iRules Optimization 101 - #02 - Expressions and Variables

Colin Walker, 2007-01-10

Building on the foundation that we laid down in last week's article, this installation of iRules Optimization 101 will focus on how to get the most out of the commands and expressions you're running within your iRules. Other articles in the series:

- iRules Optimization 101 – #01 – If, Elseif, and Switch
- iRules Optimization 101 – #02 – Expressions and Variables
- iRules Optimization 101 – #03 – For vs Foreach
- iRules Optimization 101 – #04 – Delimiters: Braces, Brackets, Quotes, and more
- iRules Optimization 101 – #05 – Evaluating iRule Performance

Two of the simplest, yet most effective ways to do this that I often discuss with people are the proper use of braces, and frugal use of variables. These are two things that are easily overlooked that can have a surprising impact on performance. As you might imagine, the more complex the rule, the more these practices matter as the performance gains are additive each time you implement the more efficient method over what might have been there otherwise. ;)

Section One: Braced Expressions

Anyone who's written code in most scripting languages is used to seeing braces "{ }" as a kind of delimiter. They're often used to delineate control structures, certain memory allocations, etc. As you've likely noticed, TCL makes liberal use of braces in many ways. Everything from defining your if statements to beginning and ending your events in your iRules (ok, that's us, not TCL...but you get the idea, and I had to make sure you were paying attention).

One of the many things that braces are used for in TCL is to encapsulate command arguments. This becomes extremely important in certain cases, as it A.) is often optional and B.) can have a huge performance impact. By making sure that we use braces when passing certain types of arguments to commands, we're able to stop the interpreter from trying to convert them, possibly multiple times, to different data types. This can be a huge performance sap depending on your code.

For example, a simple example of performing some math against an octet in an IP address might look something like:

```
when CLIENT_ACCEPTED {
  set newOct [expr 3 + [getfield [IP::client_addr] "." 4]]
  set total [expr 128 + $newOct]
...
}
```

While this is absolutely functional, and most people would look past the code without much thought, there is a huge amount of efficiency that can be gained here. By adding in braces around the values being passed to the expr command, we can stop unnecessary conversions, such as numeric -> string -> numeric, from occurring. This can mean a drastic performance difference, especially when implemented in a larger rule. The new rule would be very similar, with the addition of braces, and would read:

```
when CLIENT_ACCEPTED {
  set newOct [expr {3 + [getfield [IP::client_addr] "." 4]}]
  set total [expr {128 + $newOct}]
...
}
```

This not only allows for certain values, such as numeric values, to avoid undergoing multiple conversions out of and back to their original state, it also allows for immediate evaluation of variables by telling the interpreter that it needn't run through a second pass of evaluation as is necessary in some cases. I know it seems trivial, but this simple practice can gain you **almost an order of magnitude** in efficiency. I highly recommend taking a peek here for those of you who really want to delve into the nuts and bolts. The bottom line is, brace your expressions!

Section Two: Variable Allocation and Re-use

Another coding practice that will serve you well when aiming for optimal efficiency is the re-use of variables. I think it's safe to say that most people realize it is less expensive to re-use a variable than it is to allocate a new one. This common practice extends into the custom information made available to you in iRules by TMOS. The commands such as HTTP::host and IP::client_addr are not normal variable calls. They are calls to a piece of information already cached in memory by TMOS upon inspection.

This means that there is a large amount of effiency to be gained by avoiding unneeded variables by just not setting them in the first place, as well as making use of those special variables that we've made available to you, even if you're re-using it multiple times. For example:

```
when HTTP_REQUEST {
  set uri [HTTP::uri]
  set host [HTTP::host]
  set string "/login.php"
  if { ($host equals "bob.com") and ($uri equals $string)} {
    HTTP::redirect "https://[HTTP::host][HTTP::uri]"
  }
}
```

The above rule is again perfectly valid, it's just not very efficient. Notice how we set our own variables equal to information that TMOS has already cached in HTTP::uri and HTTP::host. Also, setting a variable equal to a constant string is just overhead that could be avoided. Try re-writing this kind of code like:

```
when HTTP_REQUEST {
  if { ([HTTP::host] equals "bob.com") and ([HTTP::uri] equals "/login.php")} {
    HTTP::redirect "https://[HTTP::host][HTTP::uri]"
  }
}
```

Not only is this a simpler, more elegant solution to read and debug, it's also far more efficient, as you are avoiding setting up three separate variables that really aren't needed. This means less used memory and less processing time to set/convert those values.

These tips may seem trivial, but you'll be surprised at the impact they have on your code's performance, which is especially important in a high-paced, wire-speed execution environment like iRules runs in. ;)

Get the Flash Player to see this player.

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