iRules logging to multiple locations with ease

dan-0, 2012-04-09

Some months back I was at an account where we were developing some iRules to provide logging detail. One of the complications was that some of the infrastructure to support remote logging was in the process of being implemented and was not immediately available. We also wanted to be able to log to local resources as an alternative and as the probing for requirements continued it appeared that the logging portion of the iRule was going to vastly outweigh the core functionality of the iRule.

Problem Domain

Typically when you have a choice between one or two items a simple if-else construct is sufficient to choose between the different options for logging. This can also be made a little more flexible by adding a variable so that all the logging can be changed from a single location.

For example:

```
1: when CLIENT_ACCEPTED {
2:     # Set the type of logging required- 0=local, 1=remote
3:     set logType = 0
4:
5:     if { $logType == 0 } {
6:         log -noname local0. "This is a local log event"
7:     else
8:         log 192.168.50.50 local0. "This is a remote event"
9:     }
10: }
```

In the example above, the variable logType provides the means to either log to the local syslog on the BigIP or to a remote syslog server. Examples like this can be seen throughout devcentral and are well understood. If the choices become larger then you risk getting stuck with an if-else-if block that compares the logging type variable to each possibility and then acts on it. A switch statement (reference) can be useful in cleaning up the code to make it easier to read and maintain:

```
1: switch logType {
2:      0
3:             {log -noname local0. "This is a local log event"}
4:      1
5:             {log 192.168.50.50 local0. "This is a remote event"}
6:      2
7:             { HSL::send $hsl $log}
```

```
8:       default
9:             {log -noname local0. "DEFAULT: This is a local log event"}
10: }
```

Most of the time solutions like these will be fine. Logging statements will get “turned on” or uncommented during the development and debugging cycle and then commented out when placed in production. This preserves the logging code should a problem materialize later or a modification be required that mandates scrutiny to evaluate the change. However writing your logging this way can be inefficient and cumbersome to maintain, especially if your iRule is or will become more extensive in the future. Additionally, if the requirements surrounding where and how you will log are fluid then you may be constantly revisiting the code which increases the chance that a functional mistake may be introduced when only a change to the logging was desired.

Another problem occurs when the choices for logging, in regards to the conjunction, become “and” as opposed to “or”. Say for example that you have the following choices with regards to logging:

logType	Requirement 1	Requirement 2	Requirement 3
No logging			
Local syslog	X	X	X
Remote syslog		X	
Remote HSL			X
STATS profile			X

Initially your requirement was to only log information to the local syslog. This is easy and takes little time to implement and maintain. Operational changes a few weeks later evolve the requirements so that remote syslog is also required in addition to local logging. This is still relatively easy but if you need to log data in 3-4 places your iRule is going to start to contain more logging code than functional code. Weeks after that, it is discovered that a single remote syslog is not adequate and that the amount of logging is detrimental to performance. It is desired that simple messages be logged locally, remote High Speed Logging is used for the heavy parts, and a stats profile will be used some of the time on a case by case, at our whim, basis. Now what?

Requirements such as these can, and should, be mitigated at design time but there are occasions when this is either not practical or the entire purpose of the iRule is logging of some sort. It’s important to be able to deliver the iRule in a manner that will be easy to maintain and modify but that also doesn’t have extensive branching code blocks just to log informational details. My solution to this was to take a page out of my previous development experiences which turned out to be a template for how I write iRules today.

Unix file permissions

Those familiar with a Unix file system know the Spartan mechanism for controlling access to files and directories. The system works by toggling bits in a large value that designate whether that specific permission is allowable for the resource it is configured for. In Unix permissions you will hear things like 644 or 777. These refer to whether you can read/write/execute as an owner/group/other user on the system. Each of those specific bits corresponds to the allowed permission and their location specifies who is granted that permission. The following table summarizes a single set of permissions:

Octal digit	**Text**	**Binary value**	**Result**
0	---	000	All access denied
1	--x	001	Execute only
2	-w-	010	Write only
3	-wx	011	Write and Execute
4	r--	100	Read access only
5	r-x	101	Read and Execute

6	rw-	110	Read and Write
7	rwx	111	Everything allowed

Applying this concept to our current problem, we get an elegant solution: Using bitwise operators, it's possible to select not only one OR another option, but any combination desired. More importantly, you can very quickly determine which option is relevant using bitwise operations (these are almost universally the fastest operations to perform in code) while avoiding several if-else-if blocks. If you are comfortable with bitwise operations you may want to skip down to the next sections otherwise read on.

