Investigating Efficiencies in iRules: Handling Wildcards in Hostnames

Jason Rahm, 2010-25-05

This isn't the first time I've looked at optimization techniques with iRules. I've blogged about why testing matters, and hosted a small contest as a result of that testing (hmm, I think I still owe natty76 a t-shirt…) to give other iRulers out there an opportunity to find efficiencies. Well, every now and then a post catches my eye and I start to think about the fundamental differences in approach to a problem. A thread in the Adv Design/Config General Discussion forum, Bypassing a VIP based on destination address, was searching for a way to match a host that had a wildcard in the middle: webmail.<something>.server.com. Hamish, one of our MVPs, suggested using the matches_glob operator. (Note: the html tags in the HTTP::respond content section were removed for clarity. The editor wasn't displaying them properly.)

iRule Solutions

matches_glob

```
when HTTP_REQUEST {
  if { [HTTP::host] matches_glob "webmail.*.server.com" } {
    HTTP::respond 200 content {
  OK
    }
  }
}
```

I came up with a couple more alternatives, one utilizing nested if conditionals and the other using the string match command.

nested conditional & string match

```
when HTTP_REQUEST {
  if { [llength [split [HTTP::host] "."]] == 4 } {
    if { [HTTP::host] starts_with "webmail" } {
      if { [HTTP::host] ends_with "server.com" } {
  HTTP::respond 200 content {
   OK
 }
      }
    }
  }
}



**********

when HTTP_REQUEST {
  if { [string match -nocase webmail.*.server.com [HTTP::host]] } {
    HTTP::respond 200 content {
  OK
    }
  }
}
```

After reaching out to unRuleY with some testing results (more on that later), he proposed a couple more options, both the same approach, but one using the domain command and the other using getfield.

domain & getfield

```
when HTTP_REQUEST {
    if { ([domain [HTTP::host] 4] eq [HTTP::host]) && \
      ([HTTP::host] starts_with "webmail.") && \
   ([HTTP::host] ends_with ".server.com") } {
  HTTP::respond 200 content {
    OK
  }
    }
}

**********

when HTTP_REQUEST {
    if { ([getfield [HTTP::host] "." 5] eq "") && \
      ([HTTP::host] starts_with "webmail.") && \
   ([HTTP::host] ends_with ".server.com") } {
  HTTP::respond 200 content {
    OK
  }
    }
}
```

If you're counting, that's five iRule solutions to one problem. Three unique solutions, really. If I were to classify these, I'd make the nested conditional one solution, matches_glob/string match another solution, and domain/getfield the final solution. So how do all these iRules stack up against one another?

Testing Methodology

Testing was performed on a BIG-IP 3600 platform running the 10.2 release. The device under test had no other traffic flowing during testing (except a control ssh session and iControl sessions from the editor) and only the rule under test applied to the virtual server. Three runs of 10,000 requests from apachebench were averaged together for each iRule solution. The iRules themselves were devoid of any logic not necessary for the solution other than the HTTP::respond to confirm the rule was working.

Results

If you're the pretty picture type, the results are shown in Figure 1 to the right. So it's safe to say that my idea of the nested conditional was not the best. It works, but it is not efficient. The matches_glob and string match solution results are the most efficient, and performance is really close, so close that I reached out to get some insight on this one. Straight from unRuleY:

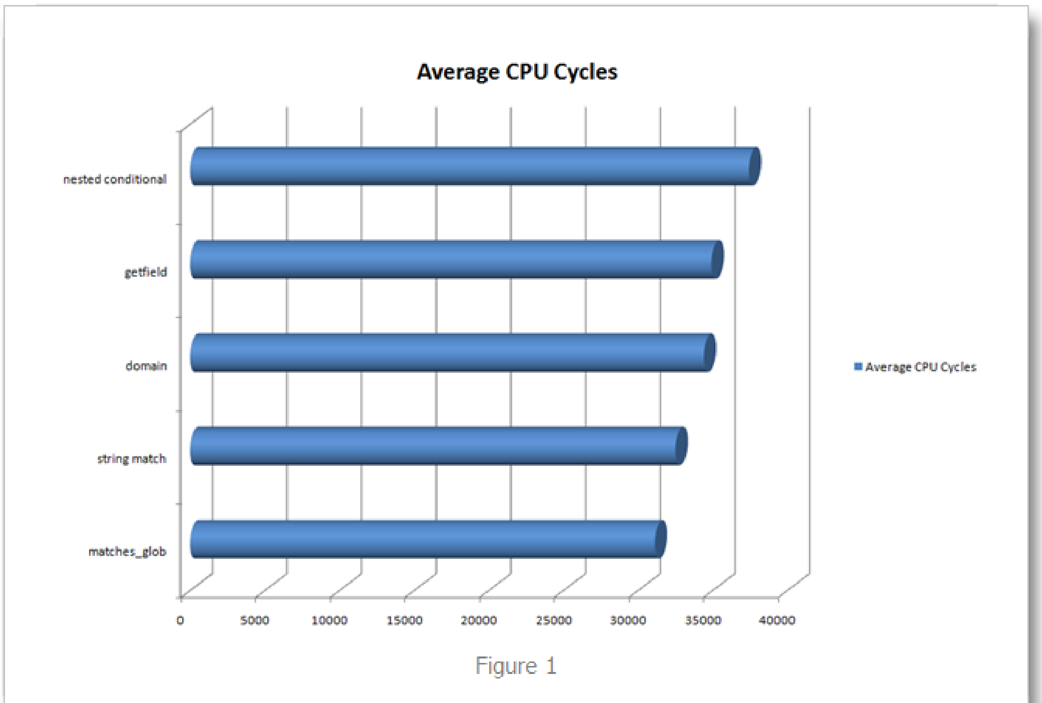

Figure 1

> "The reason is that operators are compiled directly into byte-code and hence no further parsing is necessary when the iRule is evaluated. In the [string match] command case, the string command can't be completely compiled into byte-code. Instead, the string command function is invoked. This function then also has to determine which arguments are options.

The second grouping of solutions, domain & getfield, are even closer in relation to each other, though significantly less efficient (as a group) than the first. The differences in domain and getfield are most likely due to the fact that the domain command has a built in split function (on "."), whereas with the getfield command, it must be supplied as an argument and evaluated. Actual numbers and efficiencies in relation to the other solutions are shown in the table below. Note that the percentages are in relation to the left column, so the matches_glob solution is 4.25% more efficient than the string match solution, and the domain solution is 10.55% less efficient than the matches_glob solution.

Approach	Average CPU Cycles	matches_glob	string match	domain	getfield	nested conditional
matches_glob	31116		4.25%	9.55%	10.89%	16.84%
string match	32496	-4.44%		5.53%	6.94%	13.15%
domain	34400	-10.55%	-5.86%		1.49%	8.06%
getfield	34919	-12.22%	-7.46%	-1.51%		6.67%
nested conditional	37416	-20.25%	-15.14%	-8.77%	-7.15%	

Conclusion

Be considerate of the weapons you have at your disposal in the iRules arsenal. Get the solution working, then comb every line and event looking for efficiencies. For example, I had never considered using the domain command to find an empty string. Test everything. Not only will you find efficiencies, you just might learn something as well.