

BGPmon Version7

Implementation and Technical Specification

He Yan, Mikhail Strizhov, Kevin Burnett, Dave Matthews and Daniel Massey

Colorado State University

Computer Science Department

Fort Collins, Colorado, USA 80523

[yanhe,burnet,strizhov,dvmtthws,massey]@cs.colostate.edu

ABSTRACT

This paper describes the implementation and technical specification for BGPmon, a new BGP monitoring system. BGPmon uses a publish/subscribe approach to provide real-time access to vast numbers of peers and clients. BGPmon scales up by eliminating route selection, forwarding functions, and routing table dumps to focus solely on the monitoring function. Using features such as route refresh, all events including periodic table transfers are consolidated into a single XML stream. XML allows us to add additional features such as labeling updates to allow easy identification of useful data by clients. Clients subscribe to BGPmon and receive the XML stream, performing tasks such as archiving, filtering, or real-time data analysis. This document captures the underlying implementations decisions behind BGPmon. The intended audience is a reader who wants to understand how BGPmon was built and possibly enhance the design with new features.

1. INTRODUCTION

To truly understand and properly analyze BGP routing, one needs data from a wide range of sites with different geographical locations and different types (tiers) of ISPs. Fortunately global routing monitoring projects such as Oregon RouteViews[6] and RIPE RIS[5] have been providing this essential data to both operations and research community. Google Scholar lists hundreds of papers that cite these monitoring sources. Results on route damping, route convergence, routing policies, power-law topologies, routing security, routing protocol design, and so forth have all benefited from this data. Operational uses of the data range from analyzing the impact of major events such as fiber cuts to detecting prefix hijacks in real-time. All this work is possible because there are monitoring systems to collect and publish BGP data.

One can collect BGP data directly from a commercial router (e.g. use commands such as `show ip bgp`) or one can use open source routing toolkits to collect and log data. However, experience over the years has also shown there are major limitations in adopting tools not

specifically designed for the BGP data collection process. An ideal monitoring system would scale to a vast numbers of peer routers and provide BGP data in real-time to an even larger number of clients. For example, one might like to add peers from different geographic locations and lower tier ISPs. At the same time, real-time access would enable new tools for analysis of events such as fiber cuts, prefix hijacks, and so forth. The system should also reflect the fact that BGP is still evolving and the system should be easily extended to include updates such as the expansion to four byte AS numbers, improved security for peering, and any number of current or future extensions to the protocol.

This paper presents the implementation and specification of a next generation BGP monitoring system. We propose a mesh of interconnected data collectors and data brokers that operate using a publish/subscribe model. Our approach extends the scalable event driven architecture in [7] to meet the requirements of BGP monitoring. Interested clients receive an event stream in real-time or may read historical event streams from archival sources. The single stream incorporates both incremental BGP update messages and periodic routing table snapshots. We use XML to provide easy extensibility, integrate with common tools, and allow local data annotations. The backbone of this system is BGPmon, a scalable and extensible tool for collecting and publishing BGP data.

Readers interested only in the overall BGPmon system and general approach should refer to the technical paper[1] for an overview of the BGPmon system. Readers interested only in installing, configuring, and running BGPmon should refer to the BGPmon Administrators Reference Manual[2].

This paper describes the implementation of BGPmon. The objective is to document design decisions and provide a detailed picture of how BGPmon is implemented. The intended audience is a reader who is interested in understanding the implementation details of BGPmon and it is expected many readers will use this document as a companion to the open source code. For

example, someone interested in adding a new feature to BGPmon should consult this document along with the source code in order to understand how the system currently operates and where to make enhancements.

1.1 Document Overview

The sections in the remainder of this paper roughly correspond to directories in the BGPmon source code tree. Readers interested in modifying or understanding portions of the BGPmon implementation do not need to read the entire document. The discussion below describes which sections correspond to which functions and recommends who should read a particular section.

Section 2 roughly corresponds to the main program(`main.c`) which reads and saves the configuration, listens for signals, and is responsible for starting, stopping, and monitoring all other modules' threads. The overall architecture of BGPmon is also discussed here. The section is useful to most readers in order to broadly understand the BGPmon implementation.

Section 3 roughly corresponds to the *Configuration* directory in the source code. It provides facade functions to read and save the configuration which are called by main program and call the module specific read/save functions. In addition, the *Configuration* directory also provides the utility functions to parse the XML configuration file to other modules. Programmers interested in facade functions and XML utility functions should read this section.

Section 4 roughly corresponds to the *Peering* directory in the source code and handles all functions related to opening and maintaining sessions with BGP peers. Programmers interested in adding new BGP capabilities or modifying how peering sessions are managed should read this section.

Section 5 roughly corresponds to the *Mrt* directory in the source code and handles all functions related to receiving and parsing MRT data.

Section 6 roughly corresponds to the *Labeling* directory in the source code and handles the optional storing or RIBIN tables and the optional addition of labels to BGP updates. Programmers interested in adding new labels or annotations to BGP update messages should read this section.

Section 7 roughly corresponds to the *PeriodicEvents* directory in the source code and handles all actions related to periodic events. Periodic events include requesting a route refresh from a peer, announcing a RIBIN table for peers that do not support route refresh, and sending status messages regarding peers, queues and chains. Generally speaking, this module generates every message that is reported to clients but not exchanged over a peering session. Programmers interested in adding or modifying periodic events or reporting any type of BGPmon state should read this section.

Section 8 roughly corresponds to the *XML* directory in the source code and handles the XML formatting of messages. Programmers interested in modifying the XML format or changing which XML tags are included in a message should read this section.

Section 9 roughly corresponds to the *Clients* directory in the source code and handles all actions related to accepting client connections and delivering XML formatted data to clients.

Section 10 roughly corresponds to the *Chains* directory in the source code and handles chaining BGPmon instances together in order to form a mesh. Programmers interested in enhancing BGPmon chaining features should read this section.

Section 11 roughly corresponds to the *Queues* directory in the source code and handles the message queueing operations. Programmers interested in BGPmons queue management and dampening algorithms should read this section.

Section 12 roughly corresponds to the *Login* directory in the source code and handles command line related operations. The command line interface is similar to Cisco IOS. Programmers interested in command line interface should read this section.

Section ?? concludes the document and provides references to related documents.

2. OVERALL ARCHITECTURE AND MAIN MODULE

BGPmon is divided into multiple modules. The division into modules is driven by both conceptual and implementation objectives. On a conceptual level, BGPmon is designed to be extensible and it should be possible to enhance portions of BGPmon without modifying the entire code base. For example, one may want to add new BGP capabilities or add new periodic reporting features or change the configuration of chains. Our design separates the code into distinct modules so the code related to adding a new BGP capability is isolated in the peering module, to the maximum extent possible. Similarly, code related to periodic features would be in a periodic module and code related to chains configuration is in the chain module. This modular design is intended to help facilitate additions at all levels.

