# What is the Message Routing Framework?

To understand MRF, how it works and its motivations, we must first refresh our understanding of the normal TMOS full-proxy architecture. Recall that, when configured for full-proxy operation, there are two sides to the proxy: client-side and server-side.

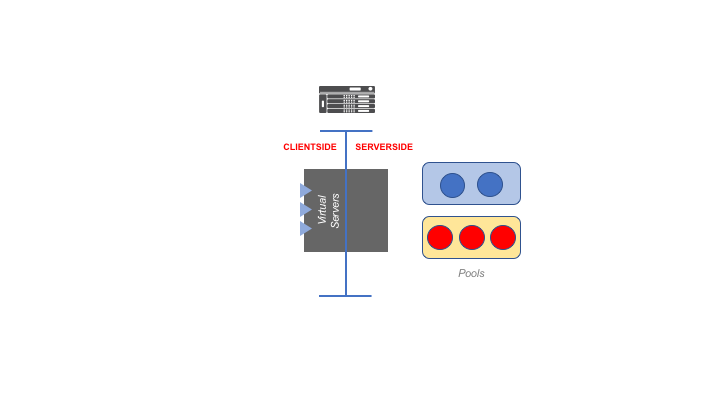


Figure 1 – BIG-IP Full-Proxy Operation

Even when a Virtual Server is configured for FastL4 – and even when things like Loose Initiation and Loose Close are employed – there is still a two-sided flow entry for each connection or flow.

Virtual Server listeners are bound to the client-side. When a flow is initiated from a remote system to the Virtual Server, a forwarding decision is made, and ordinarily, this triggers the creation of a corresponding server-side flow. The client-side and server-side transports usually match in type (e.g., TCP on both the client- and server-side), and there is a one-to-one relationship; that is, all L4 PDUs from the client-side are proxied (perhaps after modification) to the single matching server-side flow for the lifetime of the client-side flow. Similarly, all L4 PDUs that return from the server-side flow are proxied to the single matching client-side flow.

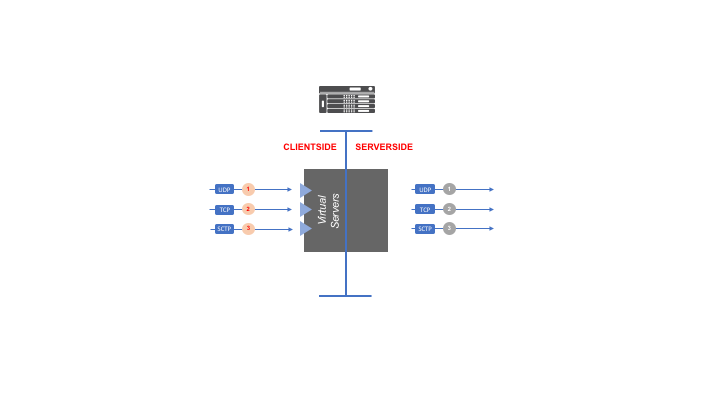


Figure 2 – BIG-IP Flow Proxying

This works well for:

* flow-oriented applications and protocols, like FTP or an RTP audio stream;
* message-oriented protocols where the protocol itself intends for message streams, including most notably HTTP (including when it employs HTTP Keep-Alive); and
* trivial message exchange protocols, like DNS, where there is usually only a single request/response pair in a transaction for each socket.

It does not work well, however, for message-based protocols where each message in a stream may profitably be routed independently of other messages in the stream. Even for HTTP, routing requests between different server-side streams can be useful. This is precisely what OneConnect does. Often, the message-based protocols that benefit from independent, per-message routing also support multiple transport layer protocols, so routing between different flows may also mean moving a message from one transport type (e.g., UDP) to another (e.g., TCP).

