LineRate: HTTP session ID persistence in scripting using memcache

Brian Talley, 2015-05-02

> Using the LineRate Node.js datapath scripting engine to achieve session-based client/server affinity

LineRate introduced the `selectServer()` method and the `newServerSelected` event extensions to the built-in Node.js http module in version 2.4. This opens up a lot of possibilities for dynamically selecting real-servers based on HTTP session information, such as HTTP headers (cookies, user agents) or any other metric, such as server load, time of day, geolocation, or pretty much any other metric you can think of. One use-case is particularly useful: obtaining server affinity based on a session ID such as `PHPSESSION` (PHP), `JSESSIONID` (tomcat), `ASP.NET_SessionId` (ASP), and `connect.sid` (Express/Connect). Read on to see how.

A quick aside: LineRate added cookie-based real-sever affinity way back in version 1.6. This is a powerful feature in it's own right and might get you want you want with a minimal amount of effort - I urge you to check it out. But, if you're averse to injecting a new cookie or you just need more fine-grained control of the affinity algorithm, read on.

I'll show you a script and walk through some of the highlights to accomplish session ID persistence via scripting below. For bonus points, I'm going to use memcache to store a key-value pair, which will consist of the session ID (key) and the real-server name (value). (See here for why you don't want to just store this data in a javascript data structure.) A Redis server is already pre-installed and ready to go on your LineRate instance, and it works great, but I already wrote an example of using Redis in another article, so I'll demo something different here. (This is in contrast to memcache where you'll need a separate memcache server/cluster running somewhere on your network.)

The selectServer() method extends the http.clientRequest class and allows you to specify the real-server to receive the request. The newServerSelected event fires whenever the system forwards the request to a real-server that you did not specify. This is particularly useful (and critical for this example) when you don't care which server handles the session, just that the *same* server *continues* to handle that session.

Keep reading for a discussion on the various pieces of the script; see the very bottom for the full script.

If you're not already familiar with Node.js and the LineRate scripting engine, be sure to check out the LineRate Scripting Developer's Guide.

requires and config

Load the required modules and update the config object as needed.

```
var vsm = require("lrs/virtualServerModule");
var cookie = require("cookie");
var async = require("async");
var memcache = require('memcache');

// Change config as needed.
var config = {
    session_id_key: "connect.sid", // session-id key being used
    vs: "vs_http",                 // name of virtual-server
    memcache_host: '172.16.87.154'
};
```

memcache

memcache is a "high-performance, distributed memory object caching system". It's a really good option for an in-memory (read: fast), distributed, key-value store. Since storing keys and values is exactly what we need to do for this script, we'll take it for a test drive.

Here's what the memcache code is doing: We load the module, create a new client object, define some event listeners and then connect to the memcache server. We'll also use the `get()` and `set()` methods later. This memcache code could be re-used in any scenario where a caching server is needed. If the client emits a 'close' or 'error' event, we wait one second before trying to reconnect using `setTimeout()`.

```
var memcache = require('memcache');

var memcache_client = new memcache.Client(11211, config.memcache_host);

memcache_client.on('connect', function () {
    console.log('Connected to memcache server at ' +
                config.memcache_host + ':11211');

});

memcache_client.on('timeout', function () {
    console.log('Memcache connection timed out; reconnecting...');
    memcache_client.connect();

});

memcache_client.on('close', function () {
    console.log('Memcache connection closed; reconnecting (waiting 1s)...');
    // wait 1s before re-connecting
    setTimeout(function () {
        memcache_client.connect();
    }, 1000);
});

memcache_client.on('error', function (e) {
    console.log('Memcache connection error; reconnecting...');
    console.log(e);
    // wait 1s before re-connecting
    setTimeout(function () {
        memcache_client.connect();
    }, 1000);
});

memcache_client.connect();
```

async waterfall

When we receive a new request from a client, a few things need to happen serially. And each of those sub-processes relies on data from the previous process. This is a classic use-case for the 'waterfall' control flow from the async module. The waterfall flow will run a series of functions and pass the results of each function to the next function. In this case, we're doing three primary things: getting the session ID from the request, selecting the real-server based on the session ID and then making the request. Each of these three functions are detailed next.

```
function onRequest(servReq, servResp, cliReq) {
    async.waterfall([
        function(callback) {
            getSessionIdCookie(servReq, callback);
        },
        function(sessionId, callback) {
```

```
        function(sessionId, callback) {
            selectServer(sessionId, cliReq, callback);
        },
        function (cachedServerName, callback) {
            doRequest(servReq, servResp, cliReq,
                        cachedServerName, callback);
        }
    ], function (err, result) {
        if (err) {
            throw new Error(err);
        }
    });
}
```

getSessionIdCookie()

Just like the name of the function sounds, here we're getting the session ID from the cookie header in the request. The `sessionID` variable is initialized to `undefined`. If the request doesn't contain a session ID, the `sessionID` variable will remain `undefined`, otherwise `sessionID` gets set to, you guessed it, the session ID.

```
function getSessionIdCookie(servReq, callback) {
    var sessionId;
    // check for existence of session-id cookie
    if (servReq.headers.cookie) {
        var cookies = cookie.parse(servReq.headers.cookie);
        sessionId = cookies[config.session_id_key];
    }
    return callback(null, sessionId);
}
```

selectServer()

Here's where things start to get a little more interesting; this is where we dynamically select a real-server to which to send the request. If the original request did not contain a session ID in the cookie header, the request will just be processed 'normally' - the LineRate system will pick a real-server based on the configured load balancing algorithm for the virtual server.