At an implementation level, the modules are designed to take maximum advantage of threading. Current trends in computer architecture are moving toward multi-core processors and threading programs can take maximum advantage of such designs. Rather than attempt to micro-manage control, we allow the operating system to do this whenever possible. For example, we may be receiving data from multiple peers and not all of these peers will be sending data simultaneously. Rather than building our own logic to select between peers, we launch a separate thread for each peer and rely on

the operating system to fairly share resources among threads.

Figure 1 shows the overall system architecture and division into modules.

2.1 Modules

Configuration Module: provides the functions to read and save BGPmon configuration. These functions call module specific read/save configuration functions. Additionally, configuration module provides the XML utility functions which are needed by other modules. There is no threads associated with this module.

Peering Module: manages the conguration for all peers and one peering session for each enabled peer. The peer conguration could be changed by command line interface via interfaces provided by peering module. Peering session management includes opening the session, negotiating capabilities, monitoring the session state, receiving data from the session and reestablishing the session when needed. The BGPmon design allocates one thread for each peering session. All messages received from (and sent to) a peer are written into the *Peer queue* for processing by later modules.

Labeling Module: manages one RIB-IN table associated with a peer if configured and uses the tables to assign labels to updates received from the peer. The RIB tables may also be used by the periodic module discussed below. Storage of RIB tables is optional and set by the administrator. A single dedicated thread handles RIB tables and labeling for all peers. The Labeling Module reads from the the *Peer Queue* and writes to the *Labeled Queue*.

Periodic Event Handling Module: manages periodic events such as requesting a route refresh from a peer and periodically announcing the status of peering sessions, queues and chains. There are two threads in this module: one handles the route refreshes requests and another handles the periodic status messages. By combining all periodic events in one place, BGPmon can better manage events. For example, the Periodic Module can stagger route refresh requests to large numbers of route table transfers do not occur at once. Any messages written by the Periodic Module are placed into the *Labeled Queue*.

XML Module: manages the conversion of all BGPmon received and generated messages into XML. A single dedicated thread handles all XML conversion. The XML Module reads from the *Labeled Queue* and writes to the *XML Queue*.

Clients Management Module: manages the server thread of BGPmon, accpeting connections from clients and other BGPmons. Once a connection is accepted, a distinct client thread is created for each client. Client modules read from the *XML Queue*

Chains Module: allows BGPmon instances to link to-

gether in a chain. In the chain, the XML output of one BGPmon is fed directly into the XML queue of a second BGPmon instance. For example one may deploy a BGPmon instance at an exchange point in London and deploy a second BGPmon instance at an exchange point in Amsterdam. Both instances could chain to a third BGPmon in Oregon. Clients that connect to the Oregon receive an XML stream that includes data from all three BGPmons. The client is unaware BGPmons chains exist. This chaining can later be combined with BGPbrokers who perform additional data processing tasks and produce a powerful monitoring network. Any data received by the BGPmon Chains Module is written directly to the *XML Queue*.

Queue Module: handles the inevitable queuing issues that arise when bursty data sources provide real-time data to clients that accept data at different rates. BGPmon implements queue management and pacing features that adjust clients who are too slow and attempt to pace bursts from routers so peering sessions are not dropped.

Login Module: handles the Cisco-like commands typed by logged users and calls the corresponding functions provided by other modules.

2.2 BGPmon Internal Format

Messages are processed by various modules and flow through various queues until eventually being converted into an XML format and sent to clients as shown in Figure 1. For example, a BGP update is received by a peering thread, placed in the *Peer queue*, processed by the label thread and written into *Labeled Queue* and finally read by the XML thread where the message is formatted into XML and placed in the *XML Queue* that is forwarded to clients. The periodic thread may also generate messages that are written into the *Labeled Queue* directly. All messages in the peer and label queues share a common BGPmon internal format shown in Figure 2.

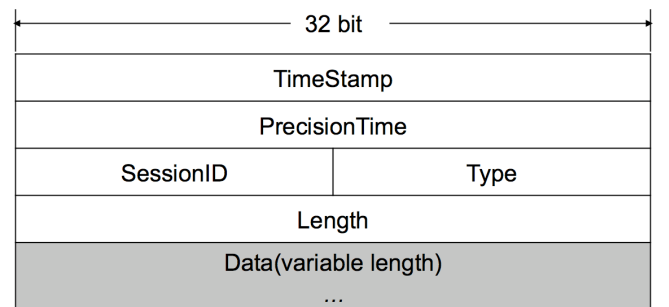


Figure 2: BGPmon Internal Format

An internal format is needed to augment the data received from (or written to) the wire. The BGP messages received from peers do not carry a timestamp and the BGP message itself does not indicate the peering

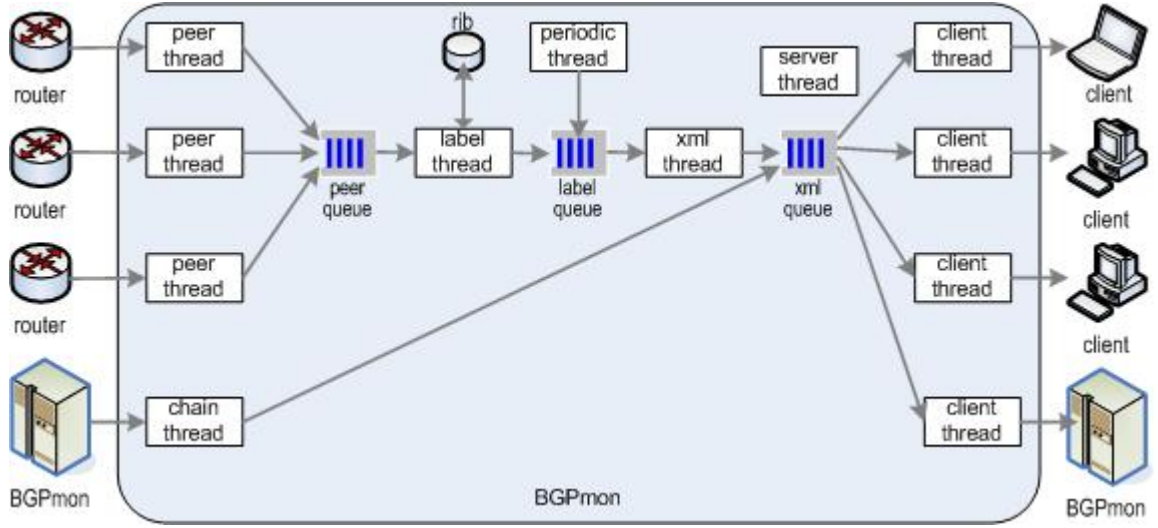


Figure 1: BGPmon Architecture.

session over which it is received. BGPmon can easily add these information by recording the time and noting the corresponding peering session when the BGP message arrives. Similarly, messages generated by BGPmon also need to have a timestamp and are typically associated with some peering sessions. Rather than directly add timestamp and peering session information as XML tags, BGPmon dened an internal message format. The use of the internal format allows internal modules to operate on xed length binary elds for e?cient computations and centralizes all XML functions into one module. Any later changes to the XML standard are now conned to the single XML module.