The list of protocols which may wish to utilize per-message routing include: HTTP (both 1.x and 2), Diameter, SIP, GPRS Tunneling Protocol (GTP), Short Message Peer-to-Peer Protocol (SMPP), RADIUS, MQTT, LDAP, DHCP, and many more. Some of these require the same transport type on both sides of the message router (for example, GTP requires UDP and SMPP requires TCP), but some allow various peers to use differing transport types (for example, SIP can use TCP or UDP, and Diameter can use TCP or SCTP). In this case, the message router may need to translate between layer4 transport types as messages are routed.

Enter the Message Routing Framework (MRF). MRF decouples the clientside and serverside for a BIG-IP flow[[1]](#footnote-1). Moreover, it performs an independent load-balancing decision (called a *routing decision* in MRF) for each incoming message. This invites the question: how does MRF know what a message is? MRF has four personalities: SIP, Diameter, MQTT[[2]](#footnote-2) and Generic[[3]](#footnote-3). Each personality has a built-in parser for extracting the matching L7 PDU from an incoming flow. For example, the SIP personality parser understands the structure of SIP messages and can extract them regardless of underlying transport (TCP or UDP) type.

SIP, Diameter and MQTT correspond to the L7 protocols with the same name. But what is Generic? This personality is a catch-all for any other type of message-based extraction and routing. It does have a built-in parser which will inspect ASCII text and break it into messages using a textual string. This would be useful for line-oriented protocols like the FTP control channel, where messages are delimited with the two-character sequence commonly represented as “\r\n”. However, when one is using the Generic personality, one usually disables the built-in parser, and instead uses an iRule to signal to MRF when a message is found. This is discussed in greater detail below.

# MRF Flow Decoupling

It is very important to understand that, when a Virtual Server uses MRF, the clientside and serverside flows become *decoupled.* As described above, in a normal proxy configuration, when a client opens a flow toward a Virtual Server, this ordinarily triggers the creation of a single corresponding serverside flow based on the result of the load-balancing or forwarding decision. For the entire lifetime of the clientside flow, all data on that clientside flow is copied to the serverside flow[[4]](#footnote-4), and all data incoming on the serverside flow is copied back to the corresponding clientside flow.

MRF introduces the notion of a *message router*. When a client connects to an MRF Virtual Server, that clientside flow is associated with the router object configured on the Virtual Server. Over time, as described below, the router may trigger the creation of one or more serverside flows. When data arrives on a flow – whether it is clientside or serverside – it is broken up into individual messages. Each message is routed according to the configuration, including any applied iRules. Importantly, the target of the route decision may be any flow associated with the router. Although clientside messages are ordinarily routed to serverside flows, and serverside messages are ordinarily routed to clientside flows, this is not required. For example, if five messages arrive on a clientside flow, each message may be routed to any other flow associated with the same router. Thus, the first message may be routed to a serverside flow, the second to a different serverside flow, the third to a clientside flow, and so forth.

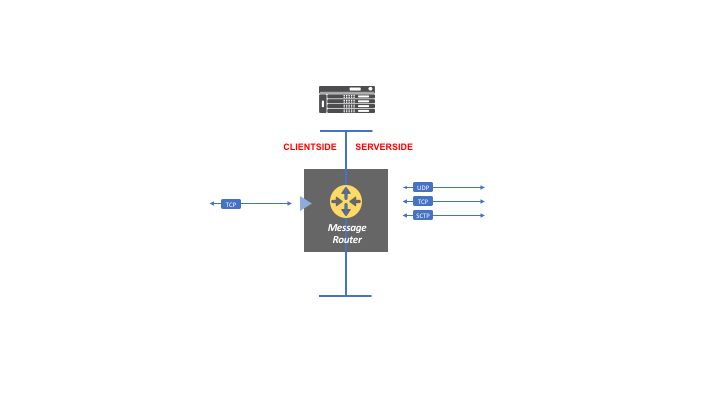


Figure 3 – Message Router and Flow Decoupling

Subtly – but importantly – with MRF, there is still the notion of a client-side flow and a server-side flow. A client-side MRF flow is a flow between an endpoint and a BIG-IP Virtual Server, while a server-side MRF flow is a flow between an endpoint and a BIG-IP self-IP or snatpool address.

