Revisiting Hash Load Balancing and Persistence on BIG-IP LTM

Jason Rahm, 2011-21-01

A good while back (Dec '07!), Deb wrote up hash load balancing in six pages of goodness. She made some really good points in her introduction that bear repeating:

> "Load balancing of cache and proxy servers has long been a standard task for F5's BIG-IP LTM. However, cache and proxies have unique requirements over other types of applications. It's not realistically possible to cache an entire site, let alone the Internet, on a single cache device. The solution is to intelligently load balance requests and 'persist' them to a specific cache not based on who the client is, but based on the content requested. In a forward proxy example, all requests for site1.com would go to cache1 while all requests for site2.com go to cache2. Since each cache gets all the requests for their respective sites, they will have the most up to date version of the content cached. This also allows the caches to scale more efficiently as each would only need to cache N percent of the possible destination content. In BIG-IP LTM nomenclature, adding more members (nodes or servers) to the pool reduces the percent of content such as URIs that any cache server needs to store.

The point of this article is not to rewrite Deb's excellent work, so please go read that article first. Back? Great…let's move on. In my previous article, Fun with Hash Performance, I tested each of the hashing algorithms (minus the persist carp function), pushing fifty-thousand executions of the hash and timing it for comparison. This time, I'd like to take a look at how each of the hashes distributes the traffic amongst different pool sizes and with two different numbers of unique URLs. In the event you are priming cache arrays, or merely want to use all the resources you allocate to an application, using a hash gives you a deterministic way to plan. However, not all hashes are created equal in this regard, and I know a little (very little) on how they work in general, but have never seen any testing to prove out exactly how they behave with load balancing. Knowing how they react to your application and how they perform arms you, the technician, with another tool in the toolbox when it comes time to architect a solution. The goal here is to chart the standard deviation for each of the hashes. The lower the standard deviation, the more evenly distributed the connections/content would be. We'll get to the charts later on, however. Let's start with the test cases, which can be seen in Table 1.

Table 1 - Hashing Algorithm Tests

Test Case #	Pool Members	Unique URLs
1	3	2000
2	3	5000
3	4	2000
4	4	5000
5	8	2000
6	8	5000
7	16	2000
8	16	5000
9	32	2000
10	32	5000

