BIG-IP® Service Provider: Generic Message Administration

Version 13.1

Table of Contents

Introduction to MRF Generic Message Protocol

Introduction

A BIG-IP® system provides advanced Message Routing Framework (MRF) capabilities. The BIG-IP Generic Message Protocol is a prototype implementation of a protocol filter compatible with MRF. It provides iRule commands to enable a script writer to parse an input stream into messages to be routed by the Message Routing Framework. It also provides a route table implemention supporting both static routes and dynamic routes.

This guide is designed to introduce the reader to Generic Message Configuration Objects as well as provide an example of how to use Generic Message.

Capabilties

The Generic Message protocol provides a sample implementation of a message routing protocol filter. This section provides a concise summary of the BIG-IP® Generic Message solution:

- Accepting client connections from a peer
- Establishing connections with a peer
- Asking the peer for its routing name
- Adding a dynamic route for the peer
- Registration of static routes to pool
- Separating packets into messages to be routed
- Designating a source and destination address for messages
- Support for both request and answer messages
- Appropriately setting the nexthop attribute of an answer message to the lasthop of the request message to instruct the message router to bypass route lookup.
- Ability for an iRule script to make the route selection and bypass route lookup.
- Ability for an iRule script to set and/or modify the source and destination addresses of a message.
- Ability for an iRule script to create a new message.
- Ability for an iRule script to drop a message.

A genericmessage profile is currently available via tmsh. There is currently no plan to include this in iControl or GUI.

Warning

Warning: *Generic Message should not be accessed though the GUI. It is a TMSH only feature. If the Generic Message Virtual Server is saved through the GUI it will cause Generic Message to stop functioning.*

Configuration Objects

Configuration Objects

MRF

The Message Routing Framework (MRF) is an extension to the BIG-IP® system to support routing of messages between connections. MRF is a protocol independent framework that is used to build protocol specific routing systems. Currently three protocol implementation of MRF are supported by the BIG-IP system: DIAMETER, SIP and Generic Message.

Virtual Server

A virtual server is a traffic-management object on the BIG-IP® system that is represented by an IP address and a service. Clients on an external network can send application traffic to a virtual server, which then directs the traffic according to configuration instructions.

All virtuals sharing the same router profile will share the same router instance. This means that they will be able to use connections created by traffic on other virtuals. They will also be able to route message between each other.

The virtual server configuration contains a destination address and mask which specifies what IP addresses and port the virtual server will listen for incoming packets. The virtual server object also contains a source address allowing it to limit packets to those packets that originate from a range of devices.

The behavior of a virtual server is determined by the set of profiles attached to the virtual server. In addition, the behavior of a virtual can further be extended by assigning iRules to the virtual. Furthermore, the transport-config object is used to define the outgoing connection, for example source address translation, and translation modes of the virtual server. Note, the virtual server configuration (e.g. pool and persistence profile) used to select a destination is not used in MRF.

Transport Config

A transport config defines the parameters of a new outgoing connection. It is a subset of a virtual server used to define parameters of an outgoing connection. This includes the profiles, iRules and source-address-translation settings.

Since a transport-config may be used in a route that may be used in multiple router instances, the router profile is not defined in a transport-config object. Instead, an outgoing connection inherits the router profile of the router instance that creates an outgoing connection.

Transport Config Attributes

Table 1: Transport Config Attributes

Attribute	Description	Default
ip-protocol	Specifies the ip protocol. This will be automatically set based on the transport profile added. This value is read-only.	none

Attribute	Description	Default
source-port	Specifies the source port to be used for the connection being created. If the source-port is zero, an ephemeral port will be used.	0
profiles	The transport protocol and the protocol-specific profile associated with this outgoing connection.	
source-address-translation	Specifies the source-address-translation type and the pool.	
rules	List of iRules associated with this outgoing connection.	none

Source Address Translation

Table 2: Transport Config Attributes

Sub-Attribute	Description	Default
type	Specifies the type of source address translation to perform	automap
pool	Specifies the name of the snap pool	none

Source Address Translation Types

Table 3: Source Address Translation Types

Type	Description
automap	The self-ip of the outgoing vlan will be used as the source address of the outgoing connection.
snat	A source address will be selected from the named snat pool
none	No source address translation will be performed.

Route

The route object is used to determine the best route to use for forwarding a message. Each route contains a source and destination address fields that are matched against the source and destination address metadata fields of the message. If the route's source or destination address field is empty, it is considered a wildcard and all values of the message's corresponding address field will be considered matching. An actual match of the contents of an address field is scored higher than a wildcard match.

The route object also contains a list of peers and a peer-selection-mode to specify how peers in the peer list are selected.

Generic Message Route Object

```
ltm message-routing generic route <name> {
   destination-address <string>
   source-address <string>
   peer-selection-mode <ratio/sequential>
   peers {
      <peers>
   }
}
```

Attribute

Table 4: Generic Message Route Object Attributes

Attribute	Description	Default
destination-address	The destination-address of the route. This is often the name of the receipent of a message	none
source-address	The source-address of the route. This is often the name of the originator of a message	none
peer-selection-mode	This specifies the method to select a peer from the provided list of peers. Possible values are: • sequential: The first peer in the list of peers is selected. If the message is retried, the next peer in the list will be used. • ratio: The probability of a peer from the list being selected will be based on the relative ratio values of each peer.	sequential
peers	Provides the list of peers used by this route	none

Example

```
# default route - source and destination address are wildcard
ltm message-routing generic route default_route {
    peer-selection-mode sequential
    peers { default_peer, backup_peer }
}

ltm message-routing generic route forwarding_route {
    destination-address help
    peers { forwarding_peer }
}

ltm message-routing generic route bad_route {
    source-address annoying_user
    peer-selection-mode ratio
    peers { default_peer, blackhole_peer }
}
```

Route Table

Generic message maintains a route table per router instance. The route table contains three types of routes, static routes from configuration, dynamic routes auto generated from peer names per connection and dynamic routes added by an iRule.

The route table will use the source and destination address of a message to determine the best matching route to use for forwarding the message. The source and destination attributes of the route will be matched against the source and destination addresses of the route to generate a score. The highest scoring route is used for forwarding the message.

Table 5: Generic Message Route Table

Matching Fields	Score
Both src_addr + dst_addr matches	3
dst_addr matches	2
src_addr matches	1
no attributes match	0

Static Route

Generic message allows for static routes to be loaded from configuration. A static route contains a list of peers, where each peer contains a list transport-config and a pool.

Pool

A standard LTM pool is used to list a set of devices that messages may be routed towards.

```
ltm pool my_default_pool members { 10.1.2.1:1234, 10.1.2.2:1234, 10.1.2.3:1234 }
ltm pool my_backup_pool members { 10.1.2.4:1234, 10.1.2.5:1234, 10.1.2.6:1234 }
ltm pool my_empty_pool members { }
```

Peer Object

A peer object is used to define a set of hosts and the the method to connect with them. Peers are used to create static routes. The peer structure is protocol independent while each protocol implementation of MRF will define its own static route structure. Furthermore, a peer object can be used to configure MRF to create outgoing connections to members of a pool, at startup, and to retry creating connection at a configurable interval (in milliseconds). In summary, peers contain a pool, a transport-config and connection distribution settings.

```
ltm message-routing generic peer default_peer {
   transport-config my_snat_tc
   pool my_default_pool
   connection-mode per-tmm
   number-connections 2
   auto-initialization enabled
   auto-initialization-interval 1000
}
```

Transport-config Not Defined

If a transport-config is not defined, the outgoing connection will use the settings of the incoming connection for creating the outgoing connection.

```
ltm message-routing generic peer backup_peer {
   pool my_backup_pool
}
```

Message Forwarding

If a pool is not defined, the outgoing connection will use the destination (local address) of the incoming connection as the destination (remote address) of the outgoing connection. This is how to configure message forwarding.

```
ltm message-routing generic peer forwarding_peer {
   transport-config my_nosnat_tc
}
```

Black Hole Route

If a pool with no pool members is used, the message routed to that peer will fail routing. This can be used as a method to create a black hole route.

```
ltm message-routing generic peer blackhole_peer {
   ratio 3
   pool my_empty_pool
}
```

Peer name

Each connection is assigned a peer name. This peer name must be set via the GENERICMESSAGE::peer name iRule command. Once the peer name has been sent, the connection is able to receive and forward messages.

The peer name is used to add a dynamic route to the route table. It is also used to set the source address of each message.

Peer Attributes

Table 6: Generic Message Route Table

Attribute	Description	Default
pool	Pool associated with the peer. If only one peer, then configure a single-member pool. If none, the message will be forwarded to the destination address and port of the originating connection.	none
transport-config	Specifies the transport-config that defines the parameters of the outgoing connection. If none, the parameters of the originating connection will be used to create the outgoing connection.	none
connection-mode	Specifies how the number of connections per peer are to be limited as follows: per-peer, per-blade, per-tmm, per-client. If a transport config is not specified, the attributes of the originating connection of the message being routed will be used to create the outgoing connection. In this case, the connection-mode in the peer object will be ignored.	per-peer
number-connections	Specifies the number of connections between the BIG-IP® system and a peer. If a transport config is not specified, the attributes of the originating connection of the message being routed will be used to create the outgoing connection. In this case, the number-connections in the peer object will be ignored.	1
ratio	Used to designate the ratio of this peer when used within a route with a peer-selection-mode of ratio.	1

Attribute	Description	Default
auto-initialization	If enabled, the BIG-IP® system will automatically create outbound connections to the active pool members in the specified pool using the configuration of the specified transport-config. For auto-initialization to attempt to create a connection, the peer must be included in a route that is attached to a router instance. For each router instance that the peer is contained in, a connection will be initiated. The auto-initialization logic will verify at a configurable interval if the a connection exists between the BIG-IP system and the pool members of the pool. If a connection does not exist, it will attempt to reestablish one.	disabled
auto-initialization-interval	Specifies the interval (in milliseconds) that attempts to initiate a connection occur. Valid ranges are from 500ms to 65535ms	5000ms

Connection Modes

Per Peer

A BIG-IP® system will make just one connection to a peer. This means that only one TMM is connected to each Peer. While this connection mode uses fewer connections it will introduce latency. This will happen when messages are disaggregated to the wrong TMM and must be forwarder. The following diagram provides additional detail.