# MRF Peers and Transports

MRF endpoints are called *peers*. This is the target of a route decision. Unfortunately, MRF configuration creates some ambiguity regarding this term. As we shall see below, there is a configuration object called a “Peer”, but this object refers to a BIG-IP Pool rather than a single endpoint. For clarity, I will use the term *peer* to unambiguously mean a single endpoint, the term *peer pool* to refer to a BIG-IP Pool referenced by a Peer object, and the term *Peer object* to refer to the configuration profile called a “Peer”.

A peer is identified by its IP address, port and by the transport through which it can be reached. When using normal proxy operation, the Virtual Server defines the transport characteristics for both the clientside and serverside. However, with MRF, since a message may be routed to peers with differing transport characteristics (and may even use different L4 transport types), each peer must be able to define its own transport characteristics.

Broadly speaking, there are two types of MRF peers: a *client peer* and a *server peer*. A client peer is any peer that initiates a flow toward the BIG-IP through a Virtual Server. A server peer is any peer to which the BIG-IP initiates a flow, typically as the result of a route decision. Since a client peer connects through a Virtual Server, its transport characteristics are defined by the Virtual Server definition. For example, if a client connects using TCP to a Virtual Server with the f5-tcp-lan profile, the TCP transport will use the values defined in the f5-tcp-lan profile. A server peer, on the other hand, usually uses a *Transport Config* object to define its characteristics. The Transport Config object defines the L4 transport type used by a server peer, its corresponding transport profiles (e.g., f5-tcp-lan or the udp profile), any iRules that apply to its flows, and a SNAT configuration, if SNAT is being used for flows toward the server peer.

Notice that I said static peers *usually* use a Transport Config object to define their transport characteristics. It is also possible for a static peer to inherit the characteristics of the Virtual Server used by the clientside peer that triggered the creation of the static peer’s transport[[5]](#footnote-5). In practice, there are a narrow set of reasons for doing this, and in general, it should be avoided unless needed. Otherwise, it is difficult to know for certain the transport characteristics of a static peer by simply looking at the config. Moreover, as we shall see, some configuration parameters of a Virtual Server have a different effect when applied to a clientside transport than when it is applied to a serverside transport. I also simplified somewhat the meaning of a clientside peer. Strictly speaking, the BIG-IP can launch an outbound connection toward a peer on the clientside of the proxy, but once again, this is a rare case with a specific use.

Since a peer may use either a Virtual Server (again, this is for clientside peers) or a Transport Config (again, this is for serverside peers) to define its transport characteristics, I will use the term *transport definition* to mean either a Virtual Server or Transport Config object when either could be applicable.

A Peer object is a configuration element that identifies a target peer pool (that is, a BIG-IP pool), a Transport Config, a *connection mode*, and a few other characteristics that will be discussed later. A peer that is known to the BIG-IP because there is an existing Peer object is also called a *static peer*. In contrast, a *dynamic peer* is a peer that is now known to the BIG-IP by configuration. This includes all client peers, and any server peer to which a BIG-IP connects by explicit declaration in an iRule.

The fact that all client peers are dynamic peers is a subtle but sometimes important point. Because client peers connect to a BIG-IP, and there is no configuration definition for these peers, there is no built-in method for monitoring client peers (or dynamic server peers, for that matter). Dynamic peer state changes can be captured in iRule events (usually, CLIENT\_CLOSED or SERVER\_CLOSED).

