BGPmon Test Framework

June 22, 2012

1 Introduction

BGPmon Test Framework is a collection of software and testing data that tests BGPmon components by running it under various conditions. BGPmon Test Framework has multiple test units that run specific tests and also provide a way to monitor and analyse results.

BGPmon Test Framework has following objectives:

- It should help in automation of BGPmon testing process.
- It should increase development productivity.
- It should increase quality of BGPmon components and application.
- Test units need to include conditions that BGPmon application meet in production environment and conditions that difficult to simulate.
- It should generate human-readable test reports.

BGPmon Test Framework is designed for people who interested in testing newly developed or existing components in BGPmon application. BGPmon Test Framework might be interested to people who wants to verify that each particular piece of code that has been written performs the function that it is designed to do. Thus, the audience or users for this document are software developers and quality managers.

1.1 BGPmon Test Framework Overview

Figure 1 shows ideal setup of BGPmon Test Framework that has 7 components. Test Framework is designed in the way that it tests functionality of modules in BGPmon application. BGPmon test instance that shown in the center of the figure has 3 distinct types of input: *Peer*, *MRT* and *Chain* units. First, *Peer* unit, sends BGP messages over the BGP peering session. Second, *MRT* unit, provide BGP messages in MRT format. *Peer* unit is different from *MRT* unit in the way that *Peer* sends BGP messages that are collected directly from a peer, while *MRT* unit provide data from indirect peer through third party. *Chain* logical unit sends BGP messages in a form of XML messages. *Chain* unit is different from *Peer* and *MRT* because it provides messages from other BGPmons. *Peer*, *MRT* and *Chain* units are shown on the left part of Figure

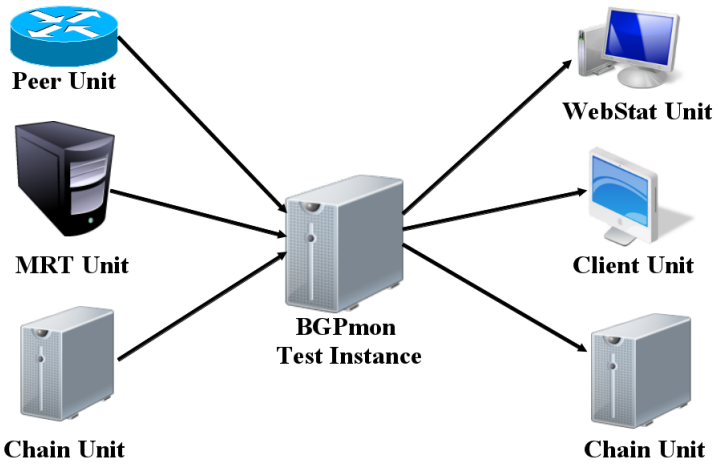

Figure 1: An overview of BGPmon Test Framework architecture.

1. BGPmon test instance has three types of output: *WebStat*, *Client* and *Chain* units. All 3 units receive XML messages from BGPmon test instance but use it in different way. *WebStat* unit use XML feed to generate status report about *Peer*, *MRT* and *Chain* units. *Client* is a setup that used for analysis of XML update messages. *Chain* unit is a separate unit that use XML messages to create a mesh network of BGPmons. *WebStat*, *Client* and *Chain* units are shown on the right part of Figure 1.

In order to run over IPv4 and IPv6 communication protocols, design of BGPmon Test Framework need be to be updated by introducing IPv4 and IPv6 units in input and output to BGPmon test instance. Thus, 3 types of input (*Peer*, *MRT* and *Chain*) and 3 types of output (*WebStat*, *Client* and *Chain*) need to have IPv4 and IPv6 units. For instance, *Peer* logical unit includes *IPv4 Peer* and *IPv6 Peer* testbeds, *MRT* unit have *IPv4 MRT* and *IPv6 MRT* and so forth. Thus, BGPmon Test Framework presents complete picture of a testbed that designed to support all distinct types of input and output in BGPmon application.

1.2 Current BGPmon Test Framework Setup

This section describes installation of BGPmon Test Framework and its current status. Figure 2 shows existing installation and units. BGPmon test instance is installed on *marshal.netsec.colostate.edu* with *129.82.138.29* IPv4 address and *fd68:e916:5287:9f27::101* IPv6 address. Due to lack of available equipment and full IPv6 connectivity, existing BGPmon Test Framework setup has limitations and not all logical pieces present in a

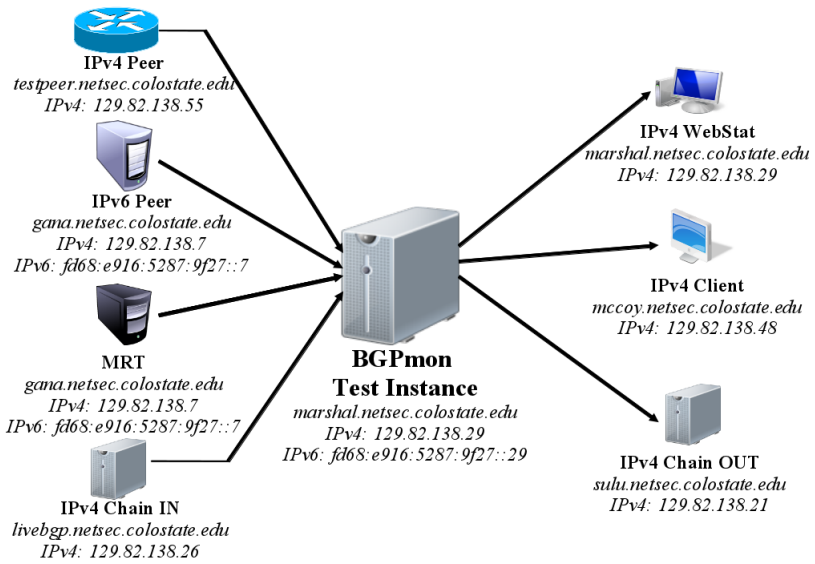

Figure 2: An overview of BGPmon Test Framework architecture.

current installation.

In Figure 2, *Peer* unit includes two testbeds: *IPv4 Peer* and *IPv6 Peer* units. *IPv4 Peer* is a setup of Cisco Router and BGPmon instance. Cisco Router has *testpeer.netsec.colostate.edu* hostname and *129.82.138.55* IPv4 address. Despite that this scheme is designed to test IPv4 peering functionality, it also tests BGPmon components to receive and process BGP update messages. Also, *IPv4 Peer* unit is designed to test BGPmon's efficiency to work with BGP capabilities. *IPv6 Peer* unit is a setup of Quagga Routing Suite software and BGPmon instance. *IPv6 Peer* unit has same goals as *IPv4 Peering*, but it designed to run over IPv6 connectivity. Quagga software is configured on *gana.netsec.colostate.edu* with two IP addresses: *129.82.138.7* is IPv4 address and *fd68:e916:5287:9f27::101* IPv6 address. Section 2 has detailed information about *IPv4 Peer* and *IPv6 Peer* unit goals, configuration defaults and other.

MRT logical unit includes *IPv4 MRT* testbed that is setup of MRT software and BGPmon test instance. MRT software is an application that runs on *gana.netsec.colostate.edu* with *129.82.138.7* IP address. MRT application is designed to provide routing data as a third party. In particular, it provides routing data from existing route collectors like Oregon RouteViews project, but it could be used to give routing data from any other MRT source provider. Section 3 has more details about *IPv4 MRT* unit.

Chain logical unit has *IPv4 Chain IN* and *IPv4 Chain OUT* units. First, *IPv4 Chain IN* unit is configuration of BGPmon application that provide stream of table and update messages in a form of XML messages. *IPv4 Chain IN* runs it on *livebgp.netsec.colostate.edu* and table and update messages are available on ports *50001*

and *50002* respectively. *IPv4 Chain OUT* is setup of BGPmon application that configured to receive XML feed from BGPmon test instance. *IPv4 Chain OUT* is configured on *sulu.netsec.colostate.edu* with *129.82.138.21* IPv4 address. Thus, *IPv4 Chain IN*, BGPmon test instance and *IPv4 Chain OUT* creates a mesh of BGPmons. Overall, *IPv4 Chain* that includes both *IPv4 Chain IN* and *IPv4 Chain OUT* units tests BGPmon's Chain module functionality to create a mesh network of BGPmons, ability of receive and process XML data streams. Section 4 describes details of this test unit.

There are 3 types of output that are shown on the right of Figure 2 *WebStat* unit, *Client* unit and *Chain* unit.

WebStat unit include *IPv4 WebStat* testbed that has WebStat software and BGPmon instance. WebStat software is application that runs on *marshal.netsec.colostate.edu*. WebStat application consist of two components, first, *StatClient* and second, *WebGen*. *StatClient* is application that connects to BGPmon instance and store extracted XML feed in a file system. *WebGen* is unit that generates HTML report that makes summary of configured inputs to BGPmon test instance. Section 5 includes testbed description, test unit goals and others.

Client unit has *IPv4 Client* testbed that is setup of Client software and BGPmon test instance. Client software is application that is configured on *mccoy.netsec.colostate.edu* with *129.82.138.48* IPv4 address. Client software is designed to receive and print XML update messages in a human readable format. Section 6 describes *IPv4 Client* in details.

1.3 BGPmon Test Framework Essentials

This sections describes Test Framework Essentials that are important before running any test units.

Test Framework is configured on *marshal.netsec.colostate.edu*. To get an access to *marshal.netsec.colostate.edu* users of framework need to have access to *bgpmoner* account on *marshal.netsec.colostate.edu*. *bgpmoner* account has all necessary privileges in a system to run or execute Test Framework.

1.3.1 BGPmon Launch

To start using BGPmon Test Framework, BGPmon application has to be launched. To start BGPmon, run following *init.d* script in a terminal:

```
$ sudo /etc/init.d/bgpmon start
```

This command launches BGPmon application in a background.