In BGPmon each peering session has a internal *SessionID* and all peering sessions are indexed by *SessionID*. The Labeling module can use *SessionID* to determine whether RIB tables and/or labels should be created for this peering session. The XML generation module can use the *SessionID* to determine any necessary output information, ranging from the peers address and AS number to the number of bytes sent or received over this peering session.

A small number of messages are not associated with any peering session and are assigned a *SessionID* of zero in the internal format. For example, control messages indicating BGPmon has started or is shutting down will have *SessionID* = 0.

As the name implies, the internal format is strictly internal to a BGPmon instance. Clients are never exposed to the BGPmon internal format. Messages received from other BGPmon instances arrive already formatted in XML and are placed directly into the XML queue as shown in Figure 1.

A BGPmon Internal Format message consists of six fields shown in Figure 2.

- *TimeStamp*: is a 32 bit unix time stamp indicating when the message was received (for messages from peers) or generated (for messages sent by BGPmon).
- *PrecisionTime*: is a 32 bit field indicating the number of microseconds and is used for more precise timing of messages, when the underlying operating system allows.
- *SessionID*: is a 16 bit number indicating the peering session over which the message was sent or received.
- *Type*: is a 16 bit number identifying the message type. A listed of types is given in table 3.
- *Length*: is a 32 bit number indicating the length of the data field in this message.
- *Data*: is a sequence of *Length* octets and corresponds to the message itself.

Note session ID in the message with type 6,7,9 and 10 is set to 0 as they are not specific to one peer.

2.3 The Startup and Main Program

2.3.1 Parse Command Line Arguments

The main program starts with parsing the command line arguments. Currently there are only a few simple command line arguments. First, a configuration file name can be provided using the *-c filename*. The configuration file, discussed in more detail in section 3, provides essential information ranging from the ports to listen for connections, the peers to monitor, and so

0	Reserved	Reserved for future use
1	BGPMsgSent	a BGP message generated by BGPmon and sent to a peer, generated by peering module
2	BGPMsgRcvd	a BGP message received from a peer, generated by peering module
3	BGPMsgLabeled	a BGP message received from a peer and enhanced with labels, generated by labeling module
4	BGPTableTransMsg	updates reporting the RIB-IN table as part of a route refresh, generated by periodic module
5	SessionStatus	instructs the XML module to report session status for a peer, generated by periodic module
6	QueuesStatus	instructs the XML module to report the status of all queues, generated by periodic module
7	ChainsStatus	instructs the XML module to report the status of all chains, generated by periodic module
8	FSMChange	reports a transition in a peer's finite state machine, generated by peering module
9	BGPMonStart	a status message reporting BGPmon has started, generated by configuration module
10	BGPMonEnd	a status message reporting BGPmon is shutting down, generated by configuration module

Figure 3: BGPmon Internal Message Types

forth. If no configuration file is specified, a default configuration file name specified in *site_defaults.h* will be used.

The *-r port* instructs BGPmon to start the Command Line Interface on the recovery port. The recovery port is discussed later in section 2.4 .

The remaining optional command line arguments sets the logging functions. The program can be set to run in an interactive mode that sends all messages to stdout using the *-i* option. Alternately, the program can be set to log all messages using syslog using the *-s* option. The *-i* and *-s* options are mutually exclusive and the program exits with an error if both are specified. If neither option is specified, a default value is built into the code and specified in Util/log.h.

The *-l loglevel* option specifies the log level and uses the standard syslog values as follows. Emergencies, Alerts, Critical Errors, and Errors are levels 0 to 3 (respectively). These messages are always logged regardless of the log level setting. Warnings, Notices, Information, and Debug output are levels 4 to 7 (respectively). Setting *loglevel = L* will log all messages at and below the *L*. For example, a log level of 4 will display Alerts, Critical Errors, Errors, and Warnings, but will not display Notices, Information, or Debugging output. If the *-l loglevel* option is not specified, a default logvalue is built into the code and specified in Util/log.h. **For full debug output, compile with DEBUG set to 1.**

If messages are logged to syslog, the syslog Facility can be set with the *-f facility* option. If the *-f facility* option is not specified, a default syslog facility is built into the code and specified in Util/log.h. This option has no effect if messages are written to standard output (e.g. if *-i* was specified).

2.3.2 Read Configuration File

After parsing the command line, the main program initialize the settings of all other modules by calling per-module specific initialization function. This needs to be done before reading configuration file. Then it continues to read the configuration file by calling the

facade function "readConfigFile" provided by configuration module. If the configuration file is corrupted or not readable, main program will first backup the corrupted file by calling "backupConfigFile" and then try to save as much as it can by calling "saveConfigFile". The new saved configuration file is readable and a part of corrupted one. After that, BGPmon will exit. If no configuration file is existing, main program will go ahead to start other threads and simply waits for the user to configure BGPmon via command line interface.

2.3.3 Launch and Monitor Other Threads

After reading configuration file, main program launches all other threads including peer threads, labeling thread, XML thread, periodic thread, clients management thread, chains threads, login thread and signal handling thread. As BGPmon could be running for a long time, it is likely some threads get hang or die. If one critical thread dies, the entire BGP won't work correctly. So main program also needs to make sure all the threads are still alive. Specifically, each thread keeps updating a timestamp to indicate its aliveness and main program keeps checking the timestamp of each module. If one timestamp hasn't been updated for THREAD_CHECK_INTERVAL, main thread will infer the corresponding thread died. That also requires every module needs to make sure its threads update their timestamps at least once per THREAD_CHECK_INTERVAL. BGPmon will also exit gracefully upon receipt of an interrupt signal. All signal processing and shutdown procedures are handled by the signal handling thread.