Twiddling bits

Bitwise operations are very simple to understand but it helps to have a chart to cement the concept if you are not familiar with them. To the right is a simple chart that that shows the bitwise result of an AND operation. If both of the inputs are a "1" then the result is a "1" or "True". Otherwise the result of the operation will be a "0" or "False". We will expand on how this will be useful to us in a moment.

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

To get an example started, let's look at a few logging options and chart out how they will work. PAY CLOSE ATTENTION TO THE DECIMAL COLUMN:

Option	**Decimal**	**Binary Value**	**Result**
No Logging	0	000	No logging performed
Local Syslog	1	001	Syslog on local device
Remote syslog	2	010	Syslog to remote device
Remote HSL	4	100	Syslog using HSL

Notice how the Decimal value increases in powers of 2. This is important. If we expand on the previous table we can get our list of logging options we can select from:

Option	**Decimal**	**Binary Value**	**Result**
No Logging	0	000	No logging performed
Local Syslog	1	001	Syslog on local device
Remote syslog	2	010	Syslog to remote device
All syslog	3	011	Local and Remote syslog
Remote HSL	4	100	Syslog using HSL
Local / HSL	5	101	Local syslog and HSL
Remote / HSL	6	011	Remote syslog and HSL
Kitchen Sink	7	111	Everything

The above table fills in the blanks if we add in the additional decimal values that are missing and determine what the result would be. To see why this works look at the Binary value for the option "All Syslog". The value 011 would result in a Boolean "True" if compared to 1 and 2. This is also why on the previous table I mentioned that you should pay close attention to the Decimal values in the table. You want EACH option to be represented by a single binary digit – hence a power of two. Which options you select can be any value on this table.

Clear it up with an example

Enough theory, let's put some of this into code to determine how this will work:

```
1: when RULE_INIT {
2:     # Sets iRule debugging (0=no logging; 1=local syslog; 2=remote syslog; 4=remote HSL logging;)
3:     set static::log_enable 0
4:
5:     # logging resources
6:     set static::r_syslog_srv 192.168.50.50
7:     set static::hsl_pool HSL_POOL
8: }
9:
10: when CLIENT_ACCEPTED {
11:     # Create HSL handle if using HSL logging
12:     if {$static::log_enable & 4} { set hsl [HSL::open -proto UDP -pool $static::hsl_pool] }
13:
14:     # Log based on requested destination
15:     if {$static::log_enable & 1} {log -noname local0. "logged to local syslog" }
16:     if {$static::log_enable & 2} {log $r_syslog_srv local0. "logged to remote syslog" }
17:     if {$static::log_enable & 4} {HSL::send $hsl "logged to HSL"}
18: }
```

Before you can test this fully you will need to create a pool named HSL_POOL with at least one syslog server in it and a syslog server at 192.168.50.50 or an IP address of your remote syslog box. We have one small piece of housekeeping we need to do in regards to HSL logging and that is opening up the HSL channel. We do that by checking to see if we indeed are logging to HSL using a bitwise comparison and if that is true then opening the HSL channel. There is no HSL::close so there is no need to check at the end of the iRule to clean anything up.

Now we have a clean framework where we can easily change our logging option by modifying the log_enable variable. This also removes the end user from having to sort through lines of code and make changes that may not be obvious or well known to them. Formatting wise, this is a lot easier to read and understand and we also have the option to log to multiple places, or nowhere, if desired. Lastly, it would be easy to expand other options like a different remote syslog server or even programmatically changing the logging based on conditions.