To stop using BGPmon Test Framework, stop BGPmon application:

```
$ sudo /etc/init.d/bgpmon stop
```

1.3.2 BGPmon Source Code

Source code of BGPmon application is installed in home directory of *bgpmoner* in following directory:

```
/home/bgpmoner/Development/bgpmon-dev
```

1.3.3 BGPmon Init.d Script

Before start using BGPmon application, structure of *init.d* script should be discussed. */etc/init.d/bgpmon* script has following definitions:

```
BGPMON_EXEC=/usr/local/bin/bgpmon
CONFIG_FILE=/usr/local/etc/bgpmon_config.txt
PIDFILE=/var/run/bgpmon.pid
ARGS="-d -c $CONFIG_FILE -s -l 7"
```

BGPMON_EXEC is variable that specifies location of BGPmon executable file. *CONFIG_FILE* shows location of configuration file that store BGPmon settings. *PIDFILE* is variable that points to process ID location that is used by *init.d* script. Lastly, *ARGS* is list of command line arguments. BGPmon application starts with parsing the command line arguments. There are few simple command line arguments that could be specified. In example above, BGPmon uses *-d* to run in daemon world, *-c $CONFIG_FILE* to load default configuration file, *-s* to print log messages to stdlog, *-l 7* defines the log level . In order to debug problems in BGPmon, last two options worth discussion.

User of the framework may configure BGPmon log functionality to use *stdout* or *syslog* modes. First, *stdout mode*, configures BGPmon to run in an interactive mode that sends all messages to standart output (i.e. terminal). To enable this option in *init.d*, change *ARGS* line to following:

```
ARGS="-d -c $CONFIG_FILE -i -l 7"
```

Second, *stdlog mode*, configures BGPmon to run in log mode that sends all messages to a file that specified in syslog. *-s* command line argument enables *syslog mode* and *marshal.netsec.colostate.edu* prints log messages to */var/log/messages* file. To enable *init.d* script use *stdlog mode* use following configuration in *ARGS*:

```
ARGS="-d -c $CONFIG_FILE -s -l 7"
```

BGPmon application supports different log levels. *Init.d* script */etc/init.d/bgpmon* uses option *-l* with log value of 7 (*Debug*) and it includes log values from 0 to 7 (*Emergencies, Alerts, Critical Errors, Errors, Notices, Information, or Debug*). *Debug* provide complete picture of messages that BGPmon application generates. However, users of framework may configure BGPmon Test Framework log reports to use different levels. This option provides different view of generated logs. For instance, if user wants to run framework with lower log level like 4 (*Warning*), it need to run *init.d* script with specified log level value. To launch Test Framework with *Warning* log level, change *ARGS* line in *init.d* script to:

```
ARGS="-d -c $CONFIG_FILE -s -l 4"
```

In general, user of framework is free to set any log level value ranging from 0 to 7 by changing *ARGS* value:

```
ARGS="-d -c $CONFIG_FILE -s -l logvalue"
```

To truly understand the work flow of components in BGPmon Test Framework, BGPmon application supports modular debugging. This feature includes critical messages that are produced by each component during the execution of BGPmon. In order to get the right picture of how components in BGPmon work and communicate between each other, user may enable *DEBUG mode*. Every source file in BGPmon home directory has the following macro at the beginning (after system libraries linking):

```
//DEFINE DEBUG
```

Functions in BGPmon components use *controlled text* to print values to stdout. For example:

```
#ifdef DEBUG
  debug(__FUNCTION__, "New session with id %d for peer %d", i, peerID);
#endif
```

This block is called a *conditional block*. *Debug()* is *controlled text* that will be executed in the output of the preprocessor if and only if *DEBUG* macro is enabled. Most of the functions in BGPmon have at least one conditional block that is wrapped around in *DEBUG* macro.

There are many variants how user can use *DEBUG* macro. For example, to see work flow messages from all components, user may recompile BGPmon source code with "-DDEBUG" option. This will enable *DEBUG mode* in each component in BGPmon application. In this example, every component would start sending very large amount of log messages to log output and it may create difficulty in understanding and debugging founded problems in BGPmon code. Instead, user may uncomment *DEBUG* macro in specific modules. For instance, to enable *DEBUG mode* in *Peering* module in BGPmon application, uncomment *DEBUG* macro :

```
//DEFINE DEBUG
```

to

```
DEFINE DEBUG
```

in each **.c* source file in *Peering* directory in BGPmon source home directory. To see debug messages, user need to recompile source code and install executable files in a system. In this example, all *conditional blocks* that defined in Peering module will be executed. Also, *DEBUG mode* functionality could be enabled in any module or groups of modules to debug the problems in BGPmon application.

However, some users may feel that *DEBUG mode* is not sufficient or there are too little messages in the output. They can create their own conditional blocks. Simply, include conditional block in a function that require additional debugging in source file:

```
#ifdef DEBUG
  debug(__FUNCTION__, "Test values are  %d and  %d", value1,  value2);
#endif
```

In this example, *debug()* function will print the name of the function (where this block was executed) and two integer values *value1* and *value2* .

or what another program was doing at the moment it crashed.

1.3.4 Test Framework Misconfigurations

In order to give a complete picture of BGPmon Test Framework Essentials, user need to know about possible misconfigurations in a testbed and further consequences. Test units in Test Framework might be configured with any test settings including valid or invalid values. Any misconfigurations might brake not only test framework setup, but it may affect already working systems. For example, user may configure to send set of custom MRT update files to *bgpdata.netsec.colostate.edu* hostname that is not shows on Figure 2. Results would be tragic, *bgpdata.netsec.colostate.edu* will provide incorrect data to end clients and it will cause a lot of problems to managers of BGPmon project. In order to prevent such events, user of framework need to use resources (machines, routers, etc) that are that are shown on Figure 2 only. Any usage of other resources is **strictly forbidden** or it has be discussed with developers and managers of BGPmon project.

2 Peer Unit

Peer unit is an input source that sends BGP messages over the BGP peering session. *Peer* unit includes *IPv4 Peer* and *IPv6 Peer* setups that are necessary for testing IPv4 and IPv6 peering function in BGPmon test instance. Overall, *Peer* unit has following goals:

- Keep BGP session: its important to verify that BGPmon can keep peering session alive with IPv4 or IPv6 enabled peer.
- Receive routes: this verifies that BGPmon is able to receive BGP update messages from IPv4 or IPv6 peer.
- Test BGP capabilities: this verifies that BGPmon is able to work with any BGP capabilities.

2.1 IPv4 Peer Unit Overview

Figure 3 shows a test unit design: it has IPv4-enabled Cisco router and testing instance of BGPmon. Cisco router is configured on *testpeer.netsec.colostate.edu* with *129.82.138.55* IPv4 address. BGPmon test instance is installed on *marshal.netsec.colostate.edu* with *129.82.138.29* IPv4 address. In order to run *IPv4 Peer* unit, user of the framework need to be familiar with Cisco router default configuration and BGPmon default configuration that is used for BGP peering session.

2.1.1 IPv4 Peer Unit Launch

To launch BGP peering session between Cisco router and BGPmon test instance:

1. Power on Cisco router in NetSec server room. This will enable Cisco *default configuration*. *Default configuration* setup is discussed in details in Section 2.1.2.
2. Login to *marshal.netsec.colostate.edu*

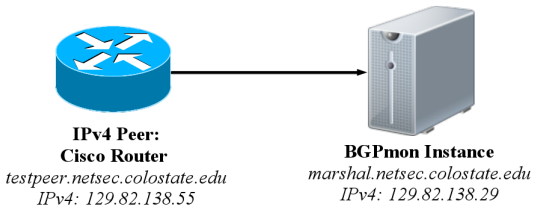

Figure 3: An overview of IPv4 Peering Unit.

(a) Make sure that BGPmon process is up and running. If not, see Section 1.3.

(b) Telnet to *localhost* port *50000* to Command Line Interface.

(c) In *router mode*, launch:

```
marshal(config)# neighbor 129.82.138.55 enable
```

(d) This configuration enabled BGP peer session between BGPmon and Cisco router.

There are few important tips that user of framework need to know while running the *IPv4 Peer* unit.

1. For testing BGP capabilities, BGP peering session may need restart. To reload peer settings and restart peer session, run following in Cisco's telnet:

   ```
   testpeer#clear ip bgp 129.82.138.29
   ```

2. User of framework may reload *default configuration* at any time. This may be useful in starting the new test or in case having misconfiguration in Cisco router. In order to restart *default configuration*, run:

   ```
   testpeer#reload
   System configuration has been modified. Save? [yes/no]: no
   Proceed with reload? [confirm]
   ```

3. User may want to configure Cisco router to announce another network prefix. Cisco router announce network prefixes that it can reach. For example, to start

announcing *11.0.0.0/8* prefix, Cisco router need to be able to reach this prefix via one of interfaces. Thus, full reconfiguration of network interfaces is required. See Section 2.1.2 for setting up *default configuration*.

2.1.2 Cisco Router Default Configuration

Cisco router has minimal set of settings to create BGP session with BGPmon test instance. This section describes Cisco's *default state* and IOS configuration that enables this state. Any configuration requires telnet access to Cisco router.

Figure 3 shows Cisco instance that has *testpeer.netsec.colostate.edu* hostname. To configure hostname, run following command in *configure mode*:

```
testpeer(config)#hostname testpeer.netsec.colostate.edu
```

In order to configure BGP peering settings, Cisco router requires configuration of two network interfaces. First network interface is configured to *129.82.138.0/23* subnet and has *129.82.138.55* IPv4 address. *129.82.138.0/23* network is a network of NetSec Group at Colorado State University and *129.82.138.55* IP address should be used for Cisco router only. Second network interface is setup of *139.179.0.0/16* subnet with *139.179.96.1* IPv4 address. *139.179.0.0/16* subnet is used for test purpose only and it should never appear in global routing table or anywhere else except the Test Framework.