2.4 Design Philosophy

Besides the design issues mentioned above such as modular architecture and internal message format, how to configure BGPmon is another critical design issue. If we don't design it carefully and make the learning curve low, it would be very difficult to be deployed widely in the operational community. In the previous design, BGPmon used to be configured by manually editing a XML configuration file. Then we found 2 main issues of this approach:

- The configuration of BGPmon could be complex if there are hundreds of peers. That means editing configuration file manually could be tedious and error-prone.
- Learning the format of configuration file would be a barrier of using BGPmon.

In order to address these 2 issues, we decided to mimic cisco IOS configuration. Specifically, everything is configured via command line interface which is a telnet client typically. All the configuration via command line interface can be saved in a file and normal users are unaware of the existence of this file just like a cisco router. But the only difference is expert users have the ability to save multiple configuration files and switch among them in BGPmon. By doing this, the learning curve will be less steeper especially for those who are familiar with cisco IOS configuration.

As the configuration file is not supposed to be edited manually by normal users, it shouldn't be corrupted unless hardware errors, BGPmon's bugs and edition by expert users. But even now the probability of having a corrupted configuration file is low, we still need to design a failover mechanism in case of corruption. In our current design, if the configuration file gets corrupted,

- First, the corrupted configuration file is backed up.
- Secondly, BGPmon tries to save as many parts as possible into a new configuration file.
- At last, BGPmon exits to alert users the corruption of configuration file.

We now argue why those actions are necessary. If BGPmon just quits without saving a new readable configuration file, the user has to manually fix the corrupted file. With a new configuration file, the user can restart the BGPmon easily. And we don't want to just delete the corrupted file either as it is not acceptable all other peers' configuration get lost only because of the corruption of first peers' configuration. The backed up corrupted file is mainly for the expert users to figure out the problem. Our aim here is to avoid loss of configuration and manual fix in case of corruption.

Last but not least, BGPmon typically listens on a default or configured port for configuration via command line interfaces. But in the following 2 cases, the recovery port(-r port) option is needed to start a BGPmon.

- The default port is in use when BGPmon starts for the first time, specially if you don't want to change the default configuration and recompile BGPmon.
- The configured port is blocked by misconfiguring the access control. Then no one can change the access control list via command line interface as the configured port is blocked. With recovery port,

administrator could restart BGPmon with a non-blocked port and use command line interface to correct the misconfiguration.

3. CONFIGURATION MODULE

Configuration of BGPmon is entirely via command line interface which is very similar to a cisco router. Internally all the configurations will be stored in a XML file which can be loaded later. The configuration of a module corresponds to a part of the XML file. In a high level, configuration module builds a bridge between main program and all the other modules in order to facilitate the configuration management. More specifically, configuration module mainly consists of 3 parts. First configuration module provides a facade to main program that allows it to read, save and backup XML configuration file without knowing the details of other modules. Secondly configuration module provides some XML utility functions to other modules as each of them needs the same set of functions to read configuration from XML file and save configuration into XML file. At last, configuration module is also a centralized place to define the XPath's of all the configuration.

3.1 Read, Save and Backup XML Configuration File

Configuration module provides the following 3 functions to main program:

- *readConfigFile*: This facade function is called by main program to load the configuration into memory from XML file. Instead of direct reading the XML configuration file it delegates reading functions to each module. In other words, each module provides a reading configuration function to load its own configuration from XML file and the facade function just needs to call these read functions in order to load all the configuration. If the XML configuration file is corrupted, this function will try to load as much as possible into memory and return an error code.
- *saveConfigFile*: This facade function is called by main program and login module to write the configuration from memory into XML file. Similar to *readConfigFile*, it doesn't directly write the configuration into XML file. Each module provides a writing configuration function to write its own configuration into XML file and this facade function just calls them one by one.
- *backupConfigFile*: This function is called by main program to back up the current configuration file. It is called typically when the current configuration file is corrupted. We described in details how main program uses the 3 functions in section 2.

3.2 XML Utility Functions

Configuration module provides a couple of get and set functions to read and write XML file. The caller of these functions needs to pass in the XPath to locate a particular configuration item. These get functions can return the configuration item in a specified data type and check the value against the specified conditions. For example, the caller can specify to get a configuration item in integer and check if the value is between 0 and 10. If this configuration item in XML file cannot be converted to a integer or its value is larger than 10, a error code will be returned.

3.3 XPath Definitions

Each module needs to define a bunch of XPaths in order to read its own configurations from XML file. For example, for the clients management module its configurations look like this:

```
<BGPmon>
  <CLIENTS>
    <LISTEN_ADDR>ipv4loopback</LISTEN_ADDR>
    <LISTEN_PORT>50001</LISTEN_PORT>
    <ENABLED>0</ENABLED>
    <MAX_CLIENTS>10000</MAX_CLIENTS>
  </CLIENTS>
</BGPmon>
```

As a result, clients management module needs to define 4 XPaths for the 4 items: LISTEN_ADDR, LISTEN_PORT, ENABLED and MAX_CLIENTS. In order to get the 4 values, clients management module needs to call the get functions mentioned before and pass in the XPaths. The XPath definitions of all modules can be found in Configconfigdefaults.h.

3.4 Design Philosophy

As each module has the best knowledge of its own configuration, our design divides the entire configuration of BGPmon into a couple of small pieces according to the division of modules. Each module only handles its own piece. In this way, the changes of configuration will be localized inside one module and none of them will be exposed to main program or other modules. XML utility functions are defined here as most of modules need them to handle the XML file. Also in order to manage all the XPaths of modules efficiently, they are centralized stored in the configuration module. The last design issue here is about default configuration. There are 2 reasons why we need this default configuration.

- It includes the minimal set of configuration to start BGPmon for the first time. For example, the command line interface needs a default enable password and a default port to listen on even if there is no configuration yet.

- It provides the defaults for all the optional configuration. For example, the BGP version of peer configuration is optional and the default value will be used if it is not specified by the user.

The default configuration of BGPmon can be found in site defaults.h. It can be changed by editing this file and then recompile BGPmon. And default configuration will be overwritten by the configuration via command line interface.

4. PEERING MODULE

Peering module manages the configuration for all peers and maintains peering sessions for enabled peers. Every peer has its own configuration that can be changed via command line interface. But only enabled peer will have one and only one peering session that is established by using the peer's configuration. If the peer's configuration changes after its peering session gets established, some of the new changes will not be applied to the established peering session immediately. For instance, the changes of monitor side address and port cannot be applied to the existing peering session. This kind of changes can only take effect by closing the existing session and opening a new session.

Each peering session is a separate thread that basically maintains a BGP finite state machine such as initialize a tcp connection, exchange BGP open and keepalive messages with the peer and receives BGP update messages from the peer. It also writes all messages between BGPmon and the peer into the Peer Queue. There are 3 types of messages that can be added into Peer Queue: messages from peer (BMF type 2), messages to peer (BMF type 1) and FSM state changes (BMF type 6).