If the request does contain a session ID cookie, we look up the session ID in memcache. If memcache already has an entry with this session ID, the request is part of a previous session and we send the request to the same server to which previous requests were sent using selectServer(). The unassuming `cliReq.selectServer(serverName);` line is the key to this whole script and is what gives us real-server affinity using the session ID. If the selected server is different than what the system would have chosen using the configured load balancing algorithm, this will cause the 'newServerSelected' event to fire (see next section).

```
function selectServer(sessionId, cliReq, callback) {
    if (!sessionId) { // no session-id cookie, proceed to next step
        return callback(null, null);
    }

    // lookup session id in memcache
    memcache_client.get(sessionId, function (err, result) {
        if (err) {
            console.log(err);
            return callback(err);
        }
        else {
            var serverName = result;
            if (serverName) {
```

```
                cliReq.selectServer(serverName);
            }
            return callback(null, serverName);
        }
    });
}
```

doRequest()

There's some interesting stuff happening in this function. We're piping the original request to the real-server we found in the `selectServer()` function. We're also listening for the real-server's response (cliResp). Why? This is where we snoop for a 'set-cookie' header signalling to our script that this is the first response in a new session. This set-cookie contains the session ID for the session. We record this session ID and real-server name pair in memcache.

The `newServerSelected` event is guaranteed to be emitted before the `response` event for `cliReq`. This ensures that we have all the data we need when the memcache `set()` is called in the `response` event handler. Also note that the "selected server" will be `null` for the first request of any new session. This is expected and allowed. The request will default to choosing a real-server based on the configured load balancing algorithm.

Lastly, the astute reader will note that we're using the expiration time of the session cookie to configure the expiration time of the memcache entry. This will ensure that the session info is in memcache for the duration it's needed and then cleans up after itself once it's expired.

```
function doRequest(servReq, servResp, cliReq,
                        cachedServerName, callback) {
    var selectedServerName;

    // Register "newServerSelected" handler to save the real-server name
    // which was used to send out the request.
    cliReq.on("newServerSelected", function(newServerName) {
        selectedServerName = newServerName;
    });

    // Register response handler to optionally update cache if a new server
    // selection was made.
    cliReq.on("response", function(cliResp) {
        if(!selectedServerName) {
            selectedServerName = cachedServerName;
        }

        // retrieve session-id cookie from "set-cookie" header in response
        var set_cookie_header = cliResp.headers["set-cookie"];
        if (set_cookie_header) {
            // Note: the "set-cookie" header is always an array containing the
            // set-cookie string as the first element.
            var set_cookie_object = cookie.parse(set_cookie_header[0]);
            var sessionId = set_cookie_object[config.session_id_key];
            // calculate memcache exptime from session expiration time
            var expiration = (new Date(set_cookie_object['Expires']).getTime())/1000;
            if (sessionId) {
                // set the memcache entry expiration to the expiration time
                // of the set-cookie
                memcache_client.set(sessionId,
                      selectedServerName,
                      function(err, result) {
                          // error handling here; depends on your env
                }, expiration);
            }
```

```
        }

        // Fastpipe response
        cliResp.bindHeaders(servResp);
        cliResp.fastPipe(servResp);
    });

    // Fastpipe request
    servReq.bindHeaders(cliReq);
    servReq.fastPipe(cliReq);

    return callback(null);
}
```

Testing

I have a pair of test real-servers running Express with the session module, called 'srv1' and 'srv2'. Each Express server responds with a JSON object with some information about the session.

Here's a simple series of commands demonstrating what happens when the script is running. Note the 'real-server' is the same for every request and the session ID is the same in the 2nd and 3rd request and in the cookie.jar file. (I truncated the session ID for readability.)

```
> curl -c cookie.jar http://172.16.87.157
{"real-server":"srv1","name":"connect.sid","session_id":"HQ1xyGhmrAUjZeJCG15JO019Pkb8K4NJ","expires":
> curl -b cookie.jar http://172.16.87.157
{"real-server":"srv1","name":"connect.sid","session_id":"HQ1xyGhmrAUjZeJCG15JO019Pkb8K4NJ","expires":
> curl -b cookie.jar http://172.16.87.157
{"real-server":"srv1","name":"connect.sid","session_id":"HQ1xyGhmrAUjZeJCG15JO019Pkb8K4NJ","expires":
```

Here's a contrasting series of commands demonstrating what happens when the script is **not** running. Note the round-robin'ing between real-servers 'srv1' and 'srv2'.

```
> curl -c cookie.jar http://172.16.87.157
{"real-server":"srv2","name":"connect.sid","session_id":"OlxEv3KPZHqZ0J0Nby0FVaEO_S8x061D","expires":
> curl -b cookie.jar http://172.16.87.157
{"real-server":"srv1","name":"connect.sid","session_id":"IJ2x2jxnEt5NP6y3pO_7AZalX2sHBfOu","expires":
> curl -b cookie.jar http://172.16.87.157
{"real-server":"srv2","name":"connect.sid","session_id":"OlxEv3KPZHqZ0J0Nby0FVaEO_S8x061D","expires":
> curl -b cookie.jar http://172.16.87.157
{"real-server":"srv1","name":"connect.sid","session_id":"SuIOpe7qSm9hj0bcimAAIqOturyir0-l","expires":
> curl -b cookie.jar http://172.16.87.157
{"real-server":"srv2","name":"connect.sid","session_id":"OlxEv3KPZHqZ0J0Nby0FVaEO_S8x061D","expires":
```

And, pulling it all together, here's the full script. Happy cloning!

Please leave a comment or reach out to us with any questions or suggestions and if you're not a LineRate user yet, remember you can try it out for free.

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc.
Corporate Headquarters
info@f5.com

F5 Networks
Asia-Pacific
apacinfo@f5.com

F5 Networks Ltd.
Europe/Middle-East/Africa
emeainfo@f5.com

F5 Networks
Japan K.K.
f5j-info@f5.com