To configure first network interface, run following command in Cisco's telnet:

```
testpeer(config)#interface FastEthernet 0/0
testpeer(config-if)#ip address 129.82.138.55 255.255.255.128
testpeer(config-if)#no shutdown
```

To configure second network interface, run:

```
testpeer(config)#interface FastEthernet 1/0
testpeer(config-if)#ip address 139.179.96.1 255.255.0.0
testpeer(config-if)#no shutdown
```

In Figure 3 Cisco router and BGPmon instance negotiate peering session. Cisco router uses following network configuration for peering:

- IPv4 address: *129.82.138.55*.
- AS number: *64513*.

To bring up BGP peering session, user need to enable IP routing and configure network prefix. Run following in telnet:

```
testpeer(config)#router bgp 64515
testpeer(config)#network 139.179.0.0 255.255.0.0
```

To enable BGP peering session with BGPmon test instance, run:

```
testpeer(config)#neighbor 129.82.138.55 remote-as 64515
testpeer(config)#end
```

Installation above is minimal set of configuration rules that *IPv4 Peer* unit require.

In order to have this state permanently, user need to save this configuration as *default configuration*. To save, run following command:

```
testpeer#copy running-config startup-config
```

This command saves Cisco *default configuration* to the startup configuration file in NVRAM. *Default configuration* set is used everytime when Cisco router is turned on.

2.1.3 BGPmon Test Instance Default Configuration

By default BGPmon application is installed on *marshal.netsec.colostate.edu* and uses *129.82.138.29* IPv4 address and *64515* AS number for BGP peering session with Cisco router.

To enable BGP peering session, run following command in *router mode* in CLI:

```
marshal(config)# neighbor 129.82.138.55 enable
```

2.1.4 Result Reporting

In order to verify if peering configuration in BGPmon was successfully enabled, run following command in *privileged mode* in CLI of BGPmon test instance:

```
marshal#show bgp neighbor 129.82.138.55
```

This command shows status information about peering session, both BGPmon and Cisco configuration values, announced BGP capabilities and others.

In order to check that BGPmon received BGP update message from Cisco router, run following command:

```
marshal# show bgp routes 129.82.138.55
```

This command shows IPv4 prefix that Cisco router announce. Also, it includes NextHop, AS path, AS length information from Cisco peer.

In order to verify if peering configuration in Cisco router was enabled, Cisco router provides a report of IPv4 peering session with BGPmon. To verify that IPv4 peering session is established and running, run following in *configuration mode* in Cisco's telnet:

```
marshal# show ip bgp neighbors 129.82.138.29
```

This command will show the status of IPv4 peering on Cisco router. For example, in case of successfully configured peering session it will show following log:

```
BGP neighbor is 129.82.138.29,  remote AS 64515, external link
BGP version 4, remote router ID 129.82.138.29
BGP state = Established
```

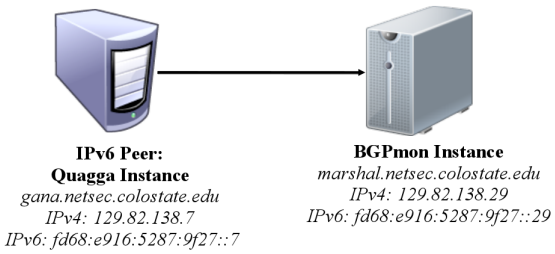

Figure 4: An overview of IPv4 Peering Unit.

2.2 IPv6 Peer Unit

IPv6 Peer unit includes setup of Quagga Routing Suite software and BGPmon instance. Due to the old version of Cisco IOS in *IPv4 Peer* unit and lack of IPv6 protocol support in IOS, Quagga Routing Suite software was choosen to run BGP peering session with BGPmon instance. Figure 4 shows a *IPv6 Peer* unit design: it has dual-stack IPv4 and IPv6 enabled Quagga and BGPmon instances. Quagga instance is installed on *gana.netsec.colostate.edu* and it has one physical network interface that configured for IPv4 and IPv6 IP addresses. Quagga instance has following configuration for BGP peering session with BGPmon test instance:

- Hostname: *gana.netsec.colostate.edu*
- IPv4 address: *129.82.138.7*
- IPv6 address: *fd68:e916:5287:9f27::7*
- AS number: *64514*

BGPmon is installed on *marshal.netsec.colostate.edu* with IPv4 and IPv6 addresses. BGPmon test instance use following configuration for BGP peering session with Quagga:

- Hostname: *marshal.netsec.colostate.edu*.
- IPv4 address: *129.82.138.29*.
- IPv6 address: *fd68:e916:5287:9f27::29*.
- AS number: *64515*.

In order to run *IPv46Peer* unit, user of the framework need to be familiar with Quagga and BGPmon configurations that are used for BGP peering over IPv6 protocol.

2.2.1 IPv6 Peer Unit Launch

To launch BGP peering session between Quagga instance and BGPmon test instance:

1. Login to *gana.netsec.colostate.edu*
   (a) Make sure that *bgpd* processes is up and running. Bgpd process uses *default configuration* set that makes Quagga instance ready for BGP peering. For more details, see Section 2.2.2.
2. Login to *marshal.netsec.colostate.edu*
   (a) Make sure that BGPmon process is up and running. If not, see Section 1.3.
   (b) Telnet to *localhost* port *50000* to Command Line Interface.
   (c) In *router mode*, launch:

   ```
   marshal(config)# neighbor  fd68:e916:5287:9f27::7 enable
   ```

   (d) This configuration enables IPv6 BGP peer session between BGPmon and Quagga instance.

There are few important tips that user of framework need to know while running the *IPv6 Peer* unit.

1. For some tests like testing BGP capabilities, BGP session restart may be required. To reload peer settings and restart peer session, run following in Quagga's telnet:

   ```
   testpeer#clear ip bgp fd68:e916:5287:9f27::29
   ```

2. User of framework may reload *default configuration* at Quagga instance at any time. This may be useful in starting the new test or in case having misconfiguration in Quagga instance. In order to restart *default configuration*, run following command on *gana.netsec.colostate.edu*:

   ```
   $ sudo /usr/local/etc/rc.d/quagga restart
   ```

2.2.2 Quagga Default Configuration

Quagga software uses *bgpd* daemon for BGP peering. To configure BBG peering, start *bgpd* process on *gana.netsec.colostate.edu*

```
$ sudo /etc/init.d/quagga start
```

Quagga instance has minimal set of settings to create BGP session with BGPmon test instance. This section describes Quagga's *default state* and configuration that enables this state. Any configuration requires telnet access to Quagga instance.

Figure 4 shows Quagga instance that has *gana.netsec.colostate.edu* hostname. To configure hostname, run following command in *configure mode*:

```
gana(config)#hostname gana.netsec.colostate.edu
```

To bring up BGP peering session, user need to enable IP routing and configure network prefix. Run following in telnet:

```
gana(config)#router bgp 64515
gana(config-router)#bgp router-id 129.82.138.7
gana(config-router)#address-family ipv6
gana(config-router)#network 2001:220::/35
```

To enable BGP peering session with BGPmon test instance, run:

```
gana(config-router)#neighbor fd68:e916:5287:9f27::29 remote-as 64515
gana(config-router)#end
```

Installation above is minimal set of configuration rules that *IPv6 Peer* unit require.

In order to have this state permanently, user need to save this configuration as *default configuration*. To save, run following command:

```
gana#copy running-config startup-config
```

This command saves Quagga *default configuration* to the startup configuration file. *Default configuration* set is used everytime when Quagga instance is started on *gana.netsec.colostate.edu*.

2.2.3 BGPmon Test Instance Default Configuration

By default BGPmon application is installed on *marshal.netsec.colostate.edu* and uses *fd68:e916:5287:9f27::29* IPv6 address and *64515* AS number for BGP peering session with Quagga instance.

To enable BGP peering session, run following command in *router mode* in CLI:

```
marshal(config)# neighbor fd68:e916:5287:9f27::7 enable
```

2.2.4 Result Reporting

In order to verify if configuration in BGPmon test instance was successfully enabled, run following command in *privileged mode*:

```
marshal#show bgp neighbor fd68:e916:5287:9f27::7
```

This command shows status information about BGP peering session, announced BGP capabilities and others.

In order to check that BGPmon received and processed BGP update message correctly, run following command:

```
marshal# show bgp routes fd68:e916:5287:9f27::7
```

This command shows IPv6 prefix that Quagga instance announce. Also, it includes NextHop, AS path, AS length information from Quagga peer.

To verify if configuration in Quagga instance was enabled, Quagga provides a report of IPv6 peering session with BGPmon. To verify that IPv6 peering session is established and running, run following command in *configuration mode* in Quagga's telnet:

```
bgpd# show ip bgp neighbors fd68:e916:5287:9f27::29
```

This command shows the status of IPv6 peering, received BGP capabilities and others.

3 MRT Module Test Unit

The *MRT* unit allows one to inject MRT messages into the BGP Test Framework. MRT messages do not come directly from a peer router. Instead they are collected by a third party and then reported to BGPmon. An MRT message consists of a header followed by a BGP message. The header specifies the time when the message was collected and the peer that sent the message.

The MRT unit supports two models for injecting MRT data. In the *end-user* model an external user supplies the MRT data as set of files. For example, a user may create MRT files that include BGP messages from one of her peer routers. In the *collector* model, the tester provides a name of RouteViews collector, a start time, and an end time. For example, the tester may specify *route-views2.oregon-ix.net*, starting from September 1st at 9:00 am, ending on September 12th at 3:00 pm.

