {'name' : 'Michael Foord',
          'location' : 'pythontab',
          'language' : 'Python' }

headers = { 'User-Agent' : user_agent }

data = urllib.urlencode(values)
req = urllib2.Request(url, data, headers)
response = urllib2.urlopen(req)
the_page = response.read()
```
Online Resources

Please check the following URL for more detailed information

https://docs.python.org/2/howto/urllib2.html