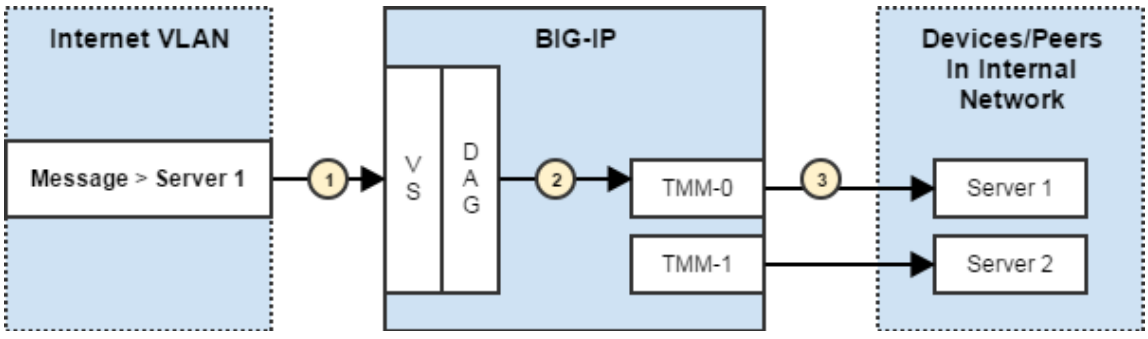

Figure 1: Optimum scenario

1. Message arrives on a Virtual Server (VS)
2. Message is disaggregated to TMM-0
3. TMM-0 is connected to the correct server so the message is sent

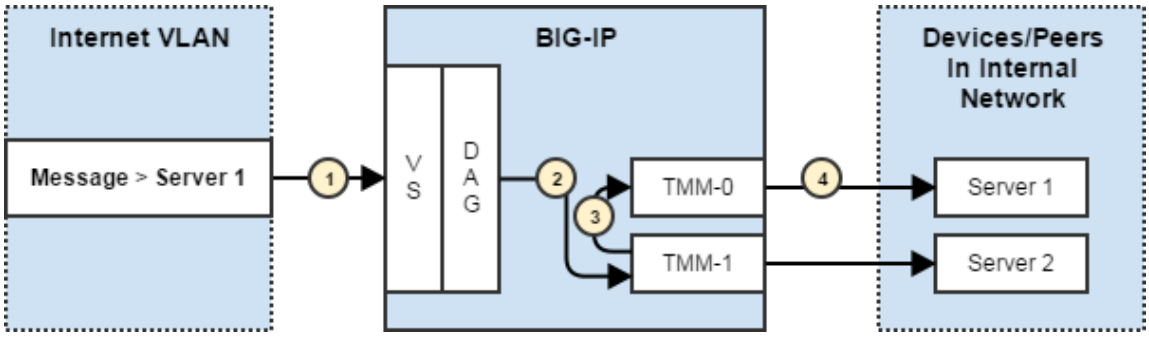

Figure 2: Performance impacted scenario

1. Message arrives on a Virtual Server (VS)
2. Message is disaggregated to TMM-1
3. TMM-1 is not connected to Server 1 so message must be forwarded to the correct TMM. This will introduce latency.
4. TMM-0 is send message to Server 1

Per TMM

A BIG-IP® system will make a connection from every TMM to the same peer. This means a machine with 8 cores will have 8 connections per peer. While this increases the number of active connections, it also improves performance because there is no need to forward messages between TMMs.

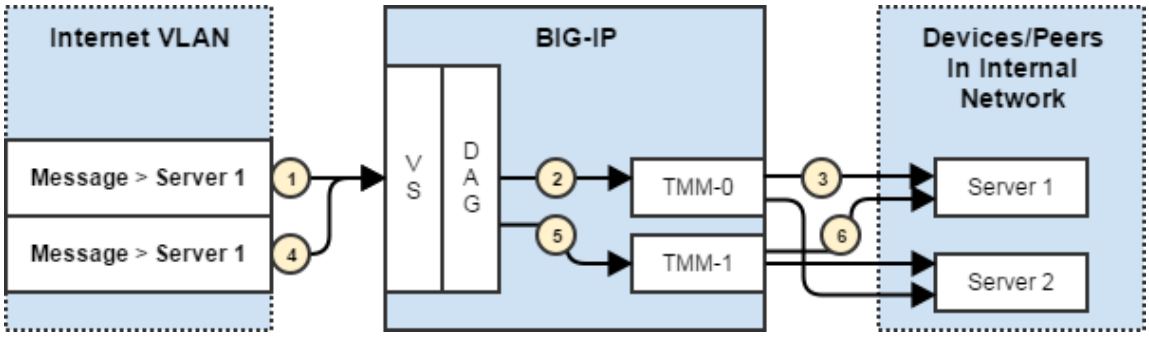

Figure 3: Every TMM is connected to every peer which decreases latency but increases the number of connections

1. Message arrives on a Virtual Server (VS)
2. Message is disaggregated to TMM-0
3. TMM-0 is connected to the correct server so the message is sent
4. Second message arrives
5. Message is disaggregated to TMM-1
6. TMM-1 is connected to the requested server so the message can be sent directly

Per Blade

A BIG-IP® system creates one connection per blade to each peer. This provides a balanced performance approach between the per peer connection mode (only one connection) and a per tmm connection mode (a connection from each TMM). This mode only makes sense for a hardware chassis with multiple blades.

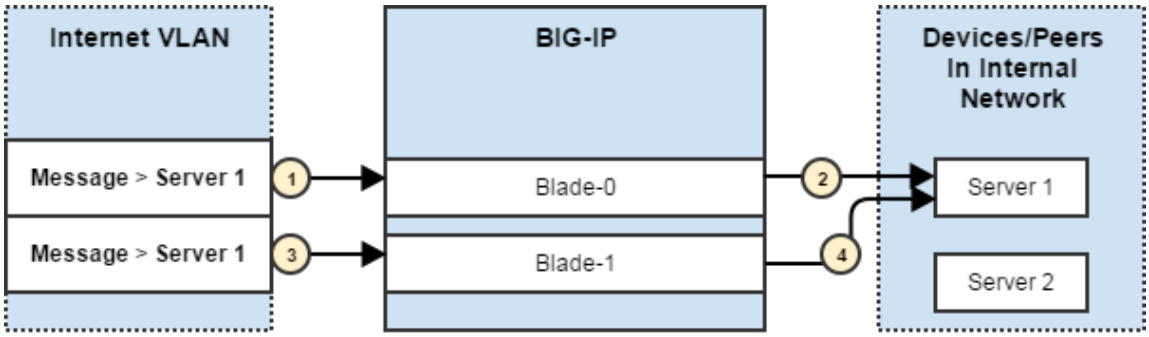

Figure 4: Each blade will make a single connection to each peer.

1. Message arrives on Blade 0
2. Blade 0 opens a connection to Server 1 and forwards the message
3. Second message arrives
4. Blade 1 opens a connection to server 1 and forwards the message

Note: *A connection will not be opened to Server 2 until a message targeted at that server arrives.*

Per Client

A BIG-IP® system will create a new outgoing connection for each client. Note, clients do not share connections.

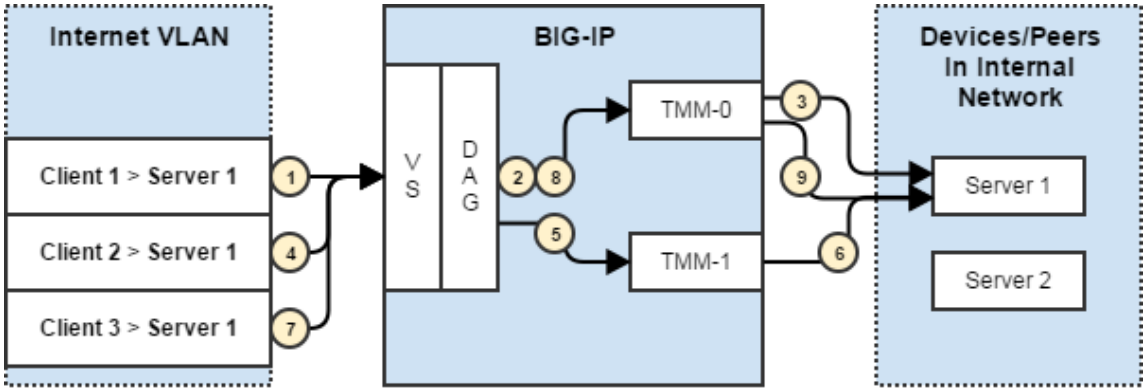

Figure 5: Connections will be created for each client.

1. Client 1 message arrives on a Virtual Server (VS)
2. Message is disaggregated to TMM-0
3. TMM-0 connects to Server 1 and sends the message
4. Second message arrives
5. Client 2 message is disaggregated to TMM-1
6. TMM-1 connects to Server 1 and send the message
7. Third message arrives
8. Client 3 message is disaggregated to TMM-0
9. TMM-0 creates a new connection for Server 1 for Client 3

Note: *A connection will not be opened to Server 2 until a message targeted at that server arrives.*

Message

The Generic Message implementation of MRF provides a series of iRule commands for generating and populating a message. A message container wraps data that will be routed between connections. Associated with the message are meta-data fields used to assist routing of the message.

Message Types

Generic Message understands three types of message types: request, response and notifications.

Request Message

Request messages are messages that expect a response to be returned from the destination. A request message contains payload data and a source and destination address. The source address will be auto populated by Generic Message with the peer name of the originating connection. The destination address will need to be populated by the script. The source and destination address will be used by the route table to find the best matching route to use for delivering the message.

Setting the destination address of a message will identify the message as a request message. The destination address is often generated by data contained in the message.

Response Message

Response messages are messages returned by the destination to be delivered to the originator of the request. A message without a destination address is assumed to be a response message.

Generic message will assign a request-sequence-number to all request messages before raising GENERICMESSAGE_EGRESS event. The details of the originating flow (TMM and flow_id) of the request will be stored in a pending request table on the outgoing connection.

Each response message is initially assigned the oldest request-sequence-number in the pending request queue before raising GENERICMESSAGE_INGRESS event. Upon completion of the event the message will be forward to stored connection from the pending request table using the request-sequence-number.

If the pending request table is empty, the response message will be routed via the route table.

The default response routing implementation assumes that responses will arrive in the order that the requests were forwarded. If response message can arrive out of order, the script author will need to set the appropriate request-sequence-number during GENERICMESSAGE_INGRESS event.

Out of Order Message Solutions

- Store the request-sequence-number as an attribute in the request message that will be returned by the server as part of the response message.
- Maintain a table mapping an attribute in the request/response to the request-sequence-number.
- Implement an independent response routing system.

If the pending request table is not usable for the use case, the script author will need to disable automatic response routing by setting the 'no-response' attribute of the message via the "GENERICMESSAGE::message no-response true" command. To disable the table for all messages the no-response attribute in the protocol profile should be set to 'yes'.

Response messages may be routed by setting the nexthop or route attribute of the message.

Notification Messages

Notification messages are messages that do not expect a response. Setting the no-response attribute of the message will instruct generic message that no response is expected for the message. If all messages will be notification message, the no-response attribute in the protocol profile should be set to 'yes'.