Unlike live data, MRT files have been collected already and each MRT message includes a time stamp. The testing unit plays back the MRT messages, preserving the timing between messages. For example, if the first BGP message originated at *1:20:00 am* and second BGP message at *1:25:00 am*, the MRT testing unit will send the first BGP message, sleep for *5 minutes* and only then send the second BGP message.

Figure 5 shows the test unit design: it includes an MRT sender and BGPmon receiver. The MRT sender is installed on *gana.netsec.colostate.edu* with an IPv4 address of *129.82.138.7* and an IPv6 address of *fd68:e916:5287:9f27::7*. The BGPmon receiver is installed on *marshal.netsec.colostate.edu* with an IPv4 address of *129.82.138.29* and IPv6 address of *fd68:e916:5287:9f27::29*.

3.1 Configuring BGPmon to Receive MRT

To configure BGPmon to receive MRT data:

1. Login to *marshal.netsec.colostate.edu*
2. Make sure that the BGPmon process is up and running. If not, see Section 1.3.
3. Telnet to *localhost* port *50000* to access the Command Line Interface.
4. In *configuration mode*, launch the *mrt-listener*:

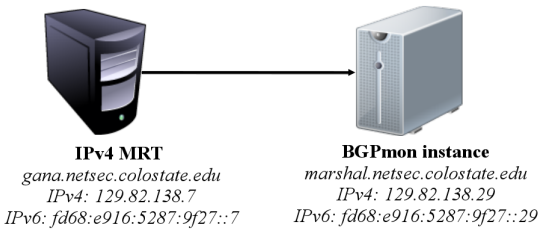

Figure 5: An overview of IPv4 MRT Unit.

```
marshal$ enable
marshal# configure
marshal(config)# mrt-listener enable
marshal(config)# end
```

This command instructs BGPmon to listen for TCP connections on *marshal.netsec.colostate.edu*. *Mrt-listener* will use IPv4 address of *129.82.138.29*, port *7777* and IPv6 address of *fd68:e916:5287:9f27::29*, port *7777* for incoming connections.

The user of framework may configure *mrt-listener* to listen for TCP connections on IPv4 address only. Run following command in *configuration mode*:

```
marshal$ enable
marshal# configure
marshal(config)# mrt-listener address 129.82.138.29
marshal(config)# mrt-listener enable
marshal(config)# end
```

To configure *mrt-listener* to listen for TCP connections on IPv6 address only, run:

```
marshal$ enable
marshal# configure
marshal(config)# mrt-listener address fd68:e916:5287:9f27::29
marshal(config)# mrt-listener enable
marshal(config)# end
```

The tester of the framework can check status of *mrt-listener*. Run following command in Command Line Interface:

```
marshal>show mrt-listener status
```

This command shows *mrt-listener* status. Successful configuration of *mrt-listener* will show *enabled* status. Failed configuration of *mrt-listener* will show *disabled* status.

The tester of the framework may check the network IP addresses configured in *mrt-listener*:

```
marshal>show mrt-listener address
```

To disable the *mrt-listener* in BGPmon, run:

```
marshal$ enable
marshal# configure
marshal(config)# mrt-listener disable
marshal(config)# end
```

This command instructs *mrt-listener* to close TCP connections and change internal status to *disabled*.

3.2 Sending MRT Messages

The MRT unit supports two models for injecting MRT data. In the *end-user* model an external user supplies the MRT data as set of files. In the *collector* model, the tester provides a name of RouteViews collector, a start time and an end time.

3.2.1 The End-user Model

This model assumes the tester has one or more MRT files. The tester may use his own MRT file or sample MRT file. A sample MRT file (*updates.20110916.1715*) is available on *gana.netsec.colostate.edu* in the directory *∼bgpmoner/Development/bgpmon-dev/test/mrt_harness/etc*. This file was generated by *route-views.oregon-ix.net* on *September 16th 2011, 17:15 pm*.

The tester of the framework need to be familiar with MRT unit directories on *gana.netsec.colostate.edu*. *∼bgpmoner/Development/bgpmon-dev/test/mrt_harness/etc* is directory that designed to store long-lasting files. Directory includes files that are the part of the MRT unit. *∼bgpmoner/Development/bgpmon-dev/test/mrt_harness/data* is temporary working directory. *∼bgpmoner/Development/bgpmon-dev/test/mrt_harness/data* is created to store an optional data including the testing MRT files. The user of the framework is allowed to store temporary files in *∼bgpmoner/Development/bgpmon-dev/test/mrt_harness/data* directory.

To start MRT unit test, run following:

1. Download the MRT files into *∼bgpmoner/Development/bgpmon-dev/test/mrt_harness/data* folder or copy sample MRT file from *∼bgpmoner/Development/bgpmon-dev/test/mrt_harness/etc* directory.

2. Login to *gana.netsec.colostate.edu*.

   (a) To send MRT data from *filename*, run:

```
$ mrtfeeder -f filename -d marshal.netsec.colostate.edu
```

where:

- *filename* is MRT file name in *~bgpmoner/Development/bgpmon-dev/test/mrt_harness/data*. See Section 3.4.2 for the details about supported *filename* extensions.
- *marshal.netsec.colostate.edu* is hostname of BGPmon receiver. MRT feeder converts *marshal.netsec.colostate.edu* hostname to an IPv4 and an IPv6 socket address structures. MRT feeder connects to BGPmon receiver using one of socket address structures.

(b) The tester of the framework may test MRT unit over IPv4 connection. Use the IPv4 address of BGPmon receiver to run MRT feeder:

```
$ mrtfeeder -f filename -d 129.82.138.29
```

(c) The tester of the framework may test MRT unit over IPv6 connection. Use the IPv6 address of BGPmon receiver to run MRT feeder:

```
$ mrtfeeder -f filename -d fd68:e916:5287:9f27::29
```

3. To see the results, see Section 3.3.

The MRT feeder send MRT file with MRT messages to BGPmon receiver. As long as TCP connection is alive between MRT feeder and BGPmon receiver, BGPmon receiver keeps track of MRT peers and received routing data. Once the MRT feeder reaches the last MRT message in MRT file, it will close TCP connection to BGPmon. *Mrt-listener* will be triggered to destroy all MRT sessions and processed routes. To avoid this situation and help the tester in debugging, the MRT feeder has an option that keeps TCP connection alive. The tester of the framework can run the MRT feeder with following option:

```
$ mrtfeeder -q -f filename -d marshal.netsec.colostate.edu
```

where

- *-q* instructs MRT feeder to keep TCP connection alive after the last MRT message is sent to BGPmon.

In order to stop running MRT feeder, the tester of the framework need to type following command and press *RETURN* key on a keyboard.

```
exit
```

This command will terminate MRT unit work.

3.2.2 The Collector Model

In the collector model, the tester specifies the RouteViews collector, start time and end time. The MRT unit injects the MRT messages from a specified RouteViews collector using adjusted time frame.

The RouteViews collectors provide two types of MRT files: an MRT table file and an MRT update file. The table MRT file is the snapshot of the MRT table messages. The update MRT file is a set of MRT update messages. The tester of the the framework provide the time frame when the MRT unit starts sending MRT snapshot and MRT update files. The MRT unit approximate the start time to fetch the most recent MRT snapshot file and MRT update files. The MRT unit sends MRT update files in a sequence based on the specified start time and end time.

To start MRT unit test, run following:

1. Login to *gana.netsec.colostate.edu*.
2. Run *mrtfetcher* application with following options:

   ```
   $ mrtfetcher -c collector -s starttime -e endtime
      -d marshal.netsec.colostate.edu
   ```

   where:

   - *collector* can be *route-views.oregon-ix.net*, *route-views2.routeviews.org* or other collectors. The list of existing collectors is available at `http://www.routeviews.org/` website.
   - *starttime* is a unix timestamp that specifies the start time.
   - *endtime* is a unix timestamp that specifies the end time.
   - *marshal.netsec.colostate.edu* is hostname of BGPmon receiver. MRT feeder converts *marshal.netsec.colostate.edu* hostname to an IPv4 and an IPv6 socket address structures. MRT feeder connects to BGPmon receiver using one of socket address structures.

   Mrtfetcher will send MRT files from the collector starting at time *starttime* and ending at *endtime*. Section 3.4.3 describes the design of *mrtfetcher* application.
3. The tester of the framework may test MRT unit over IPv4 connection. Use the IPv4 address of BGPmon receiver::

   ```
   $ mrtfetcher -c collector -s starttime -e endtime
      -d 129.82.138.29
   ```
4. The tester of the framework may test MRT unit over IPv6 connection. Use the IPv6 address of BGPmon receiver:

   ```
   $ mrtfetcher -c collector -s starttime -e endtime
      -d fd68:e916:5287:9f27::29
   ```
5. The tester of the framework may configure MRT fetcher to skip sending MRT table messages and send MRT update messages only:

   ```
   $ mrtfetcher -u -c collector -s starttime -e endtime
      -d marshal.netsec.colostate.edu
   ```

where

- *-u* flag instructs MRT fetcher to send the MRT update messages to BGPmon.

6. To see the results, see Section 3.3.

BGPmon receiver receives MRT data and keeps track of MRT sessions with routing data. Once the MRT fetcher reaches the last MRT message in MRT file, the MRT fetcher will close TCP connection to BGPmon. *Mrt-listener* will be triggered to destroy all MRT sessions and processed routes. To avoid this situation and help the tester in debugging, the MRT fetcher has an option that keeps TCP connection alive. The tester of the framework can run the MRT fetcher with following option:

```
$ mrtfetcher -q -c collector -s starttime -e endtime
     -d marshal.netsec.colostate.edu
```

where

- *-q* instructs MRT fetcher to keep TCP connection alive after the last MRT message is sent to BGPmon.