The details of peer configuration and peering session are discussed below.

4.1 Peer Configuration

Similar to BGP configuration in Cisco IOS, we use peer group to simplify the peer configuration in BGPmon. If a number of peers share a common set of configuration, peer group can simplify configuration greatly. With a peer group that has the common set of configuration, to add a new peer one only needs to assign it to the existing peer group and specify a few fields if needed. Those specified fields will overwrite the same fields from the peer group. But all the other fields from the peer group will be inherited by the peer.

Different from Cisco IOS, every peer must be associated with a peer group in BGPmon. If one doesn't specify the peer group for a peer explicitly, this peer will be assigned to the default peer group that holds the default configuration. The default peer group is created when BGPmon starts. In BGPmon, every user-created peer group is also inherited from the default peer group.

by default and it cannot be changed. The fields specified in the user-created peer group overwrites those from default peer group.

In detail, peering module maintains 2 arrays: one array stores all peers and another stores all peer groups. Each peer in the first array holds a reference to a peer group in the second array. The default peer group is always created at first in the peer group array as it will be referred by any user-created peer group. Figure 4 shows the relationship between peers and peer groups. In the example, there are 3 peers and 3 peer groups including the default peer group. The arrows indicate the relationship between them. Peer1 has its own values for configure item A and B, so those values overwrite the values in its peer group "Group2". As a net result, PA1 and PB1 will be the final values for peer1. Peer2 belongs to default peer group directly and it doesn't have its own value for item B, so peer2 inherits the value of item B from default peer group and has PA2 and DefaultB as its final values. Similar peer 3 doesn't have its own value for item A and its peer group(Group1) doesn't have the value either, so peers inherit the value of item A from default peer group. Finally peer3 has DefaultA and PB3 as its configure values.

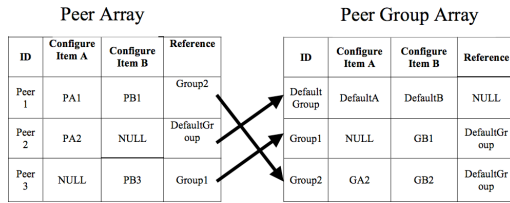


Figure 4: An example of Peers and Peer Groups

From the above example, you can see the structures of peers and peer groups share the same set of configure items. In our design, peer structure and peer group structure share the same substructure "configuration" that includes all the configure items. Specifically peer structure consists of three parts:

- *Peer ID*: It is the identifier of a peer, starting from 0. It is also the index of a peer in the array.
- *Session ID*: It is the identifier of a peering session that is associated with a peer, starting from 0. For the disabled peer, it is -1. Peering session will be discussed in the next subsection.
- *Configuration Substructure*: It contains all configure items needed in peer configuration.

And peer group structure also consists of three parts:

- *Peer Group ID*: It is the identifier of a peer group, starting from 0. It is also the index of a peer group in the array.

- *Peer Group Name*: It is the name of a peer group.
- *Configuration Substructure*: It is same as the configuration substructure in peer structure.

Configuration substructure includes all the configure items which are needed by peering session establishment, labeling module and periodic event handling module. Figure 5 shows the details of configuration substructure. All of these configure items except "routerRefreshAction" and "labelAction" are used to establish a peering session. "routerRefreshAction" is used by periodic event handling module and "labelAction" is used by labeling module.

Peering module also provides a bunch of functions to create, delete, read and write peers and peer groups. Command line interface uses these functions to manipulate peer configuration. We have discussed the peer configuration so far. Next peering session will be discussed.

4.2 Peering Session

Peering module maintains an array that holds the data for all peering sessions and each element in this array is a "Session" structure. "Session" structure is not only used by peering module but also used by labeling module, periodic event handling module and XML module. As we mentioned each enabled peer has its own thread and these threads create "Session" structures in the array and uses them to establish and maintain the peering sessions. In each peer's thread, a peering session gets established by using the latest configuration of that peer. When a peering session is established, all the configure items that are used to establish the session will be saved in the "Session" structure. After that, any changes in those configure items via command line interface will not take effect until the peering session resets. And the peering session could be reset in the following 2 cases.

- Users reset the peering session explicitly by issuing a reset command via command line interface. This happens typically when users change some peer configuration via command line interface and want these changes to be applied immediately.
- Peering session resets by itself. For example, the underlying tcp connection failure could cause a peer session reset. Or the peering session could be reset if the peer fails for some reason.

No matter why the peering session is reset, the latest peer configuration will be used when it gets established again. An important design decision here is that a new "Session" structure with new session ID will be created and used by the thread every time a peering session is reset. Note even the thread is using the new "Session" structure, the old one will still stay in the session array

Name	Structure	Structure Definition		
		Field Name	Field Type	Description
localSettings	LocalSettingStruct	addr	char[]	Monitor side Address
		port	int	Monitor side port
		AS2	int	AS number in the sending open message
		BGPVer	int	BGP version in the sending open message
		BGPID	long long	BGP ID in the sending open message
		holdtime	int	Hold time in the sending open message
remoteSettings	RemoteSettingsStruct	addr	char[]	Peer address
		port	int	Peer port
		md5_passwd	char[]	Peer's md5 password
		AS2	int	Check value of AS number in the receiving open message. 0 means accept any value.
		BGPVer	int	Check value of BGP version in the receiving open message. 0 means accept any value.
		BGPID	int	Check value of BGP ID in the receiving open message. 0 means accept any value.
		minHoldTime	int	min value of hold time in the receiving open message. 0 means accept any value.
receiveCapRequirements	An Array of PeerCapabilityRequirement Struct	code	u_char	BGP capability code
		action	int	0 = Allow 1 = Require 2 = Refuse
		checkValueLen	int	Length of the check value
		checkValue	u_char []	check value in octets
		flag	int	facilitate the checking of capabilities
announceCaps	An Array of pointers to BGPCapabilityStruct	code	u_char	BGP capability code
		length	u_char	Length of capability value
		value	u_char []	capability value in octets
routeRefreshAction	int	0:disabled, 1:enabled, Used by periodic event handling module		
labelAction	int	0:none, 1:label, 2:storerib, Used by labeling module		
enabled	int	0:disabled, 1:enabled		
groupName	char []	Peer group name in string		
groupID	int	Index of the peer group in peer-group array		

Figure 5: Configuration Substructure

for a while until it is not needed. We will discuss the design philosophy behind this in the next subsection. In the remaining part of this subsection, the detail of "Session" structure will be discussed at first and then an introduction of how to establish and maintain a peering session by using "Session" structure will be given.