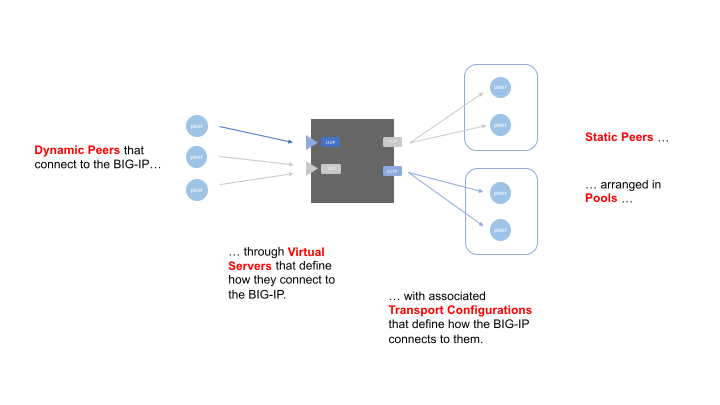


Figure 4 – Peer Types

# Message Phases

When a message arrives on an MRF flow, the message passes through three phases: the *message ingress phase*, the *routing phase*, then the *message egress phase*. The ingress phase happens on the flow on which the message arrived. Routing decides the flow to which the message should be routed. The egress phase then happens on the flow selected by the router. This can be somewhat confusing because the phase is not associated with the flow side. That is to say, the ingress phase can happen on both a client-side flow and a server-side flow. The same is true for the egress phase. It just depends on which flow the message arrives, and out which flow the message is routed. Notice that this even means that the ingress phase and egress phase may both happen on the same flow type. For example, if a message arrives on a server-side flow, and is routed out a different server-side flow, both the ingress and egress phase will happen on server-side flows for that particular message.

# Message Routing

When a message arrives on an MRF flow, it is eventually passed to the message router object associated with that flow. At this point, persistence and routing rules apply. Let’s look at the routing rules first.

Each message router has a *route table*, which, in turn, consists of zero or more *route entries*. A route entry is abstractly the tuple:

[*matching-message-criteria*, *target-peer*, *outgoing-transport*]

The simplest matching-message-criteria is the flow on which the message arrives. This is used when an iRule installs a route. Alternatively, MRF route entries can use personality-specific matchers. For SIP, this includes Request-URI, From header URI and To header URI. For Diameter, this includes the message Application ID, Destination Realm and Origin Realm. For Generic, this includes source IP address and destination IP address. MQTT has no native matchers. In each case, more than one matcher can be combined to form a logical and-wise match. For example, if a Diameter Origin Realm and Destination Realm are identified in a route, only messages that have the named Origin Realm AVP value and the named Destination Realm AVP value would match the route. Where applicable, routes are ordered from most-to-least specific. For example, imagine a Diameter router with two routes: one route contains the Application ID 10 and the Origin Realm “foo.com”, while the other route contains only the Application ID 10. If a message arrives, and it has Application ID 10 and an Origin Realm of “foo.com”, the first route entry will take precedence.

When a personality-specific route is used, its target is one or more Peer objects. Since a Peer object contains both a Peer pool and a Transport Config, the characteristics of the outgoing-transport are provided by the Transport Config object attached to the Peer object[[6]](#footnote-6). If there is more than one Peer object attached to a route, then the *peer selection mode* associated with the route is used to pick the Peer object target. There are two peer selection modes: *sequential* and *ratio*. If the selection mode is sequential, then MRF treats the set of Peers as an ordered list. If the pool for the first Peer object in the list is currently marked as “up”, then that Peer object is used. If it is down, then MRF inspects the pool state of the second Peer object in the list. If it is “up”, then that Peer object is used. If it is “down”, then MRF inspects the pool state of the third Peer object in the list. And so on. When the selection mode is ratio, then a weight is attached to each Peer object in the set. The weight is defined in the Peer object itself. Thus, if a set contains three Peer objects, and their ratio weights are 10, 20 and 30, respectively, then the first Peer object will be selected 10/60 times, the second will be selected 20/60 times, and the third will be selected 30/60 times. Any Peer object with an associated pool that is marked “down” is not used in the ratio calculation. The state of a Peer object pool is determine using the same semantics as is used for other LTM operations. That is, if all members of the pool are marked down by their respective monitors (either inherited from the pool or based on node-specific monitors), then the pool is down. If any member is marked up, then the pool is up.

Once a Peer object is selected, the load-balancing algorithm associated with the pool is used to select the specific target peer. Currently, because of how tmm transport ownership works, only static load-balancing methods (i.e., round robin and ratio) can meaningfully be used.