Message Meta-Data Fields

Source Address

The source address is a string used to identify the originating connection of the message. This is auto-populated with the peer name of the connection.

Destination Address

The destination address is a string used to identify the destination for the message. This will be used by the route table along with the source address to determine the best route to forward the message towards.

lasthop

The last hop contains an internal address for the originating connection for the message. For response routing, setting the next hop of the response message to the last hop of the request will instruct the router to bypass routing and forward the message directly to the specified connection.

nexthop

The next hop specifies an internal address for a connection that should be used for forwarding the message.

route

Setting the route attribute of a message on ingress will bypass route table lookup and direct the router to forward the message directly to the route specified in the message.

After routing has completed, the route attribute of the message will be populated with the details of the route used.

attempted_hosts

Setting the attempted hosts attribute of a message on ingress will instruct the router to skip the provided hosts when performing a load balance pick.

After routing has completed, the attempted hosts attribute of the message will contain the list of transports (virtual or transport-config) and hosts (tuple) that the message has been routed towards.

originator

The originator attribute will contain the transport (virtual or transport-config) and host (tuple) of the originating connection of the message.

route_status

After routing has completed, the route status attribute will contain the status (success or failure) of the routing operation.

Profiles

Profiles are used to configure a message routing instance and its incoming connection (via virtual servers) and outgoing connections (via transprot-configs).

Router Profile

A router profile configures the message router instance. All virtuals sharing the same router profile will share the same message router instance. They will be able to route message between the connections of all virtual in the same router instance. They will also share the same route table. The name of the router profile will be the name of the router instance.

The router contains a list of routes. The routes will be loaded as static routes into the route table.

```
ltm message-routing generic router my_gm_router {
    routes { default_route, forwarding_route, bad_route }
}
```

Mirroring is also enabled via the router profile. All virtuals that use a router profile with mirroring enabled will be mirrored.

Generic Message Protocol Profile

The protocol profile configures the generic message parser.

```
ltm message-routing generic protocol <name> {
   no-response <yes/no>
   disable-parser <yes/no>
   message-terminator <url encoded string>
   max-message-size <integer>
   max-egress-buffer <integer>
}
```

Attributes

Table 7: Generic Message Protocol Profile Attributes

Attribute	Description	Default
no-response	When set, all messages do not expect a response. A pending-request entry will not be created to route response/answer messages back to the originator. This field is a boolean. Possible values are 'yes' and 'no'.	no
disable-parser	When set, the generic message protocol's parser ignores all input from the transport stream. It also will not forward messages to the transport stream. It is assumed that all message parsing and forwarding will be implemented in a tcl script.	no
message-terminator	The string of characters used to terminate a message.	none
max-message-size	The maximum size of a message in bytes. If a message is exceeds this size, the connection will be reset.	32768
max-egress-buffer	The maximum number of bytes waiting to be sent via the selected transport before the protocol filter notifies the router to stop delivering messages. Once the number of bytes waiting drops below this size, the protocol filter will notify the router that it may resume delivering messages.	32768

Generic Message Events

Table 8: Generic Message Protocol Profile Attributes

Event	Description
GENERICMESSAGE_INGRESS	This event is raised when a message is received by the generic message filter.
GENERICMESSAGE_EGRESS	The event is raised when a message is received from the proxy.

Generic Message Commands

Table 9: Generic Message Commands

Command	Description
GENERICMESSAGE::peer name	Gets the peer’s route name.
GENERICMESSAGE::peer name <name>	Sets the peer's route name.
GENERICMESSAGE::message src	Gets the message’s source address
GENERICMESSAGE::message src <src_addr>	Sets the message's source address
GENERICMESSAGE::message dst	Gets the message’s destination address
GENERICMESSAGE::message dst <dst_addr>	Sets the message’s destination address
GENERICMESSAGE::message is_request	Returns ‘true’ if the message is a request message
GENERICMESSAGE::message is_request true OR false	Identifies the message as a request
GENERICMESSAGE::message length	Gets the message size in bytes.
GENERICMESSAGE::message text	Gets the message text.
GENERICMESSAGE::message text <new_text>	Sets the message text.
GENERICMESSAGE::message data	Gets the message data (as an array of bytes).
GENERICMESSAGE::message data	Sets the message data (as an array of bytes).
GENERICMESSAGE::message no-response	Returns 'true' if the message is an asynchronous message.
GENERICMESSAGE::message no_response true OR false	Identifies the message as an asynchronous message.
GENERICMESSAGE::message create [<text> [<destination address>]]	Creates a new empty message that can be sent. Once created a TEXTMSG_INGRESS event shall be raised where the script writer will be able to populate the message. Upon completion of the script, the message will be forwarded for routing.
GENERICMESSAGE::message drop <reason>	Terminates processing of the current message.
GENERICMESSAGE::message status	Returns the route status. Please see MR::message status
GENERICMESSAGE::message request_sequence_number	Gets the request_sequence_number used to save the last hop of a request message. This will be valid during GENERICMESSAGE_EGRESS of request messages. During GENERICMESSAGE_INGRESS of response messages, it will contain the sequence number of the oldest un-responded request.
GENERICMESSAGE::message request_sequence_number <number>	Sets the request_sequence_number used to save the last hop of a request message. During GENERICMESSAGE_INGRESS, setting this field will

Command	Description
	allow associting a response message with an un-responded request message and the request message's lasthop will be used as the next hop for the message.
GENERICMESSAGE::route add [src <src_addr>] [dst <dst_addr>] peer <peer> [peer <peer>]*	Adds a route to the current instance that the iRule context is running in. (Note both ‘src’ and ‘source’ are acceptable. Likewise ‘dst’, ‘dest’ and ‘destination’ are acceptable).
GENERICMESSAGE::route delete [src <src_addr>] [dst <dst_addr>]	Deletes a route from the current instance that the iRule context is running in. (Note both ‘src’ and ‘source’ are acceptable. Likewise ‘dst’, ‘dest’ and ‘destination’ are acceptable).
GENERICMESSAGE::route lookup [src <src_addr>] [dst <dst_addr>]	Returns a route matching the specified source address and destination address in the instance that the iRule context is running in. (Note both ‘src’ and ‘source’ are acceptable. Likewise ‘dst’, ‘dest’ and ‘destination’ are acceptable).

Virtual Server Statistics

Table 10: Generic Message Commands

Field	Description
request messages in	The number of request messages received
response message in	The number of response messages received
request messages out	The number of request messages sent
response messages out	The number of response messages sent
request bytes in	The number of bytes of request messages received
response bytes in	The number of bytes of response message received
request bytes out	The number of bytes of request messages sent
response bytes out	The number of bytes of response messages sent
failed request messages	The number of request messages that failed routing
failed response messages	The number of response messages that failed routing
failed request bytes	The number of bytes of request messages that failed routing
failed response bytes	The number of bytes of response messages that failed routing
current connections	The current number of connections.
pending requests	The current number of pending requests

How to Use Generic Message - iRules

How to Use Generic Message - iRules

Generic message is a framework to allow implementation of new protocols via iRule scripts. Most of the work done will be done via iRule scripting.

Overview

An iRule is a powerful and flexible feature within the BIG-IP® local traffic management system that you can use to manage your network traffic. It allows operators to implement custom behavior beyond the native capabilities of the BIG IP system.

The Generic Message Protocol filter can be configured to divide the input stream (from the transport filter) into messages when it finds an instance of its message terminating string ("\n"). Each message will initially be treated as a response message. If a destination-address is added to the message, it will be changed to a request message.

The first message received after a new connection will be used as the peer name for the connection. A dynamic route for the peer will be added to the peer table using the peer name. The peer name may be set using the GENERICMESSAGE::peer name <name> command. If the peer name has already been set, the first message will be routed as any other message.

The Generic Message Protocol filter maintains a list of pending request messages. When a request message egresses through the filter, the last hop of the originator of the request is added to the pending request list (unless the message has been flagged as a no-response message).

Any message containing a destination address is treated as a request message. If a message does not contain a destination address and a pending request exists, that message is processed as the answer to the pending request. The message's nexthop attribute will be set to the lasthop of the oldest pending message and the pending message will be removed from the pending message list, and forwarded. An iRule command, GENERICMESSAGE::message request_sequence_number, can be used to associate a response to any pending request.

If a message does not contain a destination address and a pending request does not exist, that message will be processed as a request and forwarded for routing.

New Connections

When a new connection is created, the iRule will need to set the peer name for the connection.

Setting Peer Name

Each message will automatically have the source-address field populated with this peer name. A dynamic route will be entered for this connection with the assigned peer name used as the destination-address of the route. If a route already exists with the peer name, the earlier route will take precedence.

Peer Name Options

Junk Data

This will meet the requirement that a peer name be set and will keep multiple dynamic route entries from being created.

```
rule gm_rules {
    when CLIENT_ACCEPTED {
        GENERICMESSAGE::peer name "."
```

```
        TCP::collect
    }

    when SERVER_CONNECTED {
        GENERICMESSAGE::peer name "."
        TCP::collect
    }
```

Connection Based Value

```
rule gm_rules {
    when CLIENT_ACCEPTED {
        GENERICMESSAGE::peer name [IP::remote_addr]
        TCP::collect
    }

    when SERVER_CONNECTED {
        GENERICMESSAGE::peer name [IP::remote_addr]
        TCP::collect
    }
```

Negotiated Value from Peer

```
tm rule gm_rule {
    when CLIENT_ACCEPTED {
        TCP::respond "What is your name\n"
        TCP::collect
        set capture_name 1
    }

    when SERVER_CONNECTED {
        TCP::respond "What do you wish to be called\n"
        TCP::collect
        set capture_name 1
    }

    when CLIENT_DATA {
        set lines [split [TCP::payload] "\n"]
        TCP::payload 0 0
        foreach line $lines {
            set line [string trim $line]
            if { [string length $line] > 0 } {
                if { $capture_name == 1 } {
                    GENERICMESSAGE::peer name $line
                    set capture_name 0
                    TCP::respond "Welcome [GENERICMESSAGE::peer name]\n"
                } else {
...
                }
            }
        }
        TCP::release
        TCP::collect
    }

    when SERVER_DATA {
        set lines [split [TCP::payload] "\n"]
        TCP::payload 0 0
        foreach line $lines {
            set line [string trim $line]
            if { [string length $line] > 0 } {
                if { $capture_name == 1 } {
                    GENERICMESSAGE::peer name $line
                    set capture_name 0
                    TCP::respond "Welcome [GENERICMESSAGE::peer name]\n"
                } else {
...
                }
            }
```

```
        }
        TCP::release
        TCP::collect
    }
```

Message Creation

The data stream will need to be parsed during CLIENT_DATA and SERVER_DATA iRule events for messages. Once the bytes of a message has been collected, a message object will need to be created and populated.