A last thought is that we get this by only adding three lines where we want to log information to. If we were doing this with if-else-if blocks it would require 7 (we could ignore the no logging option as a moot optimization) blocks. Even worse, as you add more options your logging code would quickly grow out of control.

One last thing...

There is an unsupported command you can use to condense your logging even further. While this is an improvement to having a lot of logging code to maintain any time we add or modify our iRule we still have to add all the logging options each time we have a spot we want to log. In actuality, there is no requirement that says you need to put the same options in each location. It is perfectly valid that only certain log actions would apply in certain locations and not another. That being said, we can condense this even further by using a function. iRules unfortunately does not have a mechanism for creating and using your own function but we can use a tcl command called eval to get the same result. First, let's encapsulate our log message in a string:

```
1: …
2: when CLIENT_ACCEPTED {
3:     # Create HSL handle if using HSL logging
4:     if {$static::log_enable & 4} { set hsl [HSL::open -proto UDP -pool $static::hsl_pool] }
5:
6:     # Create log message
7:     set log "I log, therefore I am"
8:
9:     # Log based on requested destination
10:    if {$static::log_enable & 1} {log -noname local0. $log }
11:    if {$static::log_enable & 2} {log $r_syslog_srv local0. $log }
12:    if {$static::log_enable & 4} {HSL::send $hsl $log}
13: }
```

The eval command (reference) takes one or more arguments together and evaluates them as a tcl script and then returns the evaluation of that script. Keep in mind that this is not a supported mechanism by F5 and the need to use it MUST outweigh the challenge of migrating or dealing with incompatibilities as you update the OS. First the code:

```
1: when RULE_INIT {
2:     # Sets iRule debugging (0 = none; 1 = local syslog; 2 = remote syslog; 4 = remote HSL logging;)
3:     set static::log_enable 0
4:
5:     # logging resources
6:     set static::r_syslog_srv 192.168.50.50
7:     set static::hsl_pool HSL_POOL
8:
9:     # create log 'function'
10:    set static::logFunction {
11:        if {$static::log_enable & 1} {log -noname local0. $log }
12:        if {$static::log_enable & 2} {log $r_syslog_srv local0. $log }
```

```
12:         if {$static::log_enable & 2} {log $hsl_syslog_srv local0. $log }
13:         if {$static::log_enable & 4} {HSL::send $hsl $log}
14:     }
15: }
16: when CLIENT_ACCEPTED {
17:     # Create HSL handle if using HSL logging
18:     if {$static::log_enable & 4} { set hsl [HSL::open -proto UDP -pool $static::hsl_pool] }
19:
20:     # Create log message
21:     set log "I log, therefore I am"
22:
23:     # Log based on requested destination
24:     if { [catch {eval $static::logFunction;} ] } {
25:         log -noname local0. "Exception caught while trying to log"
26:     }
27: }
```

As you can see we have moved the block of code that logs up into the RULE_INIT event and essentially made the static variable logFunction equal to that block of code. In order to call our 'function' we set the variable log to the desired message and then call eval on the variable. Just to be careful, we catch any exceptions/errors from evaluating the script and log it locally. If you are not concerned with catching an error in the logging script, the last bit could be condensed further to:

```
1: # Log based on requested destination
2: catch {eval $static::logFunction;}
```

Keep in mind, nothing is for free. Creating the framework takes additional lines of code and chews up a tiny bit of memory. The eval command, depending on how verbose you get with the logging code, could get expensive from a performance perspective. All of these factors should be considered when using these techniques. I find that using a template built like this in my lab allows me to quickly implement an iRule and log in numerous places while debugging. I can easily modify how and where the logging goes and I can focus on the functional pieces quickly. For production, this code could easily be commented out or even wrapped in a debug flag or left in.

Beyond logging

This technique does not need to be confined to logging, although it does express the problem and solution very easily. This idea could also be used for functional constructs where five to ten different items may or may not be needed during the execution of an iRule but it cannot be determined until runtime. You can even wrap a block of code like this into a switch statement so that a procedural, state machine-like process can be modeled.

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