#The101: Routing

JRahm, 2012-11-10

#The101: iRules was intended to be a journey through the basics of both iRules and programming concepts alike, with the intent to get everyone up to speed on the necessary topics to tackle iRules in all their glory. Whether you're a complete newbie to scripting or a seasoned veteran, we want everyone to be able to enjoy iRules equally, or at least as near as we can manage. Hopefully the series has been helpful to that end thus far. In case you haven't been following along, or in the even that you're in need of a recap, here's what we've covered so far:

1. Introduction to Programming & Tcl
2. Introduction to F5 Technology & Terms
3. Introduction to iRules
4. Events & Priorities
5. Control Structures & Operators
6. Variables
7. Logging & Comments

In this eighth entry into the series we're taking a look at what I think is likely the thing that many people believe is the primary if not one of the sole uses of iRules before they are indoctrinated into the iRuling ways: Routing. That is, directing traffic to a particular location or in a given fashion based on … well, just about anything in the client request. Of course, this is only scratching the surface of what iRules is capable. There's content manipulation, security, authentication…well, there's a huge amount that iRules can do beyond routing. That being said, routing and the various forms that it can take with a bit of lenience as to the traditional definition of the term, is a powerful function which iRules can provide. There are many different ways in which you can use iRules to affect the routing of traffic.

First of all, keep in mind that the BIG-IP platform is a full, bi-directional proxy, which means we can inspect and act upon any data in the transaction bound in either direction, ingress or egress. This means that we can technically affect traffic routing either from the client to the server or vice versa, depending on your needs. For simplicity's sake, and because it's the most common use case, let's keep the focus of this article to only dealing with clientside routing, e.g. routing that takes place when a client request occurs, to determine the destination server to deliver the traffic to.

Even looking at just this particular portion of the picture there are many options for routing from within iRules. Each of the following commands can change the flow of traffic through a virtual, and as such is something I'll attempt to elucidate:

· **pool**

· **node**

· **virtual**

· **HTTP::redirect**

· **reject**

· **drop**

· **discard**

As you can see there are some very different commands here, not all of them what you would consider traditional "routing" style commands, but each has a say in the outcome of where the traffic ends up. Let's take a look at them in a bit more detail.

pool

```
pool <pool_name>
```

The pool command is probably the most straight-forward, “bread and butter” routing command available from within iRules. The idea is very simple, you want to direct traffic, for whatever reason, to a given pool of servers. This simple command gets exactly that job done. Just supply the name of the pool you’d like to direct the traffic to and you’re done. Whether your criteria for allowing traffic to a given pool, or whether you’re trying to route traffic to one of several different pools based on client info or just about anything else, an iRule with a simple pool command will get you there. The pool command is the preferred method of simple traffic routing such as this because by using the pool construct you’re getting a lot of bang for your buck. The underlying infrastructure of monitors and reporting and such is a powerful thing, and shouldn’t be overlooked.

There is, however, one other permutation of this command:

```
pool <pool_name> [member <addr> [<port>]]
```

Perhaps you don’t want to route to one of several pools, but instead want to route to a single pool but with more granularity. What if you don’t want to just route to the pool but to actually select which member inside the pool the traffic will be sent to? Easy enough, specify the “member” flag along with with the IP and port, and you’re set to go. This looks like:

```
when CLIENT_ACCEPTED {
  if { [IP::addr [IP::client_addr] equals 10.10.10.10] } {
    pool my_pool
  }
}
```

```
when HTTP_REQUEST {
  if { [HTTP::uri] ends_with ".gif" } {
    if { [LB::status pool my_Pool member 10.1.2.200 80] eq "down" } {
      log "Server $ip $port down!"
      pool fallback_Pool
    } else {
      pool my_Pool member 10.1.2.200 80
    }
  }
}
```

node

```
node <addr> [<port>]
```

So when directing traffic to a pool or a member in a pool, the pool command is the obvious choice. What if, however, you want to direct traffic to a server that may not be part of your configuration? What if you want to route to either a server not contained in a particular pool, whether that's bouncing the request back out to some external server or to an off the grid type back-end server that isn't a pool member yet? Enter the node command. The node command provides precisely this functionality. All you have to do is supply an IP address (and a port, if the desired port is different than the clientside destination port), and traffic is on its way. No pool member or configuration objects required.