4.2.1 "Session" Structure

Most fields of a "Session" structure are related to the peering module and a few fields are related to other modules. The details are shown as follows:

- *sessionID*: is the identification of a peering session which starts from 0. It is also the index of a peering session in the array.
- *ConfigInUse substructure*: is same as the configuration substructure shown in Figure 5. It is basically a copy of peer configuration when peering session gets established. It will not be changed once the peering session gets established. This substructure is important if one wants to know what are the differences between the latest peer configuration and the configuration used to establish the peering session.
- *FSM substructure*: is a group of fields that are used to maintain the BGP Finite State Machine (FSM) such as state of FSM, socket and a couple of timers. Figure 6 shows the details of FSM substructure.
- *Statistics substructure*: is a group of fields related to the peer's statistics such as the time of last session reset and the number of received updates. Figure 7 shows the details of statistics substructure
- *peerQueueWriter*: is used to write the messages exchanged between BGPmon and the peer into the queue.
- *sessionStringOutgoing*: is a XML string generated by peering module. It is used by XML module as the common XML header of all outgoing messages. It consists of 2 triples: Source triple and Destination triple. Source triple contains monitor side address port and AS number. Destination triple contains peer's address, port and AS number.
- *sessionStringIncoming*: is similar to sessionStringOutgoing. It is a common XML header for all incoming messages. In this XML string, source triple contains peer address port and AS number. Destination triple contains monitor side address, port and AS number.
- *PrefixTable Substructure and AttrTable Substructure*: are used to maintain a RIB table. They

are initialized by peering module and populated by labeling module. The detail of them will be discussed in Section 6.

- *reconnectFlag*: is used to reset a peering session and set by command line interface. This flag is typically set when a user wants the changes of peer configuration to be applied.
- *lastAction*: is a timestamp that indicates when the last action of a peering session is. It is used to check if a peering session is running correctly.

4.2.2 Establish and Maintain Peering Sessions

To establish a BGP peering session, the first step in peer's thread is to create a socket and bind it to the monitor side address and a port specified in "localSettings" of the peer configuration(See Figure 5). Secondly, the thread needs to initiate a tcp connection to the peer actively. Similarly, the peer's address and port are specified in "remoteSettings" of the peer configuration. Note in our design the thread always initiates the connection actively and simply drops all the incoming connection from the peer for the security purpose.

Once the tcp connection is established, the thread will exchange the BGP open message with the peer. As shown in Figure 5, all the parameters(BGP version, BGP ID, AS number and holdtime) needed to create a BGP open message can be found in "localSettings" of configuration substructure. Also all announcing capabilities included in the open message can be found in "announceCaps" of configuration substructure. Then the thread sends the created open message via the established tcp connection and waits for the incoming open message from the peer.

Upon receiving the open message, the thread will do the following two checks.

- Check the version, AS number, Identifier and holdtime in the received open message against the expected values specified in "remoteSetting" of configuration substructure(See Figure 5).
- Check the capabilities in the received open message against the capability requirements specified in "receiveCapRequirements" of the configuration substructure(Figure 5). For each capability *A* in capability requirements, there are three possible actions:
 - *Allow*: nothing needs to be done.
 - *Require*: check if the received capabilities contain *A* and the value of the received capability is same as the check value is configured. If no, check fails.
 - *Refuse*: check if the received capabilities contain *A* and the value of the received capability

Field Name	Type	Description
socket	int	the socket used to communicate with the peer
state	int	the current state of FSM: 1 = stateIdle; 2 = stateConnect 3 = stateActive 4 = stateOpenSent 5 = stateOpenConfirm 6 = stateEstablished 7 = stateStopped
connectRetryInt	int	tcp connection retry interval
connectRetryTimer	time_t	when is the next time to retry to connect
keepaliveInt	int	keepalive message sending interval
keepaliveTimer	time_t	when is the next time to send keepalive message
holdTime	int	the maximum number of seconds that may elapse between the receipt of successive messages from the peer
holdTimer	time_t	if no messages are received by this time, the monitor will assume the peer is down.
routeRefreshType	int	which type of router refresh is supported by the peer? 2 = standard route refresh 128 = cisco route refresh
routeRefreshFlag	int	this flag is set by periodic event handling module. If this flag is 1, the peering thread needs to send a route refresh request to the peer.
peerCapabilities	An Array of PeerCapabilityRequirement Struct	all the received capabilities from the peer

Figure 6: FSM Substructure

Field Name	Type	Description
connectRetryCounter	int	the number of times to retry a tcp connection
SessionDownCounter	int	the number of times BGP session goes down
lastDownTime	time_t	last BGP session down time
messageRcvd	int	the number of received BGP messages
establishTime	time_t	BGP session establishment time
lastRouteRefresh	time_t	last route refresh sending time
NannRcvd	int	the number of 'new announcement' updates
LastHourNannRcvd	int	the number of 'new announcement' updates during the last hour
DannRcvd	int	the number of 'duplicate announcement' updates
LastHourDannRcvd	int	the number of 'duplicate announcement' updates during the last hour
SpathRcvd	int	the number of 'same path' updates
LastHourSpathRcvd	int	the number of 'same path' updates during the last hour
DpathRcvd	int	the number of 'different path' updates
LastHourDpathRcvd	int	the number of 'different path' updates during the last hour
WithRcvd	int	the number of 'withdraw' updates
LastHourWithRcvd	int	the number of 'withdraw' updates during the last hour
DuwiRcvd	int	the number of 'duplicate withdraw' updates
LastHourDuwiRcvd	int	the number of 'duplicate withdraw' updates during the last hour
memoryUsed	long	the number of bytes used to store RIB-IN table
prefixCounter	int	the number of prefixes in RIB-IN table
attrCounter	int	the number of unique attributes sets in RIB-IN table

Figure 7: Statistics Substructure

is same as the check value is configured. If yes, check fails.

If any of these checks fails, the thread will send a notification message to the peer and close the connection. If all the checks pass, the thread will send a keepalive message to the peer and wait for another keepalive message from the peer. Once this keepalive message is received, the BGP peering session is successfully established.

After the peering session gets established, the thread will periodically sent out keepalive messages if holdtime is not zero and route refresh requests if configured.

Each thread also writes two types of messages into the peer queue:

- **BGP Message:** The BGP messages exchanged with the peer are written into the peer queue. Note no 'Update' messages will be sent from BGPmon.
- **FSM Message:** The state changes of BGP finite state machine are written into the peer queue such as from 'Idle' to 'Connect', from 'OpenConfirm' to 'Established' and so on.

These types of messages are converted into BGPmon internal format before being written into peer queue. After conversion, the session ID in the message indicates it belongs to which peering session.