The GENERICMESSAGE::message create command is used to create and populate a message.

Once a message is created, the GENERICMESSAGE_INGRESS event will be raised and the script author may modify and the message.

Upon completion of the GENERMESSAGE_INGRESS event, the message will be forwarded to the router for routing.

An example script that parses the input stream to create message follows. This example identifies a request message as having the destination address specified at the start of the message separated from the data of the message by a ':'.

```
when CLIENT_DATA {
        set lines [split [TCP::payload] "\n"]
        TCP::payload 0 0
        foreach line $lines {
            set line [string trim $line]
            if { [string length $line] > 0 } {
                if { $capture_name == 1 } {
...
                } else {
                    set tokens [split $line ":"]
                    if {[llength $tokens] > 1} {
                        GENERICMESSAGE::message create [join [lrange $tokens 1 end] ":"]
[lindex $tokens 0]
                    } else {
                        GENERICMESSAGE::message create $line
                    }
                }
            }
        }
        TCP::release
        TCP::collect
    }

    when SERVER_DATA {
        set lines [split [TCP::payload] "\n"]
        TCP::payload 0 0
        foreach line $lines {
            set line [string trim $line]
            if { [string length $line] > 0 } {
                if { $capture_name == 1 } {
...
                } else {
                    set tokens [split $line ":"]
                    if {[llength $tokens] > 1} {
                        GENERICMESSAGE::message create [join [lrange $tokens 1 end] ":"]
[lindex $tokens 0]
                    } else {
                        GENERICMESSAGE::message create $line
                    }
                }
            }
        }
        TCP::release
```

```
        TCP::collect
    }
```

Message Routing

When a message is received for routing, MRF will raise the MR_INGRESS event. The script author may set the nexthop or route attribute of the message to bypass the normal route table lookup. Response messages may already have the nexthop attribute set but the protocol if a pending request existed in the table.

Upon completion of the MR_INGRESS event, fi the message's nexthop attribute is set, the message will be forwarded to the connection specified in the nexthop attribute.

If the message's route attribute is set, route lookup will be skipped and the route value specified in the message's route attribute will be use to determine the distination host of for the message.

If the message's route attribute is not set, the route lookup will be performed using the message's source and destination address. The message's route attribute will be populated with the selected route's value.

After route selection, a peer from the route value will be selected and a pool member will be selected from the selected peer.

If available connection exists to the selected pool member, the message will be forwarded using that connection.

If an available connection does not exist, a new connection will be created.

The MR_EGRESS event will be raised as the message is leaving the router to be forwarded to the destination.

If a route could not be found or a connection could not be created, a MR_FAILED event will be raised. The script author may attempt to retry routing using the MR::retry command.

Message Delivery

When the outgoing message is received by the protocol a GENERICMESSAGE_EGRESS event will be raised. If the protocol's parser is disabled, the script author will need to output the data of the message in the script.

```
when GENERICMESSAGE_EGRESS {
    TCP::respond "[GENERICMESSAGE::message data]\n"
}
```

MRF iRule Events and Commands

MRF Events

Table 11: MRF events

Event	Description
MR_INGRESS	This event is raised when a message is received by the message proxy and before a route lookup occurs. Setting the route for a message will bypass route lookup.
MR_EGRESS	This event is raised after the route has been selected and processed and the message is delivered to the mr_proxy for forwarding on the new connflow.
MR_FAILED	This event is raised when a message has been returned to the originating flow due to a routing failure.

MRF Commands

Table 12: MRF Commands

Command	Description
MR::instance	Returns the name of the current mr_router instance. The instance name will be the same name as the router profile.
MR::protocol	Returns ‘generic, ‘sip’ or ‘diameter’
MR::store <name> …	Stores a tcl variable with the mr_message object. This variable will be delivered with the message to the egress connflow. Adding variables does not effect the content of the message
MR::restore [<name> …]	Returns adds the stored variables to the current context tcl variable store. If no name is provided, it will add all stored variables.

MR::peer <name>	Returns the content of the named peer. If a local peer has been created with the provided name (using MR::peer <name> ...), the local peer's contents will be returned. If a local peer has not been created with the provided name, the static peer from configuration will be returned. The returned value will be formatted as:
	(versions 11.5 - 12.1)
	<destination> using <transport>
	where:
	destination = <destination_type> "<destination_value>"
	destination_type = pool | virtual
	transport = <transport_type> "<transport_name>"
	transport_type = virtual | config
	for example:
	`pool "/Common/default_pool" using config "/Common/sip_udp_tc"`
	(version 13.0 +)
	<transport> <destination>
	where:
	destination = <destination_type> <destination_value>
	destination_type = pool | virtual
	transport = <transport_type> <transport_name>
	transport_type = virtual | config
	for example:
	`virtual /Common/sip_tcp_vs host [10.2.3.4]%0:5060`

MR::peer <name> [[virtual <virtual_name>] OR [config <transport_config_name>]] [[host <host tuple>] OR [pool <pool name>]]	Defines a peer to use for routing a message to. The peer may either refer to a named pool or a tuple (IP address, port and route domain iD). When creating a connection to a peer, the parameters of either a virtual server or a transport config object will be used. The peer object will only exist in the current connections connflow. When

Command	Description
	adding a route (via MR::route add), it will first look for a locally created peer object then for a peer object from the configuration. Once the current connection closes, the local peer object will go away.
MR::peer <name> [[virtual <virtual_name>] OR [config <transport_config_name>]] [[host <host tuple>] OR [pool <pool name>]] ratio <ratio_value>	Defines a peer to use for routing a message to. The peer may either refer to a named pool or a tuple (IP address, port and route domain iD). When creating a connection to a peer, the parameters of either a virtual server or a transport config object will be used. The peer object will only exist in the current connections connflow. When adding a route (via MR::route add), it will first look for a locally created peer object then for a peer object from the configuration. Once the current connection closes, the local peer object will go away. Adding the ratio keyword allows setting the ratio of the peer.
MR::message lasthop	Returns the message's lasthop (details of the connection that originated the message). The lasthop is presented as <TMM number>:<FlowID> for example `0:800000000005`
MR::message nexthop	Returns the message's nexthop (details of the connection the message is to be forwarded to). If the new_nexthop parameter is present, a nexthop may be set for the message. The nexthop is formated as <TMM number>:<FlowID> for example `0:800000000029`
MR::message nexthop <new_nexthop>	Sets the message's nexthop (details of the connection the message is to be forwarded to). The new_nexthop parameter is present, a nexthop may be set for the message. The nexthop is formated as <TMM number>:<FlowID>
MR::message route	Returns a rendering of the mr_route_value selected for this message. The returned value will be formatted as: (versions 11.5 - 12.1) { <destination> using <transport> [<destination> using <transport>] } where: destination = <destination_type> "<destination_value>" destination_type = pool | virtual transport = <transport_type> "<transport_name>" transport_type = virtual | config for example: `{ pool "/Common/default_pool" using config "/Common/sip_udp_tc" host "[10.2.3.4]%0:5060" using virtual "/Common/sip_tcp_vs" }` (version 13.0 +) <transport> <destination> [<transport> <destination>]

Command	Description
	where: destination = <destination_type> <destination_value> destination_type = pool | host transport = <transport_type> <transport_name> transport_type = virtual | config for example: `virtual /Common/sip_tcp_vs host [10.2.3.4]%0:5060 config /Common/sip_udp_tc pool /Common/default_pool`
MR::message route peer <peer_name> [peer <peer_name>]	Instructs the route table to route the message to the provided peer list. This form of the MR::message route command takes the names of configured peers or dynamic peers created via the MR::peer command.
MR::message route mode <sequential | ratio> peer <peer_name> [peer <peer_name>]	Instructs the route table to route the message to the provided peer list. The peer list will have the peer-selection-mode set the the provided mode. This form of the MR::message route command takes the names of configured peers or dynamic peers created via the MR::peer command.
MR::message route [[virtual <virtual_name>] OR [config <config_name>]] [[host <host tuple>] OR [pool <pool_name>]]	Instructs the route table to route the message to the provided host or pool.
MR::message attempted	Returns a list of hosts that the message has been routed towards. The returned value will be formatted as: <transport> <destination> [<transport> <destination>] where: destination = <destination_type> host <host_value> transport = <transport_type> <transport_name> transport_type = virtual | config for example: `virtual /Common/sip_tcp_vs host [10.2.3.4]%0:5060 config /Common/sip_udp_tc host [20.3.4.5]%0:5060`
MR::message attempted none	Clear list of attempted hosts from the message.
MR::message attempted [[virtual <virtual_name>] OR [config <config_name>]] [host <host tuple>]	Sets the list of attempted hosts in the message. If set before routing (during MR_INGRESS or MR_FAILED), the hosts in the attempted hosts list will be avoided when performing a lb_pick.
MR::message originator	Returns the transport type, transport name and ip address/port/route domain ID of the originator of the message. The returned value will be formatted as: <transport> <destination>

Command	Description
	where: destination = host <host_value> transport = <transport_type> <transport_name> transport_type = virtual | config for example: `virtual /Common/sip_tcp_vs host [10.2.3.4]%0:5060`
MR::message drop <reason>	Drops the current message.
MR::message retry_count	Returns the number of attempts to route this message that have occurred.
MR::message status	Returns the status of the routing operation (valid only at MR_EGRESS). Possible values are: "unprocessed", "route found", "no route found", "dropped", "queue_full", "no connection", "connection closing", "internal error", "persist key in use", and "standby dropped"
MR::flow_id	Returns the flow_id of the current connection (in hex).
MR::transport	Returns the transport type and name of the current connection. for example `config /Common/sip_udp_tc`
MR::prime [config <config_name>] OR [virtual <virtual_name>] [host <host tuple>] OR [pool <pool name>]	Initialize a connection to the specified peer (or active poolmembers of the specified pool) using the specified transport.
MR::retry	This command is only available during MR_FAILED event. It re-submits the current message for routing to an alternate pool member. If the previous routing attempt set the message's nexthop or route, these fields should be cleared before retrying routing (use "MR::message nexthop none" and "MR::message route none"). The message's route_status will automatically be reset by this command. If the the retry also fails and the retry_count has reached the max_retries setting in the router profile, the message will be given a "Max retries exceeded" route status.
MR::max_retries	Returns the configured max_retries of the router instance.
MR::connection_instance	Returns the instance number and number of connections of the current connection within a peer. It will be formatted as "<instance_number> of <max_connections>". For incoming connections, this will return "0 of 1". for example `0 of 5`
MR::connection_mode	Returns the connection_mode of the current connection as configured in the peer object. Valid connection_modes are "per-peer, per-blade, per-tmm and per-client". For incoming connections, this will be "per-peer".