Keep in mind, of course, that because you aren't routing traffic to an object within the BIG-IP statistics, connection information and status won't be available for this connection.

```
when HTTP_REQUEST {
  if { [HTTP::uri] ends_with ".gif" } {
    node 10.1.2.200 80
  }
}
```

virtual

```
virtual [<name>]
```

The pool command routes traffic to a given pool, the node command to a particular node…it stands to reason that the virtual command would route traffic to the virtual of your choice, right? That is, in fact, precisely what the command does – allows you to route traffic from one virtual server to another within the same BIG-IP. That's not all it does, though. This command allows for a massively complex set of scenarios, which I won't even try to cover in any form of completeness. A couple of examples could be layered authentication, selective profile enabling, or perhaps late stage content re-writing post LB decision.

Depending on your needs, there are two basic functions that this command provides. On is to add another level of flexibility to allow users to span multiple virtuals for a single connection. This command makes that easy, taking away the tricks the old timers may remember trying to perform to achieve something similar. Simply supply the name of whichever virtual you want to be the next stop for the inbound traffic, and you're set.

The other function is to return the name of the virtual server itself. If a virtual name is not supplied the command simply returns the name of the current VIP executing the iRule, which is actually quite useful in several different scenarios. Whether you're looking to do virtual based rate limiting or use some wizardry to side-step SSL issues, the CodeShare has some interesting takes on how to make use of this functionality.

```
when HTTP_REQUEST {
  # Send request to a new virtual server
  virtual my_post_processing_server
}
```

```
when HTTP_REQUEST {
  log local0. "Current virtual server name: [virtual name]"
}
```

HTTP::redirect

```
HTTP::redirect <url>
```

While not something I would consider a traditional routing command, the redirection functionality within iRules has become a massively utilized feature and I'd be remiss in leaving it out, as it can obviously affect the outcome of where the traffic lands. While the above commands all affect the destination of the traffic invisibly to the user, the redirect command is more of a clientside function. It responds to the client browser indicating that the desired content is located elsewhere, by issuing an HTTP 302 temporary redirect.

This can be hugely useful for many things from custom URIs to domain consolidation to … well, this command has been put through its paces in the years since iRules v9. Simple and efficient, the only required argument is a full URL to which the traffic will be routed, though not directly. Keep in mind that if you redirect a user to an outside URL you are removing the BIG-IP and as such your iRule(s) from the new request initiated by the client.

```
when HTTP_RESPONSE {
  if { [HTTP::uri] eq "/app1?user=admin"} {
    HTTP::redirect "http://www.example.com/admin"
  }
}
```

reject

reject

The reject command does exactly what you'd expect: rejects connections. If there is some reason you're looking to actively terminate a connection in a not so graceful but very immediate fashion, this is the command for you. It severs the given flow completely and alerts the user that their session has been terminated. This can be useful in preventing unwanted traffic from a particular virtual or pool, for weeding out unwanted clients all-together, etc.

```
when CLIENT_ACCEPTED {
  if { [TCP::local_port] != 443 } {
    reject
  }
}
```

drop & discard

drop (or discard)

These two commands have identical functionality. They do effectively the same thing as the reject command, that is, prevent traffic from reaching its intended destination but they do so in a very different manner. Rather than overtly refusing content, terminating the connection and as such alerting the client that the connection has been closed, the discard or drop commands are subtler. They simply do away with the affected traffic and leave the client without any status as to the delivery. This small difference can be very important in some security scenarios where it is far more beneficial to not notify an attacker that their attempts are being thwarted.

```
when SERVER_CONNECTED {
  if { [IP::addr [IP::client_addr] equals 10.1.1.80] } {
    discard
    log local0. "connection discarded from [IP::client_addr]"
  }
}
```

Routing traffic is by no means the most advanced or glamorous thing that iRules can do, but it is valuable and quite powerful nonetheless. Combing advanced, full packet inspection with the ability to manipulate the flow of traffic leaves you with near endless options for how to deliver traffic in your infrastructure, and allows for greater flexibility with your deployments. This kind of fine grained control is part of what leads to a tailored user experience which is something that iRules can offer in a very unique and powerful way.

In the next installment of #The101: iRules we'll dig into another massively popular and widely utilized set of functionality – string manipulation.

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