Once a peer is selected, MRF determines whether there are already one or more existing flows toward the peer using the required transport characteristics. If so, it checks the *number of connections* definition for the Peer object. If there are fewer existing flows than the defined number of connections, MRF will open a new flow toward the peer and use it. Otherwise, MRF will use one of the existing flows.

When an iRule is used to create a route entry, it may identify a Peer object, or it may identify a specific peer host and the transport definition `to use. If a Peer object is identified, then the same rules for transport creation articulated in the previous paragraph are followed. If a specific peer is identified, a new flow is created only if a matching transport does not already exist toward the peer.

SIP and Diameter both use a Request/Response transaction model. Ordinarily, after a Request message is routed to a particular destination flow, when the matching Response message arrives on that destination flow, the Response is routed back to the original Requesting peer. For Diameter, request/response matching uses the hop-by-hop-id and end-to-end-id. For SIP, Call-ID is used. This default return routing can be overridden in an iRule.

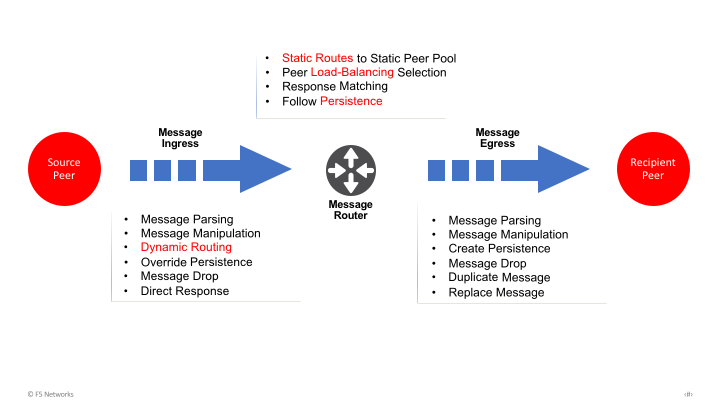


Figure 5 – MRF Message Phases

# Flow Creation

There are two conditions which will trigger the creation of an outbound flow from MRF: 1. as a result of a routing action; and 2. because of auto-initialization. When a route decision is made for a message, the route destination includes a peer and a transport definition. The router checks whether there is an existing transport toward the target peer that satisfies the transport definition characteristics. If so, that existing transport is used. Otherwise, a new transport flow is created, matching the transport definition characteristics.

Auto-initialization is defined in the Peer object. By default, it is disabled. If it is enabled, then MRF will periodically attempt to establish one or more transports toward each peer in the Peer object pool. The period and transport count are also defined in the Peer object.

Importantly, transports created by MRF are bound to the Router object which triggered the creation. For example, imagine two routers (we will call them router1 and router2) each have a route pointing to a common peer. If a message arrives on router1 and this triggers a transport toward the peer, a message arriving on router2 *cannot* use that existing transport. Moreover, if auto-initialization is set on a Peer object, and that Peer object is attached to two different routers, auto-initialized transports will open for each router.

For most MRF personalities, the flow types do not need to match across a route. For example, when using Diameter MRF, a Diameter message may arrive on an SCTP flow and be routed to a TCP flow. As of 14.1, however, MRF does not support passing messages between TCP and UDP for SIP. This is because RFC 3261 section 16.1 requires an agent that moves SIP messages between reliable and unreliable transports to support stateful timer and retransmissions to ensure reliable delivery on the unreliable transport. SIP MRF does not support this function.

# Message Persistence

It is also possible to instruct a router to use *message persistence*. If this is enabled, the router separately maintains a *persistence table* which consists of zero or more *persistence entries*. A persistence entry is abstractly the tuple:

[*persistence-field, persistence-key, source-flow, target-flow*]

The persistence-field is personality specific. For SIP, it includes the Call-ID or the source address of the peer originating a message. For Diameter, it is any AVP. The AVP may be nested inside of a grouped AVP. The persistence-key is the value in the persistence-field that is used to match. When a message from a peer matches the key for the first time, a route decision is made, then a persistence entry is created. The persistence entry notes the flow on which the message arrived and the flow to which it was routed. Any subsequent messages with the same persistence key in the named field bypass normal routing and are delivered on the persisted target-flow. Said differently, all messages from a peer that have the same value in the persistence field will go to the same destination peer.

iRules may be used to add custom persistence or to override an existing persistence entry. This is described in more detail below.