In our design, periodic event handling module centralized manages and schedules the route refresh for all the peers. Instead of sending route refresh requests by itself, periodic event handling module notifies the peering module to send route refresh requests when needed. And the field 'routeRefreshFlag' in FMS substructure(Figure 6) is used to notify peering module.

4.3 Design Philosophy

In the design of peering module, one important issue is how to handle changes in a peer's configuration when the peer already has a existing peering session. Most of configuration changes will only take effect by resetting the existing peering session such as changes in monitor address, port or AS number. But it is not practical to reset a peering session every time such a change happens. Probably one might want to reset the peering session after a series of changes is done. So we decided to let the user make the decision about when to reset the peering session. User can reset the peering session by issuing a command via command line interface.

As a result of letting user make the decision when to make configuration changes take effect, it is necessary to provide the user with the difference between the latest peer configuration and the configuration used to establish the existing peering session. This explains why we need a "ConfigInUse" substructure in a "Session" structure. The "ConfigInUse" substructure is copied from peer configuration when peering session is established

and will not be changed after that. By comparing the "ConfigInUse" of a peering session and the latest peer configuration, the user will know all the changes of a peer configuration since its peering session was established.

Another important design issue is that a new "Session" structure with new session ID will be created and used by the thread every time a peering session is reset. The reason is related to the BGPmon architecture and the format of BMF message. As we discussed, modules of BGPmon are connected by queues and every message in peer queue and label queue has a session ID field. The session ID will be used as the key to retrieve the information needed to process the BMF messages by other modules such as labeling module and XML module. For example, XML module needs the session ID to get the XML common header("sessionStringOutgoing" and "sessionStringOutgoing" in a "Session" Structure) in order to convert a BMF message to a XML string. In BGPmon it is possible that some BMF messages associated with old peering session are buffered in the queue after a peering session gets reset. In this case if the peer's thread doesn't create a new "Session" structure and simply uses the same "Session" structure, XML module will wrongly convert the BMF messages associated with old peering session by using the XML common header of new peering session. In order to avoid this, the peer's thread needs to create a new "Session" structure and keep the old "Session" structure when a peering session is reset. After peering session reset, the peer's thread will use the new "Session" structure and tag the new BMF messages with new session ID. The old "Session" structure will finally be deleted after all the BMF message associated with the old peering session are processed.

5. MRT MODULE

This section describes MRT module.

5.1 MRT overview

Multi-threaded Routing Toolkit (MRT) format was developed to encapsulate, export, and archive routing information in a standardized data representation. The BGP routing protocol, in particular, has been the subject of extensive study and analysis which has been significantly aided by the availability of the MRT format.

BGPmon MRT module gives an opportunity to receive data (updates and RIBs) from Routing Collectors (RC) such as University of Oregon Route Views Project[6], RIPE NCC RIS Project[5] and others.

MRT control module consists of a single MRT server thread and multiple MRT client threads.

- *MRT Server Thread:* It is a TCP server that listens on a specific port and spawns one thread for each client.

- *MRT Client Thread*: Each client thread receives data from a TCP connection.

5.2 MRT Server Thread

The main data structure of MRT control module consists of the following fields:

- *listenAddr*: The listening socket of server thread binds to this address. It is a string which could be a IPv4/IPv6 address or one of four keywords (ipv4loopback, ipv4any, ipv6loopback and ipv6any). After it is initialized from configuration, it could be set via command line interface (CLI) at runtime.
- *listenPort*: It is the port on which server thread listens. It is an integer. It also could be set via command line interface (CLI) at runtime after initialization.
- *enabled*: It indicates the status of server thread. If it is false, server thread stops listening but the existing clients still run. Otherwise server thread listens on 'listenPort' and accepts allowed clients. It could be set via CLI after initialization.
- *maxMRTclients*: It is the max number of MRT clients. It could be set via CLI after initialization.
- *labelAction*: It is the label action of MRT clients
- *activeMRTclients*: It is the number of connected MRT clients.
- *nextMRTclientID*: It is the ID for the next client to connect.
- *rebindFlag*: If listenAddr or listenPort changes, this flag will be set to TRUE. That means the listening socket of server thread will bind to the new address or port. It is set by CLI.
- *shutdown*: It is a flag to indicate whether to stop the server thread.
- *lastAction*: It is a timestamp to indicate the last time the thread was active.
- *MRTListenerThread*: Reference to MRT thread.
- *firstNode*: First node in list of active MRT clients
- *MRTLock*: It is a pthread mutex lock used to lock the MRT info linked list when MRT clients are added or deleted.

This structure is mainly maintained by server thread.

5.3 MRT Client Thread

MRT Client Thread has:

- *id*: Identification number of MRT client.
- *addr*: MRT Client's IP address.
- *port*: MRT Clients's Port number.
- *socket*: MRT Client's socket for reading data
- *connectedTime*: MRT Client's connected time in seconds.
- *lastAction*: MRT Client's last action timestamp.
- *qWriter*: This is a queue writer (see section 11). It is used to write messages to Peering queue.
- *deleteMRTClient*: Flag to indicate to close the MRT Client thread
- *labelAction*: Default label action.
- *next*: Pointer to next MRT Client node.
- *MRTThreadID*: Thread reference.

5.4 MRT Module Peering Design

5.4.1 Existing/3rd Party Routing Collectors

This subsection briefly describes basic functions of Routing Collectors(RC) and required software changes for MRT format support. Also, we describe few important functions of BGPmon MRT module.

Although BGPmon able to peer directly with routers, there might be existing Routing Collectors in the Internet. BGPmon is able to work with these external RCs with a few modifications to the RC software. To enable sending routing data to BGPmon, already existing Routing Collector's software should support MRT format[4]. This patch adds new functions to Routing Collector to send MRT messages via TCP connection. Figure 8 shows the overall topology, where Routing Collector has multiple TCP sessions to Router 1-4 and BGPmon.

Every MRT message has a MRT header and body structure. Header in update and RIB-IN messages has the same structure: Timestamp (time in seconds, when message originated), Type (type of MRT message), Sub-type, Length (length of MRT message without the header) and BGP message[3]. MRT body starts with peer's and RC's IP address, port number and AS number fields. MRT body in update message contains BGP message originated by peer, while MRT body in RIB-IN message has all internal routes generated by RC.

Routing Collector maintains its own peer sessions via TCP connection. For example, if TCP connection is lost to peer, Routing Collector should erase all BGP attributes and prefixes associated with peer id.