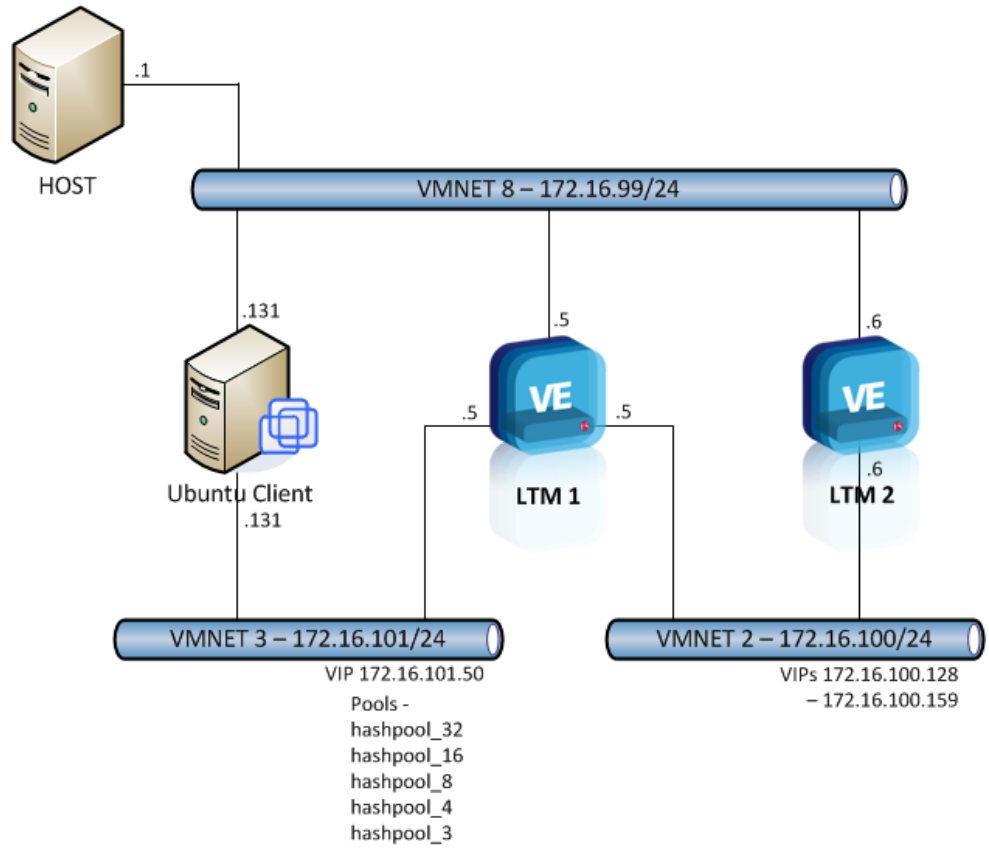
Generating the Unique URLs

I don't have five-thousand, or even two-thousand unique URLs laying around, so I'll need to generate them. I need each of them in two formats as well. When I began testing, my intention was to run every request from a class through an iRule loop, and for all but one of the hashes that's still true. For the built-in carp persistence, however, I couldn't figure a way to grab the data without a pool member connection, so I added another LTM to serve as pool members and used curl to request the URLs. It was important for consistency, so I used this python code to generate the random string and save the list as a file of complete URLs for curl and a list of URIs for the class:

```
#!/usr/bin/python
import sys, os, random, string

num = int(sys.argv[1])
classfile = "/home/jrahm/randomURLs_%s" % num
urlfile = "/home/jrahm/urllist_%s.txt" % num

c = open(classfile, 'w')
u = open(urlfile, 'w')

for x in range(1, num+1):
    y="".join(random.choice(string.letters) for i in xrange(random.randrange(3,15,1))) + "/image%s.jpg"
% x
    c.write(y+ "\n")
    u.write("http://172.16.101.50/" + y + "\n")
```

Configuring the Test Environment

For most of the hashes, I just needed an LTM and a browser. For the carp testing, I needed another LTM and a client to loop through the unique URLs. With LTM VE and a laptop with 8G of RAM, I was able to set up the entire environment virtually, as shown below.



LTM1 is doing the heavy lifting. A single virtual server (172.16.101.50) is configured with one of several pools, each with a specific number of pool members (shown in the pool name in the image above). These pools were configured in tmsh with the aptly named create_pool.tcl script from the wiki. The pool members are actually virtual servers hosted on LTM 2. I created those virtuals with a new tmsh script based on the create_pool.tcl script called, shockingly, create_virtuals.tcl, which I added to the tmsh wiki under CreateLTMVirtuals. Yea, tmsh!

The iRule I used for these virtual servers is very short, and used just to respond to queries passed on by LTM 1:

```
1: when HTTP_REQUEST {
2: HTTP::respond 200 content "<html><body>[LB::server
addr]</body></html>"
3: }
```

The Test iRule

I'm utilizing tables to store the hash results/pool hit counts, once the urls are processed, I lookup the data in the table, store it in a variable, then delete the table. The carp urls are coming from a client, the rest of the hashing algorithms are processed internally to the iRule (shown below). As I worked through the test cases, only things I changed were the class name (random_urls_2k or random_urls_5k) and the pool name. For the carp test preceding the /hashtest step, I also changed the default pool on the virtual to match the iRule. I could have automated all this to death, but chose expediency on some things.

```
1: when HTTP_REQUEST {
2: set uri [HTTP::uri]
3: persist carp $uri
4: if
{ $uri eq "/hashtest" } {
5: foreach hash [list "md5" "crc32" "sha1" "sha256" "sha384" "sha512"] {
6: set clen [class size random_urls_2k]
7: for
{ set x 0 } {$x &lt; $clen} { incr x } {
8: binary
scan [$hash [class element -name $x random_urls_2k]] w1 hashval
9: set hashval [expr {$hashval % [active_members
hashpool_3]}]
10: if { [table incr -subtable $hash -mustexist pm$hashval] eq ""} {
11: table set -subtable $hash pm$hashval 1 indefinite indefinite
12: }
13: }
14: foreach pm [table keys -subtable
$hash] {
15: append hash_$hash "[table lookup -subtable $hash $pm], "
16: }
17: table delete -subtable $hash -all
18: }
19: foreach pm [table keys -subtable carp] {
20: append hash_carp
"[table lookup -subtable carp $pm], "
21: }
22: table delete -subtable carp -all
23: HTTP::respond 200 content "<html><body>crc32:$hash_crc32<br>md5:$hash_md5<br>\
24: sha1:$hash_sha1<br>sha256:$hash_sha256<br>sha384:$hash_sha384<br>\
25: sha512:$hash_sha512<br>carp:$hash_carp</body></html>"
26: }
27: }
28: when HTTP_REQUEST_SEND {
29: if { $uri ne "/hashtest" } {
30: set psel [getfield [LB::server addr] "." 4]]
31: if { [table incr -subtable carp -mustexist pm$psel] eq ""} {
32: table set
-subtable carp pm$psel 1 indefinite indefinite
33: }
34: }
35:
}
```

