Homemade load balancing with nginx + Lua

akononov@iponweb.net
ashcherbinin@iponweb.net
Technical requirements

1. Correct answer for every user request
2. SSL
3. Redundancy
4. Sticky balancing
5. Universal company wide solution
6. Easy support and changing balancing method
7. Backend monitoring
Software which we found

1. Nginx vs HA proxy
2. Nginx vs Lighttpd
3. Nginx not enough
4. Nginx + lua - good enough!
1. Distributed load testing (jmeter)
2. Add / remove node testing
3. Testing on prod env.
4. Health checks and resurrection of dead nodes
5. Fine tune linux and nginx in clouds (aws gce)
monitoring

1. log monitoring
2. Nginx status page monitoring.
3. Upstream status monitoring via nginx
4. 3rd party monitoring
Some metrics

1. network performance 180K pps
2. network performance 500 Mbit/s
3. http requests overall 20K rps

Next - Anton`s part
The problem

Distribute HTTP requests from users among N servers so that:

- when an additional server is added, vast majority of users do not change their servers, but \( \frac{1}{N} \)th of users move from each of N “old” servers to the newly added server;

- user “key”, which determines if 2 requests come from the same user or 2 different users, may be arbitrarily complex (IP address, cookies, URL parameters, etc., or any combination)

(Sound like sticky balancing? It is NOT)
Mapping users to $N$ servers?
Piece of cake! Easier than a presidential campaign!

\[ \text{ServerNumber} = \text{hash(user)} \mod N \]
user 0: $0 \mod 3 = 0$

users 0 and 1 did not change servers they are bound to.
$2/6 = 1/3$ users

user 1: $1 \mod 3 = 1$

user 2: $2 \mod 3 = 2$

user 3: $3 \mod 3 = 0$

user 4: $4 \mod 3 = 1$

user 5: $5 \mod 3 = 2$

users 2, 3, 4, 5 changed servers they are bound to.
$4/6 = 2/3$ users
Naïve approach

Going
from $n-1$ to $n$ servers (1 more server)
or from $n$ to $n-1$ servers (1 less server)
results in
$\frac{1}{n}$ users does not change server, while
$\frac{(n-1)}{n}$ users change server
AND THEN, I TOLD THEM THERE WOULD BE CHANGE
An ideal solution is impossible?
Read Wikipedia, you worthless office plankton!

http://en.wikipedia.org/wiki/Consistent_hashing
For every server, pseudorandomly mark $p=4$ points on the circle.

Look for nearest greater point (anticlockwise).

$\psi = \text{hash(user)}$
A new server added (blue)? Add $P=4$ points for it.

Look for nearest greater point (anticlockwise)

$\varphi = \text{hash}\,(\text{user})$
Dealing with real world: Problems
- Cannot create consistent hash in nginx init;
- nginx dumped core when using Lua socket. http.request();
- deadlock when re-hashing using a single nginx worker process;
- timeouts when stress-testing in Amazon AWS.
The solution

1. ~300 lines in Lua (lots of debug messages)
2. Uses nginx + Luajit + Lua nginx module + Lua upstream nginx module
3. The config written by sysadmins is a Lua program (different in different projects)
4. Seems efficient enough to saturate 1 Gbps network interface with CPU usage < 50% on a “8-core” virtual Amazon AWS server
GetUserKey = function()

    local v = ngx.var
    local uses_msie = v.http_user_agent and v.http_user_agent:match('MSIE')
    local cookie = v.cookie_foobar
    local seller_id = v.arg_seller_id
    local buyer_id = v.arg_bidder_id

    if uses_msie then -- send all the users using MSIE to the same server
        return 'ANY_CONSTANT_STRING'
    elseif cookie and string.len(cookie) == 36 then -- valid and not opt-out
        return cookie
    elseif seller_id and seller_id ~= '' then -- seller_id not empty
        return seller_id
    elseif buyer_id and buyer_id ~= '' then -- buyer_id not empty
        return buyer_id
    end

    return v.remote_addr -- last resort is user's IP address
end
Links

1. http://luajit.org
Thank you!