iRules: Disabling Event Processing

Deb Allen, 2008-10-06

The Problem

One of our customers was recently trying to use LTM with iRules to replace their proxy servers. They wondered if, rather than building one big iRule that contained all the required logic, they could break out functional pieces into individual iRules. It's a fairly common request and does make a lot of sense in cases where one function (such as inserting a custom header in all HTTP requests) would be applied to all virtual servers whereas other functions may only be required on some virtual servers.

What they were looking for was a command that allows an iRule to stop other iRules from running. That way they could create a set of rules which they could put in sequence either by defining them in the desired order on the virtual server resource list, or by setting event priority within the iRule itself.

The solution they wanted was this:

1. Insert the real client IP in a new HTTP header for all requests.
2. If the URI matches a specific pattern, then rewrite the URI a specific way and choose a pool
3. If the URI doesn't match, rewrite the URI a different way and choose a different pool

They wanted to prevent #3 from happeining if #2 already had, so they started by using a global variable (*uri_rewritten*) to track if a decision had been made yet. However, they were fairly certain that this was not the best way to accomplish their goal.

The Initial Solution

Here are the iRules they started with, using the variable flag and event priority to control the execution:

```
rule init_rewrites {
 when RULE_INIT {
  # Setup the global variable to track if a URL has already been re-written
  set ::uri_rewritten 0
 }
}
```

```
rule insert_custom_client_ip {
 # This rule is generic and needed on all virtuals
 when HTTP_REQUEST priority 10 {
  log local0.alert "Insert Client IP"
  HTTP::header insert "X-Forwarded-For" [IP::client_addr]
 }
}
```

```
rule generic_static_content_handler {
 when HTTP_REQUEST {
  # Extract the file extension
  set extension [string range [HTTP::path] [string last "." [HTTP::path]] [string length [HTTP::path]

  # If the extension matches against the class then re-write with the appropriate directory
  if { [matchclass $extension equals $::static_content]   } {
   set new_path [format "%s%s" "/common" [HTTP::path]]
   HTTP::path $new_path
   pool static_pool
```

```
   pool static_pool
   set ::uri_rewritten 1
  }
 }
}
```

```
rule default_rewrite {
 when HTTP_REQUEST priority 1000 {
  # If the request hasn't already been re-written by a previous rule then rewrite it with this defaul
  if { $::uri_rewritten equals 0 } {
   set new_path [format "%s%s" "/proxy" [HTTP::path]]
   HTTP::path $new_path
   pool test_http_pool
  }
  set ::uri_rewritten 0
 }
}
```

What they really wanted to do, though, was to prevent the execution of the *default_rewrite* iRule entirely if the *generic_static_content_handler* iRule matched and re-wrote the URL already. (It's worth mentioning that a global variable would actually not work as intended here, as it would be shared by all connections, resulting in false positives for some connections processing in parallel. For a connection-specific flag, a local variable could be used in this manner.) But there is a better way.

A Better Solution: The "event" command

The event command is what they are looking for. It has a "*disable*" option that supports disabling specific events or all events for the remainder of that connection. If only selected events are disabled, they can be re-enabled from within another event using the coresponding "*enable*" option.

With that in mind, the set of iRules above could be adjusted just slightly to allow the iRules engine to "bail out" of the ruleset if an early match is seen for a request.

For starters, we no longer need the *init_rewrites* iRule, since the global variable it initializes for connection control is no longer needed.

```
~~rule init_rewrites {~~
~~ when RULE_INIT {~~
~~  # Setup the global variable to track if a URL has already been re-written~~
~~  set ::uri_rewritten 0~~
~~ }~~
~~}~~
```

The *insert_custom_client_ip* iRule is meant to apply to all connections, and it should run first, so we will leave it as is, including the priority 10, which will cause it to execute before any iRules with a higher priority (500 is the default priority).

```
rule insert_custom_client_ip {
 # This rule is generic and needed on all virtuals
 when HTTP_REQUEST priority 10 {
  log local0.alert "Insert Client IP"
  HTTP::header insert "X-Forwarded-For" [IP::client_addr]
 }
}
```

The *generic_static_content_handler* iRule should disable the HTTP_REQUEST event for this connection instead of setting the variable flag, and can still run at default priority 500:

```
rule generic_static_content_handler {
 when HTTP_REQUEST {
  # Extract the file extension
  set extension [string range [HTTP::path] [string last "." [HTTP::path]] [string length [HTTP::path]

  # If the extension matches against the class then re-write with the appropriate directory
  if { [matchclass $extension equals $::static_content]   } {
   set new_path [format "%s%s" "/common" [HTTP::path]]
   HTTP::path $new_path
   pool static_pool
  # disable the current event only (HTTP_REQUEST) for this connection
  event disable
   }
 }
}
```

The *default_rewrite* iRule will still run at priority 1000 (which is highest possible event priority value, so it will run last) but can now be modified to remove the checking and setting of the flag variable, since it will now run only if a match was not found in the previous iRule:

```
rule default_rewrite {
 when HTTP_REQUEST priority 1000 {
  # If the request hasn't already been re-written by a previous rule then rewrite it with this defaul
  if { $::uri_rewritten equals 0 } {
   set new_path [format "%s%s" "/proxy" [HTTP::path]]
   HTTP::path $new_path
   pool test_http_pool
  }
  set ::uri_rewritten 0
 }
}
```

Finally, we will need to add one more small iRule that runs only in the HTTP_RESPONSE event, re-enabling the HTTP_REQUEST event for the connection once the response is seen:

```
rule enable_HTTP_REQUEST_on_response {
 when HTTP_RESPONSE {
  event enable HTTP_REQUEST
 }
}
```

This last addition allows the iRule to continue to process any additional HTTP requests seen later on the same Keep-Alive connection. It's also the reason I didn't use "event disable all" in the *generic_static_content_handler* iRule: Because we needed to re-enable iRule processing for follow-on requests after this request completes, and if all events are disabled, there would be no opportunity to do so.

Get the Flash Player to see this player.

200806010-iRulesDisablingEvents.mp3

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