Because I couldn't test carp in the same was as the other hashes, each test case is a two step process. First, I hit the LTM 1 virtual server from my Ubuntu client with this command, referencing the urllist built earlier:

```
xargs curl –l < /home/jrahm/urllist_2000.txt
```

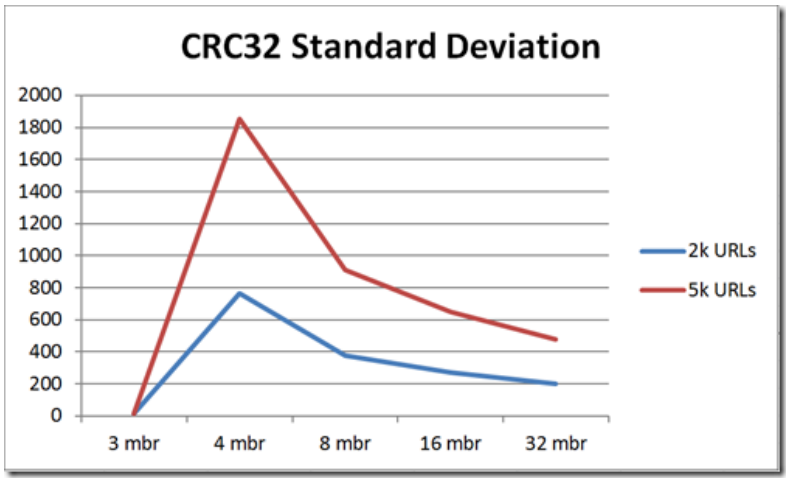
Once that completes, I hit the virtual directly with this command, which returns the data shown immediately below the command:

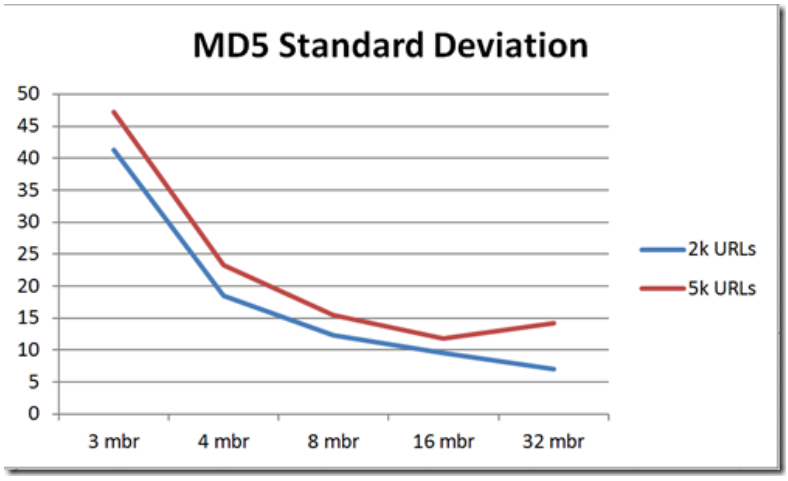
```
curl http://172.16.101.50/hashtest
<html><body>crc32:655, 682, 663, <br>md5:634, 653, 713, <br>
        sha1:684, 655, 661, <br>sha256:669, 714, 617, <br>
        sha384:689, 635, 676, <br>sha512:634, 684, 682, <br>
        carp:671, 642, 646, </body></html>
```