# Peer Sessions

MRF adds additional personality-specific semantics, configured in the Session object and the Router object. These configuration elements are described in more detail below. Note that MRF Generic does not have a Session object, but it does have a Protocol object. This object determines whether MRF uses the built-in parser for Generic message (i.e., a fixed character delimiter between messages) or whether a custom (iRules-based) parser is employed.

The Diameter architecture creates a peer-to-peer relationship between communicating endpoints. The establishment of this relationship requires a capabilities-exchange between the peers, and this, in turn, consists of the exchange of capabilities-exchange Diameter (i.e., CER and CEA) messages. MRF behaves as an explicit Diameter peer, and as such must participate in a capabilities-exchange between itself and each peer to which it connects. Moreover, it must send and respond to both Diameter watchdog requests and disconnect-peer messages. The Diameter Session object defines the capabilities advertised by MRF to peers, and well as watchdog controls. It also defines Host and Realm rewriting for messages traversing through the BIG-IP. This provides Diameter Topology Hiding. Finally, the Diameter Router object defines limits on the number of inflight messages and the amount of inflight data. It also determines whether MRF information is mirrored to a DSC next-active member.

Explicit SIP proxies insert a Via header on request messages, pop their own Via off on response messages, and forward messages to the Via header below them in the Via stack. Via headers are used to control the path for messages, and also to provide loop detection. The SIP Session object controls whether a Via header is inserted by MRF, whether Via headers are followed, whether explicit loop detection is used, and whether a custom Via value is inserted.

# Monitoring Considerations

Because peers are members of an LTM pool, they may be monitored using any of the standard monitors defined for LTM. In addition to these, BIG-IP supports a Diameter monitor and a SIP monitor. The Diameter monitor connects to the target pool member, sends a CER, awaits a CEA, sends a DWR, awaits a DWA, sends a DPR, awaits a DPA, then closes the transport. The SIP monitor connects to the target pool member, sends an OPTIONS request, awaits a valid SIP response, then closes the transport.

While these monitors are useful, they can be problematic in practice. Diameter elements, for example, usually define the set of peer objects that may connect. The monitors will be sourced from a non-floating self-IP. If the BIG-IP is part of a DSC, remote Diameter elements usually configures the floating self-IP as the BIG-IP’s IP address. It may be operationally challenging to configure a peer definition for the non-floating self-IPs on the two DSC members, in addition to the floating self-IP. More importantly, Diameter elements often report (e.g., via an SNMP trap) when a remote peer’s Diameter session and transport are closed. This condition would be triggered every time the Diameter monitor runs. These same issues may also affect SIP elements.

The Diameter protocol specifies a session state-machine, and detects peer liveness using watchdogs. It would be most desirable to use the state of the Diameter session to determine whether a peer is “up”. The *inband* monitor does this. If a Diameter session closes, or the underlying transport for the session is terminated, the inband monitor will mark the peer as “down”. At this point, however, because the peer is marked down, MRF will stop trying to establish connections. Therefore, when the inband is used, a secondary monitor must also be attached which determine when the peer (that is, the pool member) becomes available again. The usual transport-based monitors (e.g., *tcp* or *tcp-half-open*) may not be good candidates for this, if the peer sends a trap or other alarm when a transport connection to the service port goes down.

# Configuring the Message Routing Framework

The configuration objects related to MRF include Virtual Servers, Pools, iRules and MRF Profiles. In TMUI (the BIG-IP web UI), the MRF profiles are found under LTM > Profiles > Message Routing. In tmsh, they are found in the /ltm message-routing namespace, with a sub-namespace for each personality type. The specifics of configuration depend on the personality being used.