In order to stop running MRT fetcher, the tester of the framework need to type following command and press *RETURN* key on the keyboard.

```
exit
```

This command will terminate MRT fetch unit.

3.3 Result Reporting

The MRT unit provides a set of commands for debugging the problems in BGPmon.

In order to verify if MRT unit successfully established a TCP connection to BGPmon receiver, run following command in Command Line Interface on BGPmon receiver:

```
marshal>show mrt clients
```

This command lists the number of active MRT senders.

To check if BGPmon instance successfully created MRT sessions, run:

```
marshal>show mrt neighbor
```

This command shows the list of established MRT sessions that were extracted from MRT messages.

To check if BGPmon instance received routing data from MRT messages, user need to use destination IP address from MRT sessions in BGPmon. For example, if there is MRT session with *12.0.1.63* IP address, run following command to see received routes:

```
marshal>show bgp routes 12.0.1.63
```

This command shows a report that includes the routing information extracted from MRT messages for *12.0.1.63* MRT session.

3.3.1 MRT Analyzer

To aid in debugging, the MRT unit also has an MRT Analyzer. The Analyzer takes an MRT file as the input and outputs a number of statistics about the peers in MRT file. For instance, it outputs *peerlist.txt* file that contains IP addresses of peers in MRT file. MRT Analyzer uses *bgpdump* tool to analyse MRT messages. It extract routes from MRT file. For each peer Analyzer prints BGP messages in human-readable format. For instance, for a peer with *12.0.1.63* IP address, MRT Analyzer will create *12.0.1.63_table.txt* file that includes following messages:

```
BGP4MP|09/16/11 17:29:55|A|12.0.1.63|7018|
                189.45.27.0/24|7018 3549 28654 28340|IGP
BGP4MP|09/16/11 17:29:55|A|12.0.1.63|7018|1
                86.193.99.0/24|7018 3549 28654 28340|IGP
BGP4MP|09/16/11 17:29:55|A|12.0.1.63|7018|
                189.45.24.0/24|7018 3549 28654 28340|IGP
```

To run MRT Analyzer:

1. Login to *gana.netsec.colostate.edu*.
2. Run MRT Analyzer:

   ```
   $ mrtanalyzer -c filename
              -d ~bgpmoner/Development/bgpmon-dev/test/mrt\_harness/data
   ```

   where:

   - *filename* is MRT file in *∼bgpmoner/Development/bgpmon-dev/test/mrt_harness/data*.
   - *∼bgpmoner/Development/bgpmon-dev/test/mrt_harness/data* is the output directory.

3.4 MRT Source Code

Source code of MRT unit is located in *∼bgpmoner/Development/bgpmon-dev/test/mrt_harness/data* directory. *MRT feeder*, *MRT fetcher* and *MRT analyzer* source code is available in *∼bgpmoner/Development/bgpmon-dev/test/mrt_harness/data/src* folder.

If user need to reinstall MRT application on *gana.netsec.colostate.edu*, run following commands:

```
$ cd ~bgpmon/Development/bgpmon-dev/test_support/mrt_harness
$ make
$ sudo make install
```

This command will install *MRT feeder*, *MRT fetcher* and *MRT analyzer* applications into the system.

3.4.1 Timeframing Algorithm Design Overview.

The MRT unit sends the MRT messages using *timeframing* algorithm. Each MRT message in the MRT file has a timestamp. The timestamp in MRT header was created in the past, when the BGP message appeared at the third party. To play the MRT messages according to the timestamp in the MRT header, MRT unit calculates the time shift value or, simply, the *offset*. The offset helps to adjust the time when message need to be sent. The offset is calculated as the difference in local time and the timestamp from the the first MRT message. Thus, by introducing the offset value, each MRT message requires an approximation of the *sendtime*. The sendtime indicates the current send time of the MRT message. The sendtime is calculated as the sum of timestamp from MRT header and the offset value. In order to play the MRT messages at the right speed, the MRT unit compares the calculated *sendtime* and the current local time. If the difference is bigger than 0, MRT unit will sleep for the difference time and only then send the MRT message.

In order to provide valid MRT data, the timeframing algorithm perform sanity check over the MRT messages. In particular, it extracts MRT header values from the MRT message and checks the data validity. For example, a length value of BGP message in MRT header should not be bigger than the BGP message maximum size (4096 bytes). The type value in MRT header need to be equal to one from the MRT draft document. The details of values and it's sanity check functions are described in the *playMRT()* function design.

The timeframing algorithm include the *mrtplayer.c* and the *mrtplayer.h* library files. The *mrtplayer.h* is a header file that used for function declaration. The *mrtplayer.c* file has source code of functions that are used in the timeframing algorithm. The *mrtplayer.c* includes the following functions:

1. **playMRT() function**

   Definition:

   ```
   int playMRT(File *p, int socket);
   ```

   Purpose: playMRT() function reads the MRT file, extracts MRT messages and send them to the socket.

   Inputs: the MRT file descriptor, the socket file descriptor.

   Return values:

   playMRT() function has following return values:

   - 0: success. Function was executed successfully.
   - 1: failure. Problem in opening the file descriptor.
   - 2: failure. The MRT header read is less then 12 bytes.
   - 3: failure. The MRT header read failed.
   - 4: failure. Sanity check for timestamp value in MRT header failed
   - 5: failure. Sanity check for type value in MRT header failed.

- 6: failure. Sanity check for subtype value in MRT header failed.
- 7: failure. Sanity check for length value in MRT header failed.
- 8: failure. The BGP message read is less then the length bytes value.
- 9: failure. The BGP message read failed.
- 10: failure. Send of MRT header failed.
- 11: failure. Send of BGP message failed.

Functionality:

(a) The playMRT() function checks if the MRT file is open. The playMRT() function check if file descriptor is not equal to *NULL*, otherwise returns 1.

(b) The playMRT() function reads 12 bytes of the MRT header from the file descriptor. The MRT header is a structure that has four values: *u_int32_t timestamp*, *u_int16_t type*, *u_int16_t subtype*, *u_int32_t length*. The playMRT() call *fread()* system function to read the data. *fread()* function return the number of items successfully read. playMRT() function checks the return value:

- *fread* returns 12. No problem occurred.
- *fread* returns value between 1 and 11. The end of the file is reached or error occurred. The playMRT() function returns 2.
- *fread* returns 0. Error occurred. The playMRT() function returns 3.

(c) The playMRT() function performs sanity check for four values in MRT header.

- The playMRT() calls internal function *checkTimestamp()* function. *checkTimestamp()* function takes *u_int32_t timestamp* in input. The playMRT() function check the return value from *checkTimestamp()* function. Return value of 0 means sanity check for timestamp succeed. If *checkTimestamp()* function returns 1, sanity check failed and the playMRT() function returns 4.
- The playMRT() calls internal function *checkType()* function. *checkType()* function takes *u_int16_t type* in input. The playMRT() function check the return value from *checkType()* function. Return value of 0 means sanity check for type succeed. If *checkType()* function returns 1, sanity check failed and the playMRT() function returns 5.
- The playMRT() calls internal function *checkSubType()* function. *checkSubType()* function takes *u_int16_t type* and *u_int16_t subtype* in input. The playMRT() function check the return value from *checkSubType()* function. Return value of 0 means sanity check for subtype succeed. If *checkSubType()* function returns 1, sanity check failed and The playMRT() function returns 6.
- The playMRT() calls internal function *checkBGPlength()* function. *checkBGPlength()* function takes *u_int32_t length* in input. The playMRT()

function check the return value from *checkBGPlength()* function. Return value of 0 means sanity check for length succeed. If *checkBGPlength()* function returns 1, sanity check failed and The playMRT() function returns 7.

(d) The playMRT() function calls *ntohl()* system function to convert the value of length in MRT header file from network byte order to host byte order.

(e) The playMRT() function reads *N* bytes of BGP message from the file descriptor. The *N* is the converted value of length in MRT header. The playMRT() calls *fread()* system function to read the data. *fread()* function return the number of items successfully read. The playMRT() function checks the return value:

- *fread* returns *N*. No problem occurred.
- *fread* returns value between 1 and *N*. The end of the file is reached or error occurred. The playMRT() function returns 8.
- *fread* returns 0. Error occurred. The playMRT() function returns 9.

(f) The playMRT() function calls *ntohl()* system function to convert the value of timestamp in MRT header from network byte order to host byte order.

(g) The playMRT() function calculates the *offset* for the timeframing algorithm. The offset is calculated as the difference of current local time and the timestamp value from the first MRT message. The offset value is calculated once and it never should be changed.

(h) The playMRT() function calculates the *sendtime* for the MRT message. The sendtime is the sum of the timestamp value in MRT header and the offset value.

(i) The playMRT() function calculates the *M* value. The *M* is the difference of local time and sendtime. The *M* value is used to check if MRT message is ready to be sent. If the *M* value is bigger than 0, The playMRT() uses the sleep function with *M* seconds.

(j) The playMRT() function uses *send()* function to send MRT header and BGP message in the socket. *send()* function is called twice. First, to send the MRT header. Second, to send the BGP message. Upon successful completion, *send()* shall return the number of bytes sent. Otherwise, -1 shall be returned. The playMRT() function checks the return value of *send()* function:

- For MRT message, *send()* returns 12. No problem occurred.
- For MRT message, *send()* return -1. Error occured. The playMRT() function returns 10.
- For BGP message, *send()* returns *N*, where *N* is the length value in MRT header. No problem occurred.
- For BGP message, *send()* returns -1. Error occured.The playMRT() function returns 11.

(k) Upon successful sent of MRT header and BGP message, the playMRT() function continues to read the rest of file and use the algorithm described above. The playMRT() function extracts MRT header and BGP message until it reaches the end of the file and, in success, it returns 0.