Route Status

Table 13: Route Status

Status	Description
unprocessed	Message has not been submitted for routing yet
route found	Route has been found
no route found	A route has not been found
dropped	The message has been dropped by a MR::message drop command
queue full	The message was returned back to the originator because one of the MRF processing queues had reached its configured limit.
no connection	The message was routed to a connection which was no longer present.
connection closing	The message was queued to be send on a connection which was closed.
internal error	The message was unable to be delivered due to an internal error. For example, out of memory.
persist key in use	Two messages routed using the same persistence key simultanously tried to create the same persistence record.
standby dropped	The message is a mirrored message running on a standby device and was dropped as part of routing to avoid creating an outgoing connection on the standby device.
Max retries exceeded	The message was returned to the originator because the latest attempt to retry routing exceeded the configured max retry count.

Generic Message Example

Generic Message Example

This section provides an example of using Generic Message to implement a simple chat client.

Each new connection will be asked for the peer's name. The provided name will be used to add a dynamic route to the route table.

Subsequent text entered will be treated as a request or response message. Requests are of the form: "<destination>:<text>\n". Responses do not contain a destination. A response message will be routed to the originator of the oldest request received.

BIG-IP Configuration

```
ltm pool gm_pool {
    members {
        10.1.1.100:1234 {
            address 10.1.1.100
        }
    }
}

ltm message-routing generic peer default_peer {
    pool gm_pool
    transport-config my_tc
}
ltm message-routing generic protocol my_gm_proto {
    app-service none
    disable-parser yes
}
ltm message-routing generic route default_route {
    peers {
        default_peer
    }
}
ltm message-routing generic router my_gm_router {
    app-service none
    routes {
        default_route
    }
}
ltm message-routing generic transport-config my_tc {
    ip-protocol tcp
    profiles {
        my_gm_proto { }
        tcp { }
    }
    rules {
        gm_rule
    }
}

ltm virtual gm_vs {
    destination 10.1.1.50:1234
    ip-protocol tcp
    mask 255.255.255.255
    profiles {
        my_gm_proto { }
        my_gm_router { }
        tcp { }
    }
```

```
    rules {
        gm_rule
    }
    source 0.0.0.0/0
    vs-index 2
}

ltm rule gm_rule {
    when CLIENT_ACCEPTED {
        TCP::respond "What is your name\n"
        TCP::collect
        set capture_name 1
    }

    when SERVER_CONNECTED {
        TCP::respond "What do you wish to be called\n"
        TCP::collect
        set capture_name 1
    }

    when CLIENT_DATA {
        set lines [split [TCP::payload] "\n"]
        TCP::payload 0 0
        foreach line $lines {
            set line [string trim $line]
            if { [string length $line] > 0 } {
                if { $capture_name == 1 } {
                    GENERICMESSAGE::peer name $line
                    set capture_name 0
                    TCP::respond "Welcome [GENERICMESSAGE::peer name]\n"
                } else {
                    set tokens [split $line ":"]
                    if {[llength $tokens] > 1} {
                        GENERICMESSAGE::message create [join [lrange $tokens 1 end] ":"]
[lindex $tokens 0]
                    } else {
                        GENERICMESSAGE::message create $line
                    }
                }
            }
        }
        TCP::release
        TCP::collect
    }

    when SERVER_DATA {
        set lines [split [TCP::payload] "\n"]
        TCP::payload 0 0
        foreach line $lines {
            set line [string trim $line]
            if { [string length $line] > 0 } {
                if { $capture_name == 1 } {
                    GENERICMESSAGE::peer name $line
                    set capture_name 0
                    TCP::respond "Welcome [GENERICMESSAGE::peer name]\n"
                } else {
                    set tokens [split $line ":"]
                    if {[llength $tokens] > 1} {
                        GENERICMESSAGE::message create [join [lrange $tokens 1 end] ":"]
[lindex $tokens 0]
                    } else {
                        GENERICMESSAGE::message create $line
                    }
                }
            }
        }
        TCP::release
        TCP::collect
    }

    when GENERICMESSAGE_INGRESS {
```

```
#        TCP::respond "GM_INGRESS is_request [GENERICMESSAGE::message is_request]
req_seq_num [GENERICMESSAGE::message request_sequence_number]\n"
    }

    when MR_INGRESS {
#        TCP::respond "MR_INGRESS src: [GENERICMESSAGE::message src] dest
[GENERICMESSAGE::message dest] lasthop [MR::message lasthop] nexthop: [MR::message nexthop]
\n"
    }


    when GENERICMESSAGE_EGRESS {
#        TCP::respond "GM_EGRESS status: [GENERICMESSAGE::message status] is_request
[GENERICMESSAGE::message is_request] req_seq_num [GENERICMESSAGE::message
request_sequence_number]\n"
        TCP::respond "[GENERICMESSAGE::message data]\n"
    }
}
```

Client Server Configuration

Server

Execute the following command (with local IP address):

```
nc -l 10.1.1.100 1234
```

Client

Execute the following command (with local IP address):

```
telnet 10.1.1.50 1234
```

Communication Dialogue

The following example illustrates communication between different clients and a server. The example traverses down with time. Text entered is in `courier`.

Table 14: Communication Dialog

Client A	Client B	Server	Notes
What is your name `Alex` Welcome Alice	What is your name `Bob` Welcome Bob		Both clients connect and provide their routing name
`Bob: Are you there`			Request sent to route Bob (Req 1)
	Are you there `Yes, who is this?`		Request displayed (Req 1) Response (there is no destination provided), so it is routed to the originator of the oldest request (Rsp 1)
Yes, who is this? `Bob: This is Alex`			Response delivered (Rsp 1) New request (Req 2)

Client A	Client B	Server	Notes
	This is Alice `Alex: hello Alex`		Request delivered (Req 2) New request (Req 3)
hello Alex			Request delivered (Req 3)
	`Charlie: Are you there?`		New request (Req 4), destination does not match any existing routes so the default route is selected.
		What do you wish to be called `Dave` Welcome Dave	New connection established to pool member of default route
		Are you there? `Yes, who is this?`	Request delivered (Req 4) Response to Req 4 (Rsp 4)
	Yes, who is this? `Bob`		Response delivered (Rsp 4) Since there is no destination, this is a response. This will be matched to the oldest pending request on this connection which is Req 2 (Rsp 2)
Bob			Response to Req 2 delivered (Rsp 2)

Disaggregation (DAG) Modes

BIG-IP® scalability relies on load balancing (parallelizing) the processing of incoming packets across a large number of TMMs (cores). This is accomplished with various disaggregation algorithms. This section provides a concise description of these algorithms. Note, selecting the wrong disaggregation mode can have a severe impact on performance.

DAG is configured per VLAN. Note, this means that the client and server sides of BIG-IP should be configured on different VLANs. So it's possible to configure different DAG modes for client and server connections. However, when a server responds to a client request, and a connection is already established, DAG is not used.

DAG Modes via TMSH

Table 15: DAG Modes via TMSH

DAG Mode	Configuration Object	TMSH Commands
Default-DAG	VLAN	`$ modify net vlan <vlan_name> cmp-hash default`
SP-DAG	VLAN	`$ modify net vlan <src_vlan_name> cmp-hash src-ip` `$ modify net vlan <dst_vlan_name> cmp-hash dst-ip`

RR-DAG	VLAN	`$ modify net vlan <vlan_name> dag-round-robin enabled`
		`$ modify sys db dag.roundrobin.udp.portlist value "5060"`
		`$ modify ltm profile udp <udp_profile_name> idle-timeout 0`

Default DAG

The Default DAG uses a hash of source and destination port. It is useful when ephemeral ports are used in client side and server side connections. When source and destination ports are the same TMM-0 will be used. This is an issue in that the traffic will not be load balanced and TMM-0 will quickly be overloaded. This DAG requires randomness in the source or destination port. If a client doesn't specify a source port then an ephemeral port will be used and Default DAG will work properly. Note, the ephemeral port must increment randomly or by single digits. If it's incremented by an even number, such as two, or by the number of TMMs then it's possible that it will hash to the same TMM or a small set of TMMs, which will negatively impact BIG-IP® performance.

Key Points

- Port Based.
- Works best when clients use ephemeral ports.
- Can work with 1 to n clients.

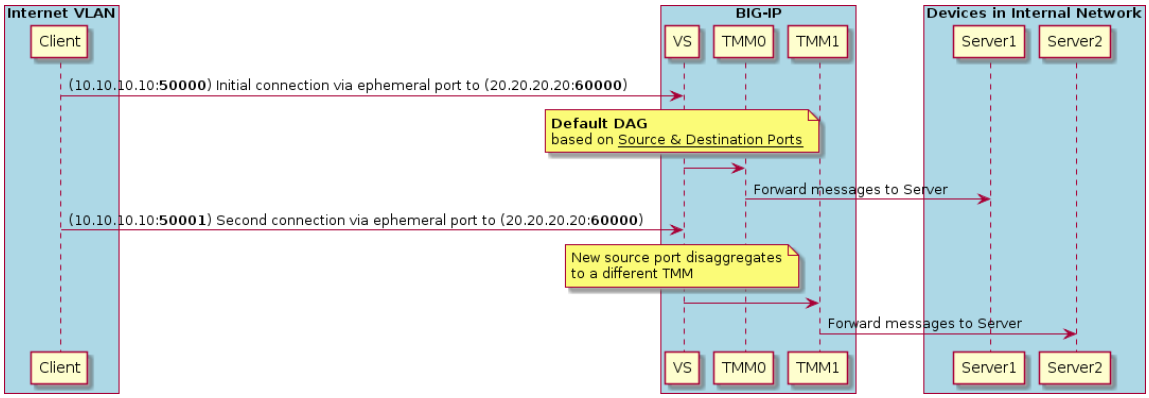

Figure 6: Default DAG example

Source/Destination DAG (SP-DAG)

The SP-DAG uses a hash of source IP (from client) and destination IP address (server). This mode should be used when source and destination ports are hardcoded (for example 5060). In that case, a BIG-IP® system requires multiple client IP address or multiple server IP addresses. Keep in mind, most connections are initiated by the client and that's the "Source DAG" option. In this case, the "Destination DAG" could be a single IP, but the source client IP should have more that a single IP address.

Key Points

- IP Address Based
- Works best when number of clients is equal to or more than the number of TMMs in BIG-IP system.
- Performance will be impacted if clients consist of only a few SIP Proxy connections. In this case the IP Address entropy will be too low to load balance the incoming packets across available TMMs.