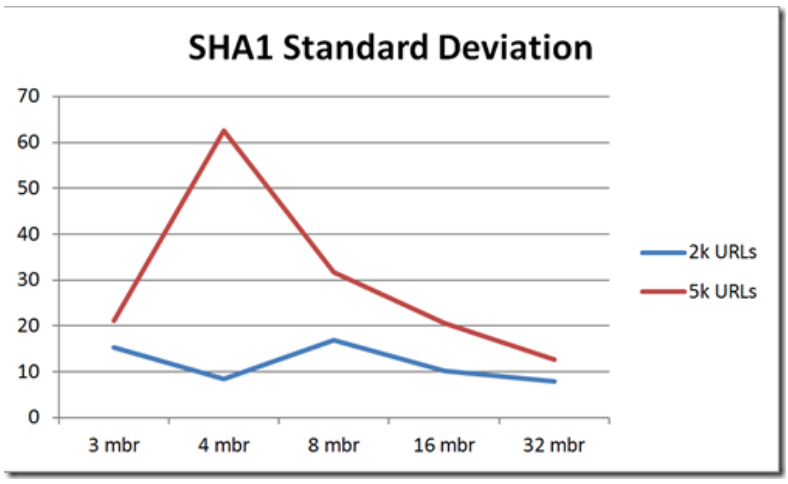
That gives me the data I need to move on and calculate and graph the standard deviation for test case 1. After going through each test case, I have results for each of the hashes for each of the urllist lengths, which I set arbitrarily at 2000 and 5000.

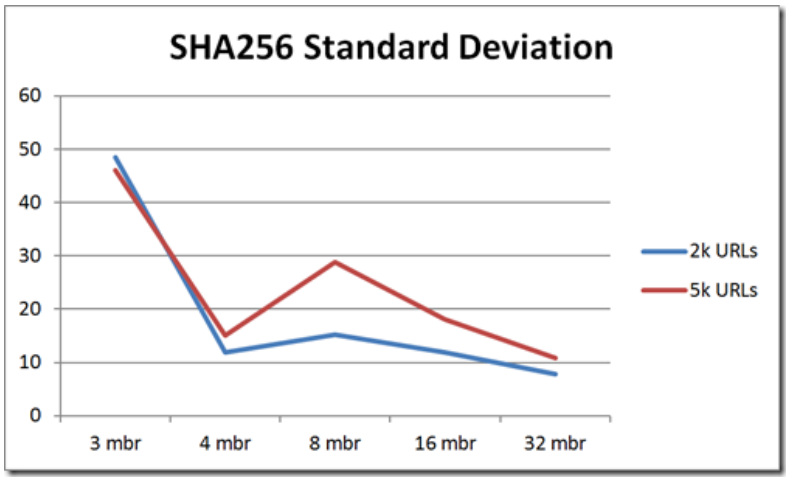
The Graphs

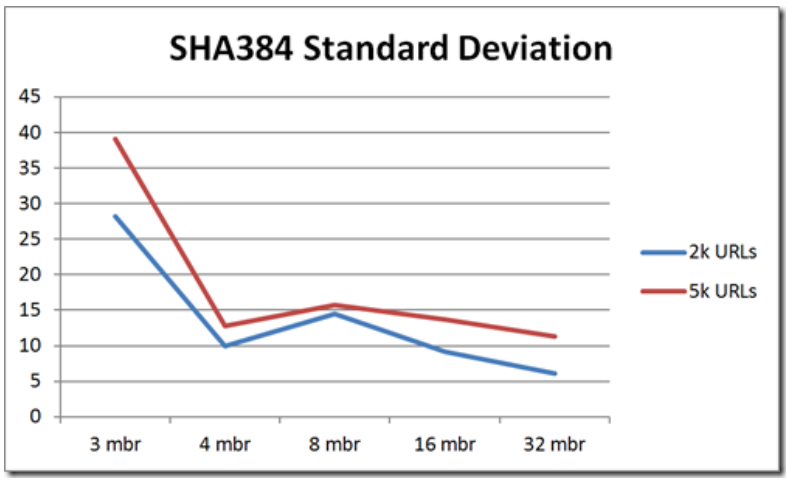
The data (and more graphs) are available if you want to manipulate and graph a different way. I broke down the data and each of the graphs below represent a single hash and it's standard deviation given the number of unique URLs and the number of members in the pool. Note that for CRC32, the max pool members with connections was always ten, so for a 16 member pool, six members had no connection at all. I wasn't entirely sure how to handle this in the calculations, so I just entered zeroes for the remaining pool members. But that said, you surely will not want to use CRC32 as a load balancing/persistence algorithm in pools larger than three. In case it's not apparent, the x-axis is number of pool members (shown as mbr), and the y-axis is the value of the standard deviation. As alluded to earlier, a lower standard deviation is more desirable if you want an even distribution of your content (in a caching scenario, or connections elsewise).