2. **checkTimestamp() function**

   Definition:

   ```
   int checkTimestamp(u_int32_t timestamp);
   ```

   Purpose: sanity check for timestamp value. checkTimestamp() function verifies if the input timestamp value belongs to the range of the 0 (unix time beginning) and the current unix timestamp.

   Inputs: the timestamp value from the MRT header structure.

   Return values:

   - 0: success. Function was executed successfully.
   - 1: failure. Sanity check failed.

   Functionality:

   (a) checkTimestamp() function uses *ntohl()* system function to convert the timestamp value from network byte order to host byte order.

   (b) checkTimestamp() function checks if the the converted timestamp value is bigger the 0 and less than the current time. If timestamp value is in the range of (0;current time), checkTimestamp() function returns 0. Otherwise, it returns 1.

3. **checkType() function**

   Definition:

   ```
   int checkType(u_int16_t type);
   ```

   Purpose: sanity check for type value. checkType() function verifies if the input type value is equal to at least one type value that defined in MRT draft document.

   Inputs: the type value from the MRT header structure.

   Return values:

   - 0: success. Function was executed successfully.
   - 1: failure. Sanity check failed.

   Functionality:

   (a) checkType() function uses *ntohs()* system function to convert the type value from network byte order to host byte order.

(b) checkType() function checks if the the converted type value is equal to one of the following type value: 11 (OSPFv2 type), 12 (TABLE_DUMP type), 13 (TABLE_DUMP_V2 type), 16 (BGP4MP type), 17 (BGP4MP_ET type), 32 (ISIS type), 33 (ISIS_ET type), 48 (OSPFv3 type), 49 (OSPFv3_ET type). In success, checkType() function returns 0. Otherwise, it returns 1.

4. **checkSubType() function**

   Definition:

   ```
   int checkSubType(u_int16_t type, u_int16_t subtype);
   ```

   Purpose: sanity check for subtype value. checkSubType() function verifies if the subtype value and the type value match to at least one category of subtype and type values in MRT draft document.

   Inputs: the type value and the subtype value from the MRT header structure.

   Return values:

   - 0: success. Function was executed successfully.
   - 1: failure. Sanity check failed.

   Functionality:

   (a) checkSubType() function uses *ntohs()* system function to convert the type value from network byte order to host byte order.

   (b) checkSubType() function uses *ntohs()* system function to convert the subtype value from network byte order to host byte order.

   (c) Not all types have a subtype, checkType() check specific subtypes for the following types: 12 (TABLE_DUMP type), 13 (TABLE_DUMP_V2 type), 16 (BGP4MP type), 17 (BGP4MP_ET type). For type 12, checkSubType() function checks if subtype if equal to 1 or 2. For type 13, checkSubType() checks if subtype is equal to 1, 2, 3, 4, 5 or 6. For type 16 or 17, checkSubType() function checks if subtype if equal 0, 1, 2, 3, 4, 5, 6 or 7. For other types, checkSubType() does not check subtype value. In success checkSubType() return 0, in failure 1.

5. **checkBGPlength() function**

   Definition:

   ```
   int checkBGPlength(u_int32_t length);
   ```

   Purpose: sanity check for length value. checkBGPlength() function verifies if the input length value is bigger then 0 but less then the max size of BGP message (4096 bytes).

   Inputs: the length value from the MRT header structure.

   Return values:

- 0: success. Function was executed successfully.
- 1: failure. Sanity check failed.

Functionality:

(a) checkBGPlength() function uses *ntohl()* system function to convert the length value from network byte order to host byte order.

(b) checkBGPlength() function checks if the the converted length value is bigger the 0 and less than 4096. In success, it returns 0. In failure, it returns 1.

6. **MRTconnect() function**

Definition:

```
int MRTconnect(char *hostname);
```

Purpose: MRTconnect() function creates a socket connection to the given hostname with PORT 7777.

Inputs: hostname string.

Return values:

- N: success. Function returns a non-negative socket file descriptor.
- -1: failure. The *getaddrinfo()* function failed.
- -2: failure. The *socket* function failed.
- -3: failure. The *connect* function failed.

Functionality:

(a) The MRTconnect() function identifies specified hostname. The MRTconnect() function uses the *getaddrinfo()* system function. The *getaddrinfo()* takes four inputs: the given hostname, the port number, *hints addrinfo* and *res addrinfo* data structures. The hostname is given hostname from the input. *getaddrinfo()* uses port 7777 to connect to hostname. The *hints addrinfo* input points to an *addrinfo* structure that specifies criteria for selecting the socket address structures returned in the list pointed to by *res addrinfo*. The MRTconnect() function checks the return value from the *getaddrinfo()* function:

- Upon successful completion, *getaddrinfo()* returns 0.
- *getaddrinfo()* returns non-zero error codes that correspond to failure . The MRTconnect() function returns -1.

(b) The MRTconnect() function uses the *res addrinfo* data structure to create a socket connection. The MRTconnect() function uses the *socket()* system function to get the socket file descriptor. The *socket* function takes the *ai_family*, *ai_socktype* and *ai_protocol* values from the *res addrinfo* data structure. The MRTconnect() function checks the return value from *socket()* function:

- Upon successful completion, *socket()* returns a non-negative integer (socket file descriptor.)
- In a failure, *socket()* function returns -1. The MRTconnect() returns -2.

(c) The MRTconnect() function uses *connect()* function to establish a connection. The *connect* function attempts to make a connection on a socket. The function takes the following arguments: the socket file descriptor, *ai_addr* and *ai_addrlen* values from *res addrinfo* data structure. The MRTconnect() function checks the return value from the *connect* function:

- Upon successful completion, *connect()* function return 0.
- In a a failure, -1 is returned. The main() function closes the socket file descriptor and returns -3.

(d) The MRTconnect() function return the socket file descriptor.

3.4.2 MRT Feeder Design Overview

The MRT feeder is an application that designed to inject MRT messages into the BGPmon Test Framework. It uses the *timeframing* algorithm to send the MRT messages from the provided MRT file. The MRT feeder application include the *mrtfeeder.c* and the *mrtfeeder.h* files. The *mrtfeeder.h* is a header file that used for function declarations. The *mrtfeeder.c* is a source code file of MRT feeder application. Also, the MRT feeder application includes the *mrtplayer.h* header file and use the functions defined in timeframing library. *mrtplayer.c* includes the following functions:

1. **main() function**

   Definition:

   ```
   int main(int argc, char **argv)
   ```

   Purpose: the main() function starts MRT feeder application.

   Inputs: *argc* and *argv* values. The argc is a count of the arguments supplied to the program and the argv is an array of pointers to the strings which are the arguments to main function. The arguments are passed to the program by the host system's command line interpreter.

   Return values:

   The MRT feeder uses the return values that are defined in *stdlib.h* system library:

   - EXIT_SUCCESS. function execution is successful.
   - EXIT_FAILURE: function execution failed.

   Error log printing:

   main() function uses the *perror()* system function to print error messages. *perror()* function produces a message on the standard error output, describing the last error encountered during a call to a system or library function.

   Functionality:

(a) The main() function starts with parsing the arguments that are provided by the user. The main() function uses the *getopt()* system function. The *getopt* is design to break up (parse) options in command lines for easy parsing by shell procedures, and to check for legal options. MRT feeder *geptop()* function takes three inputs. First, the [-f filename] option is designed to use provided filename from the arguments. Second, the [-d hostname] option is designed to use provided hostname from the arguments. Third, the [-u] options is designed to set the *keepTCP* integer to 1. Details of [-u] option are discussed further. The main() function runs the while loop with true condition and checks the return value from the *getopt()* function. Once the *getopt* parsed the command line, it returns -1. The main() function stops the while loop.

(b) The main() function checks the filename extension. The user of the MRT feeder may provide archived MRT files in input. The main() function uses the *strtok()* system function. The strtok() function parses a string into a sequence of tokens. For a given filename, *strtok()* function will lookup for a delimiter. The main() function uses ".bz2" string as a delimiter. *strtok()* inputs the filename and the delimiter and returns following values:

- A NULL pointer: strtok() function could not find any token that matches the given delimiter. The main() function will not extract the filename.
- A pointer to the last token found in filename. This means that provided filename is an archive and it need to be extracted. The main() function use *system()* function to extract the filename. *system()* function executes a command specified in input by calling */bin/sh -c* command. *system()* function uses two inputs: the *bunzip2* tool to extract the MRT file and the given filename. The main() function checks the return value of *system()* function:
  - *system* function returns -1. This an error. The main() function exits with EXIT_FAILURE value
  - *system* function returns the return status of the execution of *bunzip2* command. *bunzip2* returns 0 for a normal exit and 1 for environmental problems (file not found, etc.) In case of 1, the main() function exits with EXIT_FAILURE. In case or 0, filename was successfully extracted.

(c) The main() function opens the filename. The main() function uses the *fopen()* system function to open the filename. *fopen()* function takes two inputs, first the given filename and second, the mode. The main() function uses "rb" mode to open binary MRT file for a read. The main() function checks the return value of *fopen* function.

- *fopen* function returns a NULL pointer. An error occured. The main() function exits with EXIT_FAILURE value.
- Upon successful completion *fopen* function returns the file descriptor.

(d) The main() function uses the *MRTconnect()* function from the *mrtplayer.c* library. *MRTconnect()* takes the hostname in input and returns the socket

file descriptor. The main() function checks the return value from the *playMRT()* function.

- In a success, the *MRTconnect()* function returns a non-negative value (the socket file descriptor).
- In a failure, the *MRTconnect()* function returns a error number. The main() function uses the *switch()* system function to print the corresponding error message. In a failure, the main() function closes the file descriptor and exits with EXIT_FAILURE value.