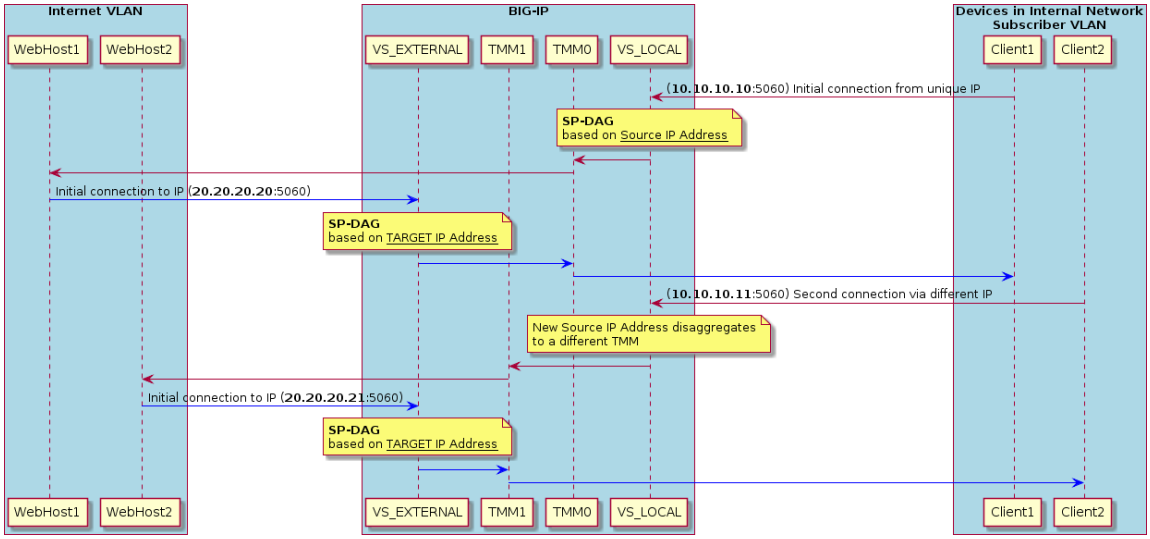

Figure 7: Source/Destination DAG (SP-DAG) example

Round Robin DAG (RR-DAG)

R-DAG was designed to overcome the low entropy limitations of Default DAG and SP-Dag; although for UDP only. Furthermore, RR-DAG is hardware only and can't be used in a VE. Round Robin DAG distributes traffic by sending each consecutive packet to a different TMM. It does not rely on the IP address, or source port, of the client. One limitation of RR-DAG is that it's global and can be configured for only one port.

SIP Specific Requirements

- Immediate timeout must be set on the UDP Profile

Key Points

- UDP Only
- Requires hardware (not an option in VE)
- Sends each consecutive packet to a different TMM.

```
$ modify net vlan <vlan_name> dag-round-robin enabled

$ modify sys db dag.roundrobin.udp.portlist value "5060"
```

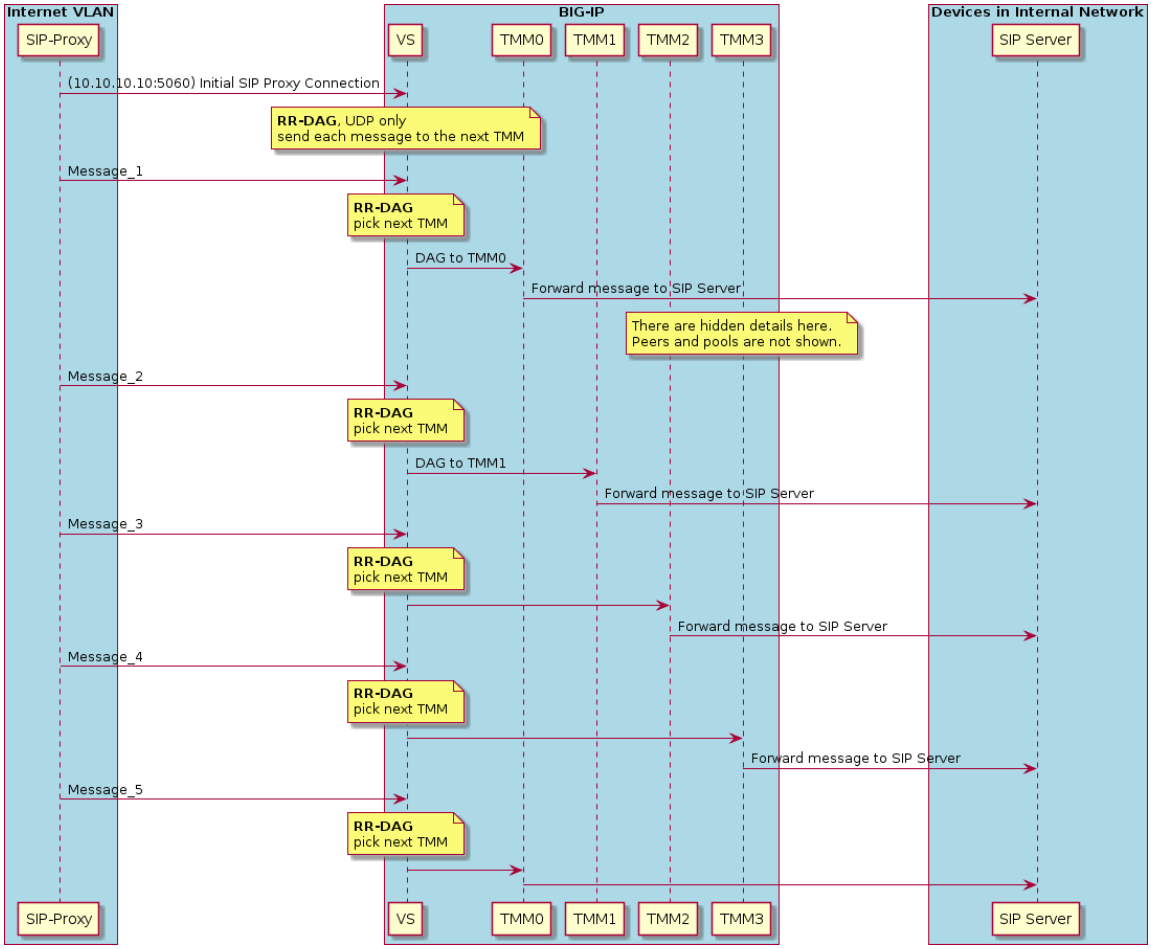

Figure 8: Round Robin DAG (RR-DAG) example

Disaggregation (DAG) Modes

Disaggregation (DAG) Modes

BIG-IP® scalability relies on load balancing (parallelizing) the processing of incoming packets across a large number of TMMs (cores). This is accomplished with various disaggregation algorithms. This section provides a concise description of these algorithms. Note, selecting the wrong disaggregation mode can have a severe impact on performance.

DAG is configured per VLAN. Note, this means that the client and server sides of BIG-IP should be configured on different VLANs. So it's possible to configure different DAG modes for client and server connections. However, when a server responds to a client request, and a connection is already established, DAG is not used.

DAG Modes via TMSH

Table 16: DAG Modes via TMSH

DAG Mode	Configuration Object	TMSH Commands
Default-DAG	VLAN	`$ modify net vlan <vlan_name> cmp-hash default`
SP-DAG	VLAN	`$ modify net vlan <src_vlan_name> cmp-hash src-ip` `$ modify net vlan <dst_vlan_name> cmp-hash dst-ip`

RR-DAG	VLAN	`$ modify net vlan <vlan_name> dag-round-robin enabled`
		`$ modify sys db dag.roundrobin.udp.portlist value "5060"`
		`$ modify ltm profile udp <udp_profile_name> idle-timeout 0`

Default DAG

The Default DAG uses a hash of source and destination port. It is useful when ephemeral ports are used in client side and server side connections. When source and destination ports are the same TMM-0 will be used. This is an issue in that the traffic will not be load balanced and TMM-0 will quickly be overloaded. This DAG requires randomness in the source or destination port. If a client doesn't specify a source port then an ephemeral port will be used and Default DAG will work properly. Note, the ephemeral port must increment randomly or by single digits. If it's incremented by an even number, such as two, or by the number of TMMs then it's possible that it will hash to the same TMM or a small set of TMMs, which will negatively impact BIG-IP® performance.

Key Points

- Port Based.
- Works best when clients use ephemeral ports.

- Can work with 1 to n clients.

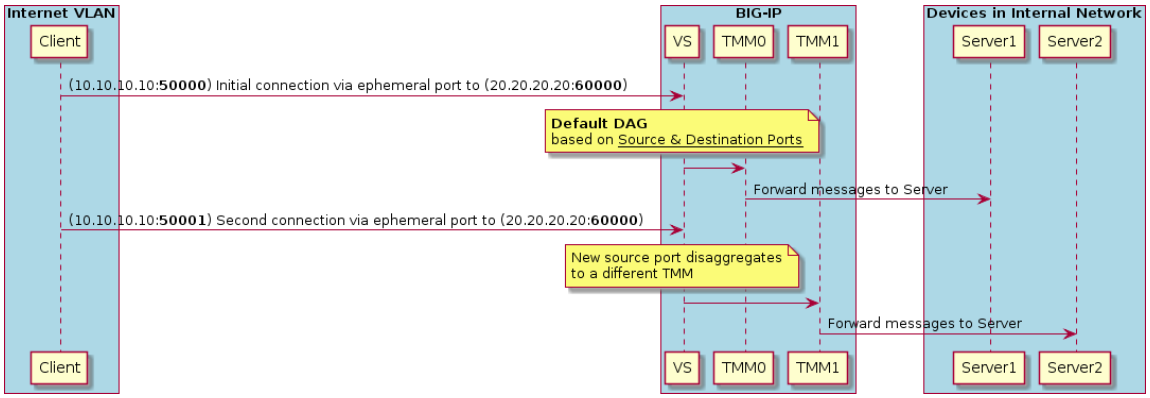

Figure 9: Default DAG example

Source/Destination DAG (SP-DAG)

The SP-DAG uses a hash of source IP (from client) and destination IP address (server). This mode should be used when source and destination ports are hardcoded (for example 5060). In that case, a BIG-IP® system requires multiple client IP address or multiple server IP addresses. Keep in mind, most connections are initiated by the client and that's the "Source DAG" option. In this case, the "Destination DAG" could be a single IP, but the source client IP should have more that a single IP address.

Key Points

- IP Address Based
- Works best when number of clients is equal to or more than the number of TMMs in BIG-IP system.
- Performance will be impacted if clients consist of only a few SIP Proxy connections. In this case the IP Address entropy will be too low to load balance the incoming packets across available TMMs.