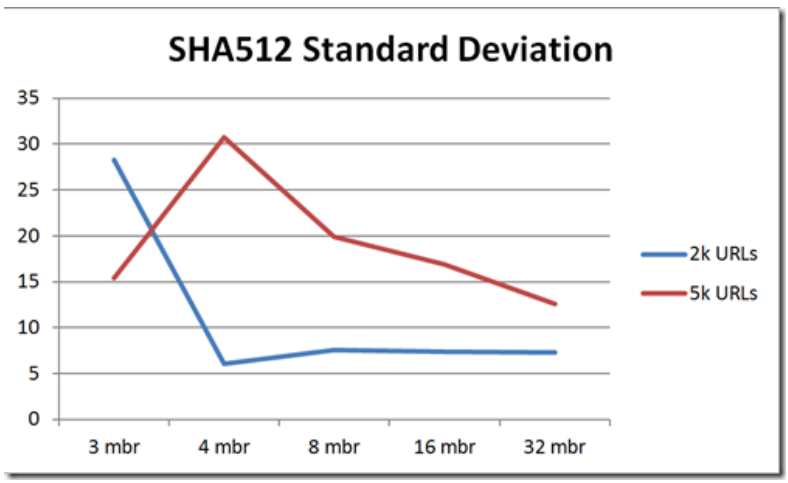


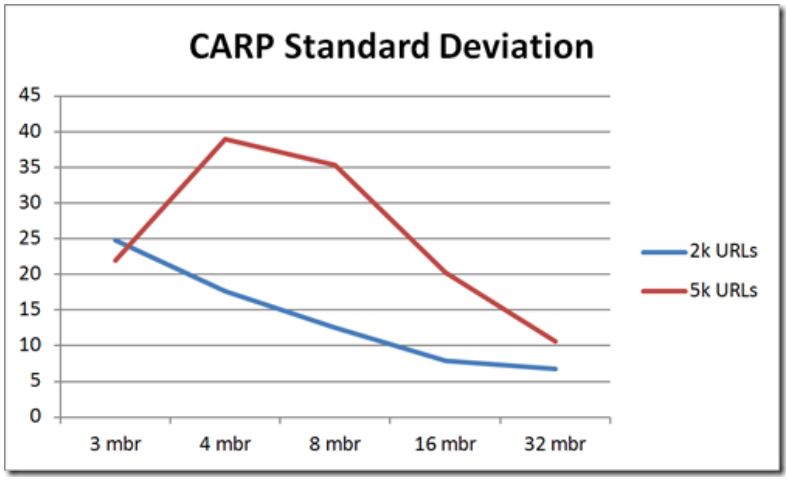



SHA1 Standard Deviation
70
60
50
40
30
20
10
0
3 mbr
4 mbr
8 mbr
16 mbr
32 mbr
2k URLs
5k URLs


SHA256 Standard Deviation
60
50
40
30
20
10
0
3 mbr
4 mbr
8 mbr
16 mbr
32 mbr
2k URLs
5k URLs


SHA384 Standard Deviation
45
40
35
30
25
20
15
10
5
0
3 mbr
4 mbr
8 mbr
16 mbr
32 mbr
2k URLs
5k URLs


SHA512 Standard Deviation
35
30
25
20
15
10
5
0
3 mbr
4 mbr
8 mbr
16 mbr
32 mbr
2k URLs
5k URLs



It's interesting data, isn't it? All the cryptographic hashes largely perform better distributions with more targets. Keep all this in context, though. Remember that SHA384/SHA512 was six times longer in processing than the CRC32, and double MD5/SHA1. Finding the sweet spot between algorithm/distribution will come down to available resources, business requirements, or both.

Conclusion

Advanced load balancing and persistence with hashing algorithms offers a great deal of benefit, but great care should be exercised in testing and knowing your environment.

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