(e) The main() function uses the *playMRT()* function from the *mrtplayer.c* library. The *playMRT()* function takes two inputs: the file descriptor and the socket file descriptor. The *playMRT()* function send the MRT data via socket connection. The main() function checks the return values from the *playMRT()* function. The main() function checks the return value from the *playMRT()* function.

- In a success, the *playMRT()* function returns 0.
- In a failure, the *playMRT()* function returns a error number. The main() function uses the *switch()* system function to print the corresponding error message. In a failure, the main() function closes the file descriptor, the socket file descriptor and exits with EXIT_FAILURE value.

(f) After the the main() function sent the MRT file, it checks the *keepTCP* integer value. The *keepTCP* value is set to 1 if the MRT fetcher application was started with *-u* options at the command line. In the design of MRT feeder, this option is created to leave TCP connection open until the user types "exit" in command line.

- If *keepTCP* is enabled, the main() uses a while loop with true condition and it uses two functions: *fgets()* and *strcmp()*. In the while loop the main()function calls *fgets* to read the data from the command line. The *fgets* function has three inputs: the *char buffer*, the size of the char buffer and the source of the stream. The main() function define the char buffer *buf* with size of 100 bytes. The source of the stream is *stdio*. The main() function checks the *fgets()* function return values.
  - On error, the *fgets()* function return NULL. The main() function closes
  - Upon successful completion, *fgets()* returns the read characters from command line.
- In the while loop, the main() function use the *strcmp()* function. The *strcmp()* function compares the two input strings. The first input string is string from the char buffer that *fgets* provides. The second string is defined "exit" string. The main() function checks the *strcmp()* function return values.
  - If two strings are equal, the *strcmp()* function returns 0. The main() function breaks the while loop.

- If two strings are different, the *strcmp()* function returns the byte difference between two strings. The main() function continues to loop the while loop and reads the input from the command line.

(g) The main() function uses *close()* function to close the file descriptor and the socket file descriptor. Lastly, the main() function will exit with EXIT_SUCCESS value.

3.4.3 MRT Fetcher Design Overview

The MRT fetcher application is designed to run in the *collector mode*. The MRT fetcher application goal is to imitate the behaviour of the routing collector. The routing collector sends two types of data: the MRT table snapshot that provides a set of routing tables from connected peers and sequential set of MRT update messages that update the routing information. The The MRT fetcher application is designed to send MRT data from the existing routing collectors. In particular, it uses the RouteViews collector's archives to fetch the MRT data. The MRT fetcher takes two unix timestamps: the *start time* and the *end time*. The *start time* specifies the time when MRT fetcher sends the MRT messages. The *end time* specifies the time when the MRT fetcher stops sending the MRT data. The RouteViews collectors provides the MRT tables that are generated every two hours and the MRT update files that are generated every 15 minutes. The MRT fetcher application goal is to send the MRT data that originated between *start time* and the *end time*.

The MRT fetcher is designed in following way. First, the MRT fetcher calculates the *table time*. The *table time* is unix time that specifies the most recent (in the range of 2 hours) MRT table snapshot. The *table time* is calculated as the nearest integer divisible by 7200. For example, if the input *start time* is equal to *1317203100* (Sept 22, 2011, 9:45:00 am), the *table time* is calculated as integral part of *star time* divided by 7200 and multiplied by 7200. The calculation gives the *table time* to be equal to 1317196800 (Sept, 22, 2011, 8:00:00 am) and it gives the most recent unix time of MRT table file. The MRT fetcher downloads the proper MRT table file from the RouteViews collector archive. The MRT fetcher extracts the archive and use *send()* function to send the MRT table file via socket connection. Second, the MRT fetcher is designed to play MRT update messages at the right speed that corresponds to specified *start time* and *end time*. In the example above, the MRT fetcher is 1 hour and 45 minutes below the requested *start time*. In order to get to the MRT update messages originated at *start-time*, the MRT fetcher sends MRT update messages starting from *table time* and ending the *start time* values. The MRT fetcher does not apply timeframing algorithm to those messages. The MRT fetcher fetches and sends 7 MRT update files that fall in the 1 hour and 45 minutes gap. Third, the MRT fetcher is ready to play the messages that originated from *start time* and ends on the *end time*. The MRT fetcher fetches each MRT update message from the RouteViews archive and sends it using the *timeframing* algorithm.

The *timeframing* algorithm plays the messages at the right speed. This means that the set of MRT messages from the MRT update file is spread across 15 minutes interval. In the *timeframing* algorithm two or more MRT messages may have a few seconds delay. And *timeframing* algorithm calculates the time time when those messages need

to be send. In MRT fetcher, fetching the MRT file and extracting the MRT file takes the time. The MRT fetcher creates a small timing gap between the two MRT files. This gap is measurable in order of few seconds and does not affect the overall speed of sending MRT files.

1. **main() function**

   Definition:

   ```
   int main(int argc, char **argv)
   ```

   Purpose: the main() function starts MRT fetcher application.

   Inputs: *argc* and *argv* values. The argc is a count of the arguments supplied to the program and the argv is an array of pointers to the strings which are the arguments to main function. The arguments are passed to the program by the host system's command line interpreter.

   Return values:

   main() function uses the return values that are defined in *stdlib.h* system library:

   - EXIT_SUCCESS. function execution is successful.
   - EXIT_FAILURE: function execution failed.

   Error log printing:

   The main() uses the *perror()* system function to print error messages. *perror()* function produces a message on the standard error output, describing the last error encountered during a call to a system or library function.

   Functionality: NEEDS WORK: section requires a lot details.

   (a) The main() uses the *getopt()* function to parse command line.

   (b) The main() checks the RouteView collector name. Based on collector name string, the main() function chooses the right URL to fetch the MRT data.

   (c) The main() function calls the *MRTconnect()* function twice to establish two TCP connecitons. One is designed to send MRT table file, another - to send the MRT update file.

   (d) The main() function uses the starttime from the input and calculates the *table time*.

   (e) The main() uses the *libcurl* library to fetch the MRT table file from the Routeview collector. The *libcurl* saves archive in */home/bgpmoner/Development/bgpmon-dev/test/mrt_harness/data* folder.

   (f) The main() function extracts the MRT table file.

   (g) The main() function sends the MRT file using the first socket file descriptor.

   (h) The main() function closes the socket file descriptor.

(i) The main() function uses three timestamps variables: the *table timestamp*, the *start time* and the *end time*. For a range of [*table timestamp; start time*), the main() function fetches the corresponding MRT update file, extract it and use *send()* system function to write the data to the socket file descriptor. If [*table timestamp; start time*) includes many MRT update files, the main() function will repeat the previous step until the last MRT update that falls within the range. For a range of [*start time; end time*), the main() function fetches the corresponding MRT update file, extract it and use the *timeframing* algorithm to send MRT messages at the right speed. If [*start time; end time*) includes many MRT update files, the main() function will repeat the previous step until the last MRT update that falls within the range.

(j) For each of MRT update files, the main() function uses the *libcurl* library to fetch the MRT update messages.

(k) The main() function closes the socket file descriptor.

4 Chain Unit

Chain unit is another type of input to BGPmon. *Chain* unit provide XML messages that include BGP messages from other BGPmon instances. However, *Chain* unit can be a type of output source: it is a client to BGPmon that receives XML messages. *Chain* unit include two types of chain setups: *Chain IN* and *Chain OUT* units. First, *Chain IN* is setup of BGPmon application that sends XML messages to BGPmon test instance. Second, *Chain OUT* is setup of BGPmon application that receives XML messages from BGPmon test instance. Thus, *Chain IN* and *Chain OUT* units are designed to create a mesh network of BGPmons. Overall, *Chain* unit is designed to archive following goals in a BGPmon Test Framework:

- Test BGPmon instance to scale out through chaining other BGPmons.
- Receive and send XML messages: this verifies that BGPmon is able to receive XML messages from connected chain instance and forward XML messages to other chain instaces.

4.1 IPv4 Chain IN Unit

Figure 6 shows test unit design: it has *IPv4 Chain IN* unit on the left and BGPmon testing instance on the right. *IPv4 Chain IN* unit is installed on *livebgp.netsec.colostate.edu* with *129.82.138.26* IPv4 address. *IPv4 Chain IN* unit provide stream of XML update messages on port *50001* and stream of XML table messages on port *50002*. BGPmon test instance is configured on *marshal.netsec.colostate.edu* with *129.82.138.29* IPv4 address.

4.1.1 IPv4 Chain IN Unit Launch

To start IPv4 Chain IN unit:

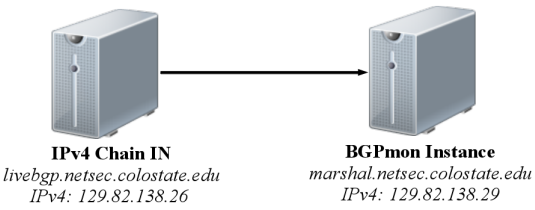

Figure 6: An overview of Chain IN Unit.

1. Login to *marshal.netsec.colostate.edu*
   (a) Make sure that BGPmon process is up and running. If not, see Section 1.3.
   (b) Telnet to *localhost* port *50000* to Command Line Interface.
   (c) In *configuration mode*, launch:

```
marshal# chain 129.82.138.26 enable
```

2. IPv4 Chain IN instance that run on *livebgp.netsec.colostate.edu* does not require any configuration, its ready for use.

4.1.2 IPv4 Chain IN Unit Configuration

In order to start providing XML messages, *IPv4 Chain IN* unit require to have at least one of three inputs of BGP messages. User if free to configure *Peer*, *MRT* or another *IPv4 Chain* unit.

4.1.3 BGPmon Test Instance Configuration

In order to run *IPv4 Chain IN* unit, user of framework need to enable chain at BGPmon test instance. To enable chain session, log in to *marshal.netsec.colostate.edu* and ensure that BGPmon process is up and running.