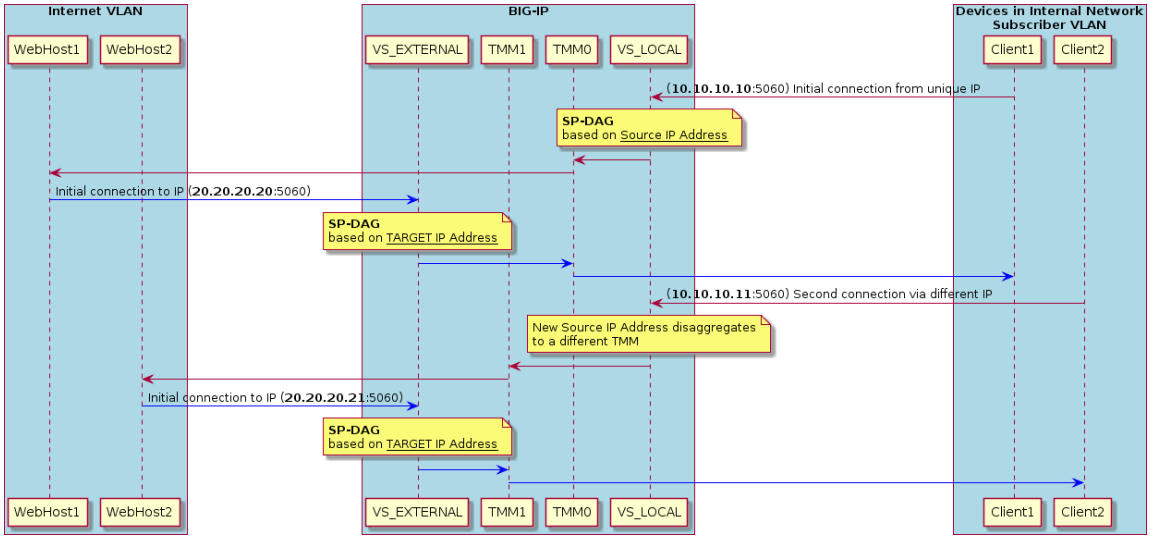

Figure 10: Source/Destination DAG (SP-DAG) example

Round Robin DAG (RR-DAG)

R-DAG was designed to overcome the low entropy limitations of Default DAG and SP-Dag; although for UDP only. Furthermore, RR-DAG is hardware only and can't be used in a VE. Round Robin DAG distributes traffic by sending each consecutive packet to a different TMM. It does not rely on the IP

address, or source port, of the client. One limitation of RR-DAG is that it's global and can be configured for only one port.

SIP Specific Requirements

- Immediate timeout must be set on the UDP Profile

Key Points

- UDP Only
- Requires hardware (not an option in VE)
- Sends each consecutive packet to a different TMM.

```
$ modify net vlan <vlan_name> dag-round-robin enabled
```

```
$ modify sys db dag.roundrobin.udp.portlist value "5060"
```

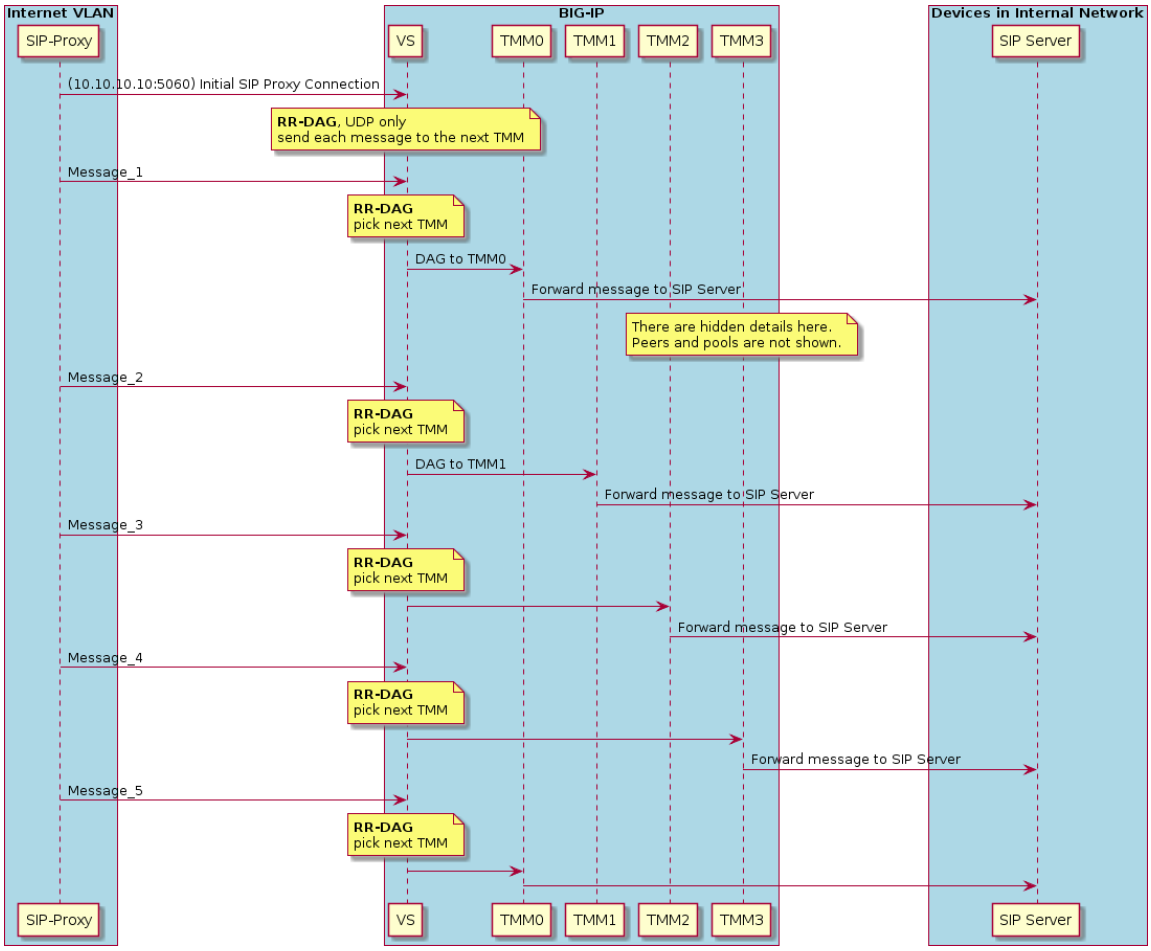

Figure 11: Round Robin DAG (RR-DAG) example

Troubleshooting

Troubleshooting

Can't find Generic Message Profile

The genericmessage profile is only available via tmsh.

Log Messages

Configuration Validation Errors

Table 17: Configuration Validation Errors

Configuration Failure Condition	Error Message
Deleting a route which is in use by a router profile	`The route <name> is referenced by one or more router profiles`
Peer refers to non-existent route	`Peer <name> refers to non-existing Static-Route <name>`
Route refers to non-existent peer.	`Static Route <name> refers to non-existing Peer <name>`

Common Configuration Problems

Moving router and/or virtual to different traffic group

The BIG-IP® system does not support changing the traffic group or a router and/or virtual server. MRF stores state that has a different lifetime than a connection in an internal in-memory database (known as session db). This includes persistence tables (SIP LB), call tables (SIP ALG), and registrations tables (SIP ALG), etc. Records stored in session db are auto replicated between the active and standby device. Part of the key for each entry in the session db is the identifier for a traffic-group. If the traffic-group of a virtual and/or router instance is changed all data stored in session db will be orphaned.

Config changes not loading, or stats don't show up on new router instance

Most changes to config are applied to existing connections. Changes to the set of profiles used by a connection only apply to new connections. Since many message routing protocols use long lived connections, some config changes will not effect existing connections. For example replacing the router profile used by a virtual server will not apply to existing connections. Thus all traffic on existing connections will still be routed through the previous router instance and the stats for that traffic will be included with the previous router instance. To apply the traffic to the new router instance, the existing connections will need to be closed forcing the clients to create new connections.

iRule changes not loading

Changes to iRule scripts attached to a virtual or transport-config do not change the scripts executing on existing connections. New connections will use the updated scripts. To cause the new script code to be applied, all existing connections (both client side and server side) will need to be closed and new

connections created. This may be avoided by moving the business logic of the script to a procedure as follows:

```
ltm rule mylib {
    proc sip_ingress {} {
        if { [SIP::is_request] and [clientside] } {
            # do something
            # change here
        } else {
            # do something else
            # change here
        }
    }
}
ltm rule routing {
    when SIP_INGRESS {
        call mylib::sip_ingress
    }
}
```

MRF Debugging

Did the message reach the message router?

There are multiple places where processing can stop or a failure can occur. The stats of the profiles added to the virtual server (or transport-config) should be used to determine if the message reached the message router. From the transport profile's stats (TCP/UDP/SCTP), it can be determined if packets were received by the transport filter. From the protocol profile's stats (sipsession), it can be determined if the received packets were correctly parsed into messages. If an error was found in the message parsing this should be detectable using the protocol's stats.

The message router profile (siprouter) stats should increment with each message received. The result of each messages routing operation should also be represented in the stats.

Why did the message fail routing

The MR_INGRESS event is raised for each message before it enters routing . Once routing is complete either MR_EGRESS or MR_FAILED event is raised. The message metadata can be logged during these events to help debug the results of routing. Some fields and their usage follows:

Metadata Field	Populated	Purpose
lasthop	before MR_INGRESS	Contains the TMM and flow_id of the originating connection of the message
nexthop	before MR_INGRESS (or during MR_INGRESS)	Selects the destination connection for the message
route	after routing (or during MR_INGRESS)	The value of the selected route (peer list). If set during MR_INGRESS, this route will be used instead of performing route lookup
originator	before MR_INGRESS	The IP, port and rtdom_id of the originator of the connection. Also contains the transport type and name of the originating connection.
status	after routing	The results of the route lookup
attempted	after routing (or during MR_INGRESS)	The list of destination hosts attempted. This list of hosts will be treated as marked down when performing peer selection and load balanced pick.

Metadata Field	Populated	Purpose
retry_count	after routing	The number of times this message has been submitted for routing

MR:route_status: "queue full"

One reason for MR_FAILED would be when MR:route_status is set to "queue full". This result can happen when the following conditions are met:

1. MRF-SIP profile with TCP for transport.
2. SIP Peer has very few pool members.
3. One of the pool member is down.
4. Burst of SIP Traffic with message size > 2K Bytes.

There are 2 configurable items (Max-Pending-Messages and Max-Pending-Bytes) in the router config to define the queue capacity. If the incoming traffic is high with large messages then the possibility of filling up the queue increases significantly before the connection request timeout occurs on the pool member which is down.

If the message size is larger than 2k then try increasing the Max-Pending-Bytes first. Otherwise, increase Max-Pending-Messages. If neither increase works, then increase both values.

