Conditioning iRule Logic on External Information - #01 - HTTP::retry

Deb Allen, 2007-18-09

Customers frequently ask if they can condition their iRule logic on information retrieved real-time from an external source. In most cases, you really can't.

Most external calls are blocked
If an external call were to take place inside an iRules event, it would pause the connection until the call returned or until the configured idle timeout expired. Of course that would introduce latency into the connection, but more importantly, during that time the system resources remain allocated to the connection. Since there is no way to efficiently manage those idle resources in this context, our developers have disabled nearly all of the TCL commands that would allow you to make an external call that pauses the connection.

But there are some exceptions
The caveat there, in case you missed it, was "If an external call were to take place inside an iRules event...". By taking advantage of the existing commands & event triggers, there are a few ways in which iRules CAN reach out to another system OUTSIDE the context of an active iRule event for information on which to condition traffic management.

The most commonly used of these is LTM authentication of client requests. Authentication iRules commands and events have been provided so iRules can manage each connection based on authentication details discovered during the authentication transaction. We'll cover that in a future article.

Another more recently added capability is DNS name resolution. The NAME commands and the NAME_RESOLVED event allow iRules to manage connections based on forward & reverse DNS resolution. We'll cover that in a separate article as well.

Today I want to show you an interesting way to use the HTTP::retry command to reach out for external information that iRules can use to decide the fate of the connection.

Using HTTP::retry to make an external call
The HTTP::retry command was originally introduced in LTM v9.2.0, and was meant to allow an iRule to replay the same request to a new server when a load balancing attempt failed. However, we've since discovered that HTTP::retry can be used to replay the same or different requests to more than one host per connection, and LTM doesn't know the difference between a real retry and a completely fabricated out-of-band request. So if the original request is HTTP, AND the external information can be accessed via HTTP, you can use HTTP::retry to either forward the request through a secondary device, then forward that response to the client, or you can conditionally manage the original request based on a response returned by a different device.

An example...
Here's an example of the latter, based on a recent request. The customer wanted to look up some detail related to the client's IP address in a remote database, and manage the traffic based on the response from the database server.

```
when RULE_INIT {
  # set string to match for valid connection
  set valid_string "This client is A-OK!"
}
when CLIENT_ACCEPTED {
  # set flag to control logical flow. 1 means lookup is pending.
  set lookup 1
  # set string to match for valid connection
```

```
   set valid_string "This client is A-OK!"
}
when HTTP_REQUEST {
  # If each request on the same connection must force a lookup,
  # re-initialize the value of the flag here
  # In this case, lookup result based on client IP is good for
  # the life of the connection, so we'll leave the flag alone here.
  # set lookup 1
  if {$lookup == 1} {
    # if client hasn't already been looked up, save the
    # request so we can replay it to the LB server later;
    set LB_request [HTTP::request]
    # inject lookup URI in place of original request;
    HTTP::uri "/client_lookup.pl?ip=[IP::client_addr]"
    # and send the out-of-band validation query to the DB_pool.
    pool DB_pool
  } else {
    # otherwise, send the request to the LB pool
    pool LB_pool
  }
}
when HTTP_RESPONSE {
  # If lookup flag is still on in response event, this is the response
  # to the lookup, so we collect entire payload (up to 1MB limit)
  # both to evaluate the DB server response and to prevent this response
  # from being returned to the client.
  # Already-validated connections will bypass the rest of the rule.
  if {$lookup == 1}{
    if {[HTTP::header exists Content-Length] && \
     ([HTTP::header Content-Length] < 1048576)}{
       set clength [HTTP::header Content-Length]
    } else {
       set clength 1048576
    }
    HTTP::collect $clength
  }
}
when HTTP_RESPONSE_DATA {
  # HTTP_RESPONSE_DATA will only be triggered for a DB lookup.
  # (All other requests have already been forwarded to the LB pool.)
  # If response from DB indicates connection is valid, reset the
  # lookup flag & replay the request to the LB server.
  # Otherwise, reject the connection
  if {[HTTP::payload] contains $valid_string}{
    pool LB_pool
    HTTP::retry $LB_request
  } else {
    reject
  }
  # depending on the app, reset the value of the flag here
  # in case of new request on same connection
  # (useful for situations where each URI requires a lookup)
  # In this case, lookup result based on client IP is good for
  # the life of the connection, so we'll leave the flag set to 0
  set lookup 0
```

```
}
```

Other ideas...

Of course there are a number of other ways the same approach could be used. The response from the lookup could contain the name of a pool, or a value from which a pool could be looked up from a class, as in this example: http://devcentral.f5.com/Default.aspx?tabid=53&view=topic&forumid=5&postid=14001 . The logic could also be adjusted to send the original request to the LB pool, then use HTTP::retry to send that response through another device (such as a transcoder), then ultimately returning the encoded data back to the client.

PLEASE NOTE - Your Mileage May Vary: Any request to an external data source mid-connection may introduce unacceptable latency, and is ultimately dependent on the availability of the external resource. In this particular example, if no member of the DB pool is available to answer queries, the virtual server will reject all connections. You can minimize the impact of the external call by providing redundant, highly available servers close to the load balancer. As with all iRules, be sure to test under load before deploying in production for critical sites.

Happy coding!

/deb