To enable chain from *IPv4 Chain IN* unit to BGPmon test instance, run following command in Command Line Interface (CLI):

```
marshal# chain 129.82.138.26 enable
```

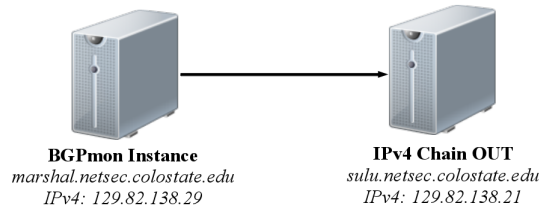

Figure 7: An overview of Chain OUT Unit.

At this point, BGPmon test instance opens two opens two TCP connections to *IPv4 Chain IN* unit and receives stream of XML messages.

To disable chain to *IPv4 Chain IN* unit, run:

```
marshal# chain 129.82.138.26 disable
```

4.1.4 Result reporting

In order to verify if configuration was successfully enabled, run following command in CLI on BGPmon test instance:

```
marshal>show chains
```

This commands lists chain session that was enabled. In case of successfully enabled chain session, CLI will show following report:

```
marshal>show chains
ID address uport rport retry status
0 129.82.138.26 50001 50002 60   enabled
```

4.2 IPv4 Chain OUT Unit

Figure 7 shows test unit design: it has *IPv4 Chain OUT* on the fight unit and BGPmon testing instance on the left. *IPv4 Chain OUT* unit is installed on *sulu.netsec.colostate.edu* with *129.82.138.21* IPv4 address. BGPmon test instance is configured on *marshal.netsec.colostate.edu* with *129.82.138.29* IPv4 address. In this design BGPmon test unit provide stream of XML update messages on port *50001* and stream of XML table messages on port *50002*.

4.2.1 IPv4 Chain OUT Unit Configuration

In order to run *IPv4 Chain OUT* unit, user of framework need to enable chain at *IPv4 Chain OUT* unit. To enable chain session, log in to *sulu.netsec.colostate.edu* and ensure that BGPmon process is up and running.

To enable chain from BGPmon test instance to *IPv4 Chain OUT* unit, run following command in Command Line Interface (CLI):

```
marshal# chain 129.82.138.29 enable
```

At this point, *IPv4 Chain OUT* unit opens two opens two TCP connections to BGPmon test unit and receives stream of XML messages.

To disable chain to BGPmon test instance, run:

```
marshal# chain 129.82.138.29 disable
```

4.2.2 BGPmon Test Instance Configuration

To start sending XML messages, BGPmon test instance require to have at least one of three inputs of BGP messages. User if free to choose any available like *Peer*, *MRT* or another *IPv4 Chain* unit.

4.2.3 Result reporting

In order to verify if configuration was successfully enabled, run following command in CLI on *IPv4 Chain OUT* unit:

```
marshal>show chains
```

This commands lists chain session that was enabled. In case of successfully enabled chain session to BGPmon test unit, CLI shows following report:

```
marshal>show chains
ID address uport rport retry status
0 129.82.138.29 50001 50002 60   enabled
```

5 WebStat Unit

WebStat unit is output type unit that connects to BGPmon. *WebStat* unit receive stream of XML messages from BGPmon. In particular, *WebStat* unit filters BGPmon's stream of XML messages to process only those messages that corresponds to XML *Status* messages. XML *Status* messages include information from three types of input to BGPmon: *Peer*, *MRT* and *Chain* units. *Peer* and *MRT* units provide a detailed summary about peering sessions, memory usage, number of received BGP messages and others. *Chain* unit provide information about connected chain instances. *WebStat* unit use data from all 3 inputs to create a HTML report and user of framework may use it to monitor the state of BGPmon.

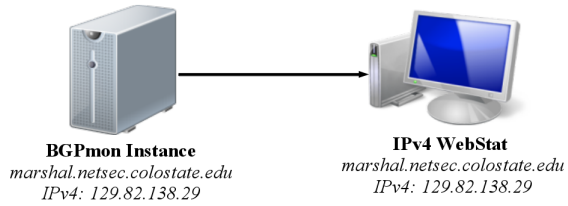

Figure 8: An overview of IPv4 WebStat Unit.

In order to use *WebStat*, user of framework need to be familiar with design of *WebStat* unit. *WebStat* unit is a set of two applications: *StatClient* and *WebGen*. *StatClient* is application that connects to BGPmon instance and retrieve XML data. Also, it filters *Status* messages from entire set of XML messages. *Status* messages contain information about *Peer*, *MRT* and *Chain* units. *StatClient* periodically saves filtered data to the filesystem. *WebGen* is an application that use data from filesystem to generate HTML page that includes statistical data and figures that shows changes over the time. In order to have the most current information from *Peer*, *MRT* and *Chain* input sources *WebGen* runs in crontab scheduler and periodically updates HTML page.

WebStat unit is designed to achieve following goals:

- BGPmon provide XML messages in output: its important to verify that BGPmon is able to send stream of XML messages to *WebStat* unit .
- Generate Statistics: HTML report shows summary of *Peer*, *MRT* and *Chain* units.

5.1 IPv4 WebStat Unit

Figure 8 shows unit design: it has *IPv4 WebStat* unit and BGPmon test instance. Both *IPv4 WebStat* unit and BGPmon test instance are installed on same host *marshal.netsec.colostate.edu* with *129.82.138.29* IPv4 address.

5.1.1 IPv4 WebStat Unit Configuration

IPv4 WebStat unit has two applications: *IPv4 StatClient* and *IPv4 WebGen*.

IPv4 StatClient is application that connects to BGPmon test instance and receives XML messages. *IPv4 Stat client* supports following input values:

```
$ statclient [IP] [PORT] [DIRECTORY]
```

where *[IP]* is IP address of BGPmon instance. *[PORT]* is a port of BGPmon server. *[DIRECTORY]* is directory on *marshal.netsec.colostate.edu* where XML messages are stored.

IPv4 WebGen is application that uses data output from *IPv4 StatClient* unit and generates a HTML report. To make HTML report, run *WebGen* as follows:

```
$ webgen [SOURCE DIRECTORY] [DESTINATION DIRECTORY]
```

where *[SOURCE DIRECTORY]* is a directory with XML data from *IPv4 StatClient* and *[DESTINATION DIRECTORY]* is public html directory where HTML will be saved.

5.1.2 BGPmon Test Unit Configuration

To enable XML feed on BGPmon instance, user has to be logged to *marshal.netsec.colostate.edu* and ensure that BGPmon process is up and running. To enable stream of XML messages, run run following command in *configuration mode* in CLI:

```
marshal(config)#client-listener enable
```

BGPmon test instance starts to listen TCP connections on *50001* and *50002* on *marshal.netsec.colostate.edu*. Port *50001* provides a stream of XML update messages, port *50002* sends XML table messages.

5.1.3 Result reporting

WebStat Client Test Unit provide a report of extracted data from XML stream. Test unit generates HTML page that is browsable at `http://bgmon.netsec.colostate.edu/framework/stat.html` page. In case of successfully received XML data, HTML report contains information about configured peer, MRT, chain sessions that are configured at BGPmon test instance. Overall, generated result report is beneficial in debugging problems related to any of *Peer*, *MRT* or *Chain* units.

6 Client Unit

Client unit is output type unit that connects to BGPmon. *Client* unit receives stream of XML update messages from BGPmon and prints it in human readable view. The output from *Client* unit includes important debugging values like network prefix and AS path. *Client* does not rely on source of input to BGPmon, it uses XML messages from BGPmon. Overall, *Client* unit is designed to achieve following goals:

- BGPmon provide XML messages in output: its important to verify that BGPmon sends XML update messages to the *Client* unit.
- XML messages include values that were configured by user of framework at *Peer*, *MRT* or *Chain* units.

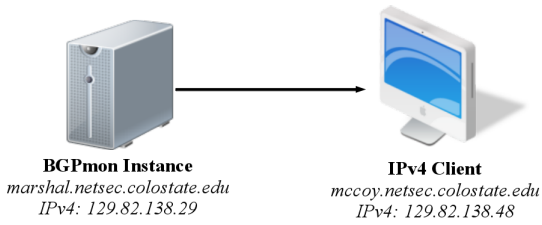

Figure 9: An overview of IPv4 Client Unit.

6.1 IPv4 Client Unit

Figure 9 shows unit design: it has *IPv4 Client* unit and BGPmon test instance. *IPv4 Client* unit is installed on *mccoy.netsec.colostate.edu* with *129.82.138.48* IPv4 address. BGPmon test instance is configured on *marshal.netsec.colostate.edu* with *129.82.138.29* IP address.

6.1.1 IPv4 Client Unit Configuration

IPv4 Client unit is application that connects to BGPmon test instance and receives XML update messages. *IPv4 Client* supports following input values:

```
$ statclient [IP] [PORT]
```

where *[IP]* is IP address of BGPmon instance. *[PORT]* is a port of BGPmon server.

6.1.2 BGPmon Test Unit Configuration

To enable XML feed on BGPmon instance, user has to be logged to *marshal.netsec.colostate.edu* and ensure that BGPmon process is up and running. To enable stream of XML messages, run run following command in *configuration mode* in CLI:

```
marshal(config)#client-listener enable
```

BGPmon test instance starts to listen TCP connections on *50001* and *50002* on *marshal.netsec.colostate.edu*. Port *50001* provides a stream of XML update messages, port *50002* sends XML table messages.

6.1.3 Result reporting

Client unit prints BGP messages in human readable format. This output can be used in debugging any of three types of input to BGPmon. For instance, user may configure setup of BGP peer that announce *11.0.0.0/8* network prefix. After BGP peer establish BGP peering session with BGPmon, it will send BGP update message to BGPmon. *Client* will receive XML update message and print network information to the user.