MRF Logging

During MR_INGRESS, the message's route can be examined as follows:

```
Log local0. "route [MR::message route]"
```

The transport type and name can be inspected (in v12.0 and later) via an iRule command as follows:

```
Log local0. "transport [MR::transport]"
```

FAQ

FAQ

Licensing

Generic Message requires an LTM license.

Transport Translation

Transport translation is not supported. In other words, a UDP client connection cannot be sent to a TCP peer and vice versa.

Message Retry

When a message fails to route, it will be returned to the originating connflow and MR_FAILED event will be raised. An iRule script will be able to examine the message and resubmit it for routing via the MR::retry command.

There are multiple steps to routing, to understand how MR::retry will work, you will need to understand the steps. To avoid some of these steps or force a different path you may need to modify some of the metadata contained with the message.

Routing steps:

1. If persistence is enabled on the originating transport, the generated persistence key (via config or iRule) will be used to look for a persistence record. If a persistence record is found, the message will be forwarded to the host specified in the persistence record (step 7).
2. The protocol specific route table implementation will lookup the best route for the message based on a protocol specific attributes contained in the message.
3. The route found contains a peer list. A peer is selected from the peer list using the peer selection mode.
4. The selected peer may contain a pool and a transport. If a pool exists, it will select the first active pool member that has not already be attempted for this message. If no pool exists, it will forward the message to the local IP and port of the incoming connection.
5. Once a host has been selected, MRF will look to see if an available connection already exists to the host. If an available connection exists, the message will be egressed to the host via that connection. If an available connection does not exist, a new connection will be created and the message will be forwarded through the new connection

Connection Auto-Initialization

If a peer object has auto-initialization enabled, the BIG-IP® system will automatically create outbound connections to the active pool members in the specified pool using the configuration of the specified transport-config. For auto-initialization to attempt to create a connection, the peer must be included in a route that is attached to a router instance. For each router instance that the peer is contained in, a connection will be initiated. The auto-initialization logic will verify at a configurable interval if the a connection exists between the BIG-IP and the pool members of the pool. If a connection does not exist, it will attempt to reestablish one.

The first auto-intialization attempt will occur at least one auto-initialization-interval delay from when the object is loaded or changed in the TMM.

If the router instance is not included in any virtual servers, connection auto-initialization will not start. Once the router instance has been included in an enabled virtual server, auto-initialization will begin and will remain running for those peers used by routes attached to the router instance even if the router instance is removed from the virtual server.

If a peer with auto-initialization enabled, is used in multiple router instances, a separate connection will be established for each router instance.

The auto-initialization logic will only attempt to create connections to enabled pool members. If the pool member is marked down by an external monitor it will be ignored unless an inband monitor is also attached.

If mirroring is enabled on the router instance, the active device will initialize outgoing connections. The new outgoing connections will be mirrored to the standby device.

iRules on all transports

With MRF the outgoing connection may not use the same transport as the incoming connection. Incoming connections are defined via virtual servers. Outgoing connections are often defined with transport-configs. If the same iRule script is desired to run on all connections, the script should be defined for all transports.

For example tests assume a simple load balancing configuration with a virtual server (VS_IN) that is part of a router instance with a single default route. This default route contains a single peer that uses a transport-config (TC_OUT) to define the parameters of the outgoing connection. In this setup, a request message would be received on VS_IN. The request message would ingress on a hudchain configured via the settings of the virtual server. As the message was processed, the SIP_REQUEST and MR_INGRESS events would be raised on the iRule scripts attached to the virtual server. The request message would be forwarded to an outgoing connection configured via the setting of the transport-config. As the message egressed through the outgoing connection, the MR_EGRESS and SIP_REQUEST_SEND events would be raised on the iRule scripts attached to the transport-config. When the response message is received by the outgoing connection, the SIP_RESPONSE and MR_INGRESS events would be raised on the iRule script attached to the transport-config. The response will be forwarded to the connection that originated the request and the MR_EGRESS and SIP_RESPONSE_SEND events would be raised on the iRule script attached to the virtual server.

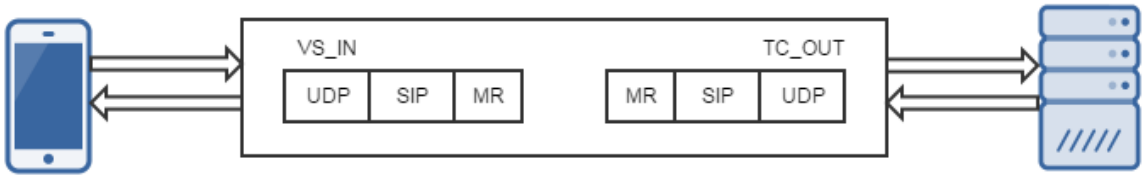

Figure 12: iRules on all transports

Sharing iRule variables between connections

MRF does not join the client side connection with the server side connection (except for SIP ALG). The traditional method of using the CLIENTSIDE or SERVERSIDE keywords to access variables will not work. Instead MRF provides a command to deliver tcl variables along side of the message to the outgoing connection. The MR::store command allows the script author to specify which tcl variables should be delivered to the outgoing connection. The MR::restore command unpacks the delivered variables on the outgoing connection and adds them to the connections context.

For example on the incoming connection:

```
when MR_INGRESS {
  set originator_ip [IP::remote_add]
  set ingress_message_count [expr $message_count + 1]
```

```
  MR::store originator_ip ingress_message_count
}
```

On the outgoing connection:

```
when MR_EGRESS {
  MR::restore
  log local0. "originator_ip $originator_ip ingress_message_count $ingress_message_count"
}
```

The effect of message pipelining on iRule variables

SIP can pipeline messages by allowing messages that require less processing to be forwarded without waiting for earlier messages that require more processing. For this reason, it is not recommended to store state in tcl variables to be used by subsequent iRule events. There is no guarantee that the next event raised after the protocol's message event will be the MR_INGRESS for the same message. For example, saving the SIP uri in a tcl variable during a SIP_REQUEST event to use for making a routing decision during MR_INGRESS is not recommended. The next MR_INGRESS event may not be for the same message as the last SIP_REQUEST event.

MRF SIP implementation allows accessing the SIP iRule commands during the MR events. This is the recommended method to make routing and delivery decisions based on attributes of a message.

For example:

```
when MR_INGRESS {
  if {[URI::host [SIP::uri]] equal "othersp.com"} {
    MR::message route config "/Common/othersp_tc" pool "/Common/othersp_pool"
  }
}
```

SNAT settings of the outgoing transport used

MRF uses the SNAT setting of the outgoing connection to determine how the source address is translated. Most outgoing connections are configured via a transport-config and the SNAT setting of the transport config will be used to select the source address. The only time the SNAT settings defined in the virtual server are used is if the setting of the virtual is used to create the outgoing connection (this occurs if no transport-config is set in the peer object).

Connection Reuse

MRF maintains a table of all existing connections on each TMM of a router instance. When a message is routed to a host, MRF will scan this table for an existing connection to the host that is available for use. If an available existing connection is not found, a new connection will be created.

There are many reasons that an existing connection may not be available for delivery of the current message (see the sub-sections below for details).

Transport

Each connection is created using the parameters of a transport object (either a virtual server or a transport-config). The transport specifies the profiles, SNAT and iRule scripts of the connection. When a message is routed, MRF will scan the list of connections for a connection created with the same transport specified. Even if the two transports contain the same parameters, a connection created with a different transport will not be used.

A pool object only allows specification of a transport-config as the outgoing connection transport. If the peer object does not specify the transport config, the transport of the message's originating connection will be used. If the system wishes to potentially deliver a message through an existing connection created

with by different virtual server on the same router, the MR::message route iRule command must be used. For example:

```
MR::message route virtual "/Common/internal_vs" host [IP::local_addr]:5060
```

Remote Port and ignore-clientside-port (or ignore-peer-port)

Many clients when creating connections use an ephemeral port for the local port. If a message is routed to that host, the port specified in the host's address will be different than the remote port of any existing connection with the host. Many MRF protocol implementation have an 'ignore_clientside_port' attribute in their router profile. Setting attribute to 'true' instructs MRF that any connection created by the host (client side) that matches the transport, remote IP and rtdom_id may be used.

Number-connections and instance number

The number-connections attribute of the peer object specifies which connection of a set of connections to a host will be used for delivering a message. It is used alongside the connection-mode instance to set the maximum number of connections between a router instance not the BIG-IP® system and a host.

use-local-connection

Many MRF protocol router profiles contain a 'use-local-connection' attribute. If this attribute is set, if a outgoing connection exists on the current TMM, it will be used even if the instance number does not match. Using this optimization will effectively limit the number of outgoing connections to one per TMM.

Source port

MRF allows setting the source port used on outgoing connections through the source-port attribute of a transport-config object. Setting this attribute to a non-zero value causes the source port of the outgoing connection to be set to the provided value. If set to zero an ephemeral port value will be used.

Pinning the source port to a fixed value will limit the number of connections available to the host. There can only be one connection using the local and remote tuples (IP/port/rtdom_id) and IP protocol (TCP/UDP/SCTP). Attempts to create another connection using the same addresses and IP protocol will fail.

For this reason it is not recommended to use set the source port for outgoing connections except when using a connection-mode of 'per-peer' and a number-connections of '1'.

Likewise trying to use the same host from peers with different transport settings (transport-config and/or virtual) and setting the source port will produce failures (unless different SNAT pools are used).

Legal Notices

Legal notices

Publication Date

This document was published on October 25, 2017.

Publication Number

MAN-0669-01

Copyright

Trademarks

For a current list of F5 trademarks and service marks, see *http://www.f5.com/about/guidelines-policies/trademarks*.

All other product and company names herein may be trademarks of their respective owners.

Patents

This product may be protected by one or more patents indicated at: *https://f5.com/about-us/policies/patents*.

Link Controller Availability

This product is not currently available in the U.S.

Export Regulation Notice

This product may include cryptographic software. Under the Export Administration Act, the United States government may consider it a criminal offense to export this product from the United States.

RF Interference Warning

This is a Class A product. In a domestic environment this product may cause radio interference, in which case the user may be required to take adequate measures.

FCC Compliance

This equipment has been tested and found to comply with the limits for a Class A digital device pursuant to Part 15 of FCC rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This unit generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a

residential area is likely to cause harmful interference, in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

Any modifications to this device, unless expressly approved by the manufacturer, can void the user's authority to operate this equipment under part 15 of the FCC rules.

Canadian Regulatory Compliance

This Class A digital apparatus complies with Canadian ICES-003.

Standards Compliance

This product conforms to the IEC, European Union, ANSI/UL and Canadian CSA standards applicable to Information Technology products at the time of manufacture.

Index

N

P

R

S

T