## Configuring Diameter MRF

The required steps for configuring Diameter MRF are:

1. Create set of Peer Pools, which are LTM pools, and the desired monitors;
2. Create Diameter Session objects;
3. Create Diameter Transport Config objects (binding Session object and layer4 transport profile objects);
4. Create Diameter Peer objects (binding Peer Pool and Transport Config);
5. Create Diameter Route objects (identifying Peer object targets);
6. Create Diameter Router objects (containing Route objects);
7. Create Virtual Servers (referencing Router object and Session object).

Here are the steps for creating a simple Diameter MRF configuration using tmsh:

create /ltm node pcrf01 address 10.0.1.10

create /ltm node pcrf02 address 10.0.1.11

create /ltm pool pool-pcrf members add { pcrf01:3868 pcrf02:3868 } \

monitor gateway\_icmp

create /ltm message-routing diameter profile session session-gx \

defaults-from diametersession origin-host bigip01 \

origin-realm gx.provider.com

create /ltm message-routing diameter transport-config tc-toward-pcrf \

profiles replace-all-with { f5-tcp-lan session-gx } \

source-address-translation { type automap }

create /ltm message-routing diameter peer peers-pcrf pool pool-pcrf \

transport-config tc-toward-pcrf

create /ltm message-routing diameter route route-to-pcrfs \

peers { peers-pcrf }

create /ltm message-routing diameter profile router router-gx \

defaults-from diameterrouter routes add { route-to-pcrfs }

## Configuring SIP MRF

The required steps for SIP MRF configuration are similar to those for Diameter:

1. Create set of Peer Pools, which are LTM pools, and the desired monitors;
2. Create SIP Session objects;
3. Create SIP Transport Config objects (binding Session object and layer4 transport profile objects);
4. Create SIP Peer objects (binding Peer Pool and Transport Config);
5. Create SIP Route objects (identifying Peer object targets);
6. Create SIP Router objects (containing Route objects);
7. Create Virtual Servers (referencing Router object and Session object).

Here are the steps for creating a simple SIP MRF configuration using tmsh:

create /ltm node sbc01 address 10.1.2.10

create /ltm node sbc02 address 10.1.2.11

create /ltm monitor sip sbc-monitor

create /ltm pool pool-sip-sbcs members add { sbc01:5060 sbc02:5060 } \

monitor sbc-monitor

create /ltm message-routing sip profile session session-sbc \

honor-via enabled insert-via-header enabled \

insert-record-route-header enabled

create /ltm message-routing sip transport-config tc-toward-sbcs \

profiles replace-all-with { f5-tcp-lan session-sbc } \

source-address-translation { type automap }

create /ltm message-routing sip peer peers-sbcs pool pool-sip-sbcs \

transport-config tc-toward-sbcs

create /ltm message-routing sip route route-to-sbcs \

peers { peers-sbcs }

create /ltm message-routing sip profile router router-sbcs \

defaults-from siprouter routes add { route-to-sbcs }

1. A BIG-IP flow represents a complete socket, which is either a connection (e.g., for TCP or SCTP) or a flow with timeout (e.g., for UDP). [↑](#footnote-ref-1)
2. MQTT was added in 14.0 [↑](#footnote-ref-2)
3. HTTP/2 is also implemented on top of MRF, but it is not exposed through the framework. Instead, it is presented as an L7 Virtual Server profile, just like the http profile [↑](#footnote-ref-3)
4. After any manipulation by the BIG-IP, of course. [↑](#footnote-ref-4)
5. A Virtual Server configuration can also be used when a dynamic serverside peer is created. Dynamic peers are described later. [↑](#footnote-ref-5)
6. If the Peer object transport-config parameter is set to None, then the Virtual Server configuration inherited from client that triggered the creation of the serverside flow is used. As noted previously, this should generally be avoided unless there is a compelling reason to do so. [↑](#footnote-ref-6)