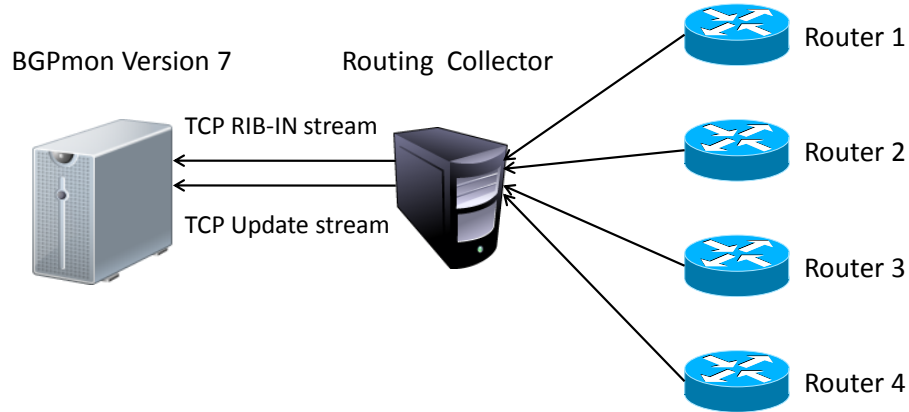


Figure 8: BGPmon and Routing Collector

If peer reestablish connection, Routing Collector will create new session structure. In our case, BGPmon MRT module is using MRT message fields (peer IP address, port number and AS number fields) for maintaining "Session ID" structure described in section 4. The details of session management logic will be discussed in section 5.4.5 and 5.4.6.

In our design, Routing Collector can create multiple TCP connections to MRT module. Once the TCP connection is established, Routing Collector could send snapshot of MRT RIB-IN messages or new MRT update messages. This duality raises important questions:

- What if first TCP stream contain MRT update messages and second TCP stream contain MRT RIB-IN messages?
- What if first TCP stream contain RIB-IN messages and second TCP stream contain MRT update messages?

Answer for the first question is simple: MRT module will process update and RIB-IN messages consequentially: create "Session" structure, extract BGP messages with attributes and prefixes from MRT messages, create new BMF message send it to PeerQueue (see sections 2 and 11).

Answer for the second question is different: MRT module will copy all MRT RIB-IN message to queue

buffer. Then, for each of update message, which contain Peer IP address, port number and AS number, MRT module will search in queue buffer to same values. If these values are found in queue buffer, MRT module will extract BGP attributes and prefixes from buffer, create new BMF messages and send them to PeerQueue. Then MRT module will convert update message to BGPmon's internal BMF format and apply this message to PeerQueue. Overall, second question describes the ideal behavior of Routing Collector: first of all, its necessary to send correct picture of routing data (RIB-IN messages). Then Routing Collector should periodically send MRT update messages to change (add or withdraw) BGPmon's internal RIB-IN table.

Also, it's important to mention another issue in this design: what should be a time period (timer) between two connections? For example, in situation described in first question, if time value is way too long, we could damage already created BGPmon's routing table. If this value is too small, two connections could overlap and create a routing mess. In our design, we assume that 5 to 10 minutes period is fine to use between connections and this value won't change BGPmon's routing table dramatically.

Another significant design issue is handling "Session" structure failure. Routing Collector maintains TCP ses-

sion with its peers and MRT message format does not allow us to know, when TCP connection fails or closes between Routing Collector and peer. In our design, instead of adding and keep tracking timeout value for each of created "SessionID's", Routing Collector sends RIB-IN message with empty routing table for peer. At BGPmon side, MRT module will erase all BGP attributes and prefixes associated with this peer. In case when Routing Collector reestablish connection with a peer, it sends newly created MRT update and RIB-IN messages so MRT module will create new "Session" structure with BGP attributes and prefixes.

In case if Routing Collectors sends entire MRT RIB-IN messages periodically, BGPmon MRT module will reset all routing tables of "SessionID's" associated with Routing Collector's data and create new "Session" structures.

5.4.2 BGPmon MRT module overview

BGPmon MRT module supports 2 types of MRT messages (see [4]):

- *Update message*: This message has a header structure with "BGP4MP" type and "BGP4MP_MESSAGE" or "BGP4MP_MESSAGE_AS4" subtype and message body with BGP attributes and prefixes.
- *RIB-IN message*: This messages has header structure with "TABLE_DUMP_V2" type and contains all routing data from Routing Collector (RC).

Messages with other MRT header types (for example OSPF, ISIS) are ignored and error message will appear in log file or stdio output.

5.4.3 Processing MRT updates

MRT update message, received from Routing Collector, has following fields:

- *Peer address*: Peer IPv4 or IPv6 address
- *Local address*: RC IPv4 or IPv6 address
- *Peer AS*: Peer AS number, could be 2-bytes or 4-bytes length
- *Local AS*: RC AS number, could be 2-bytes or 4-bytes length
- *BGP Message*: BGP Message with attributes and prefixes

MRT module parses update message following MRT draft specification and converts it to internal BMF BGPmon message. Then new BMF message applied to PeerQueue.

5.4.4 Processing MRT RIB-IN tables

Routing Collector sends RIB-IN table message only once after successful TCP connection. RIB-IN messages

have MRT header structure with "TABLE_DUMP_V2" type and current timestamp. Main body message contains list of all peer clients connected to Routing Collector and their BGP attributes and prefixes. BGPmon MRT module parses this message, creates and sends new BMF message to PeerQueue.

5.4.5 Session Management

When Routing Collector establishes TCP connection to BGPmon and new messages (update or RIB-IN table) arrives, MRT module makes the following checks based on first arrived message:

- *First message is RIB-IN message*: MRT module creates RIB-IN queue buffer and copy all RIB-IN messages to the buffer.
- *First message is update message*: MRT module creates new Session structure with sessionID (based on Peer IP address, port and AS number), searches through the RIB-IN queue buffer for BGP attributes and prefixes associated with sessionID. Then MRT module creates new BMF messages and sends them to PeerQueue.

5.4.6 Session Closing

RC's TCP connection sends multiple update messages from its peers, MRT module keeps track of all SessionIDs created with particular RC. If RC's connection fails or closes, MRT module will:

- *Change Label Action*: Change all states of previously created Session IDs to "stateError".
- *Delete Attributes*: Delete all BGP attributes and prefixes from BGPmon internal RIB-IN table.

5.4.7 2 bytes and 4 bytes AS length format

BGPmon MRT module supports update and RIB-IN messages containing 2-bytes or 4-bytes length AS Path. Current software implementation of Routing Collectors could send updates or RIB-IN messages using different format. Based on received data, MRT module makes following changes to ASPath associated with SessionID structure:

- *First message is RIB-IN and has 2 byte ASN*
 - *Second message is update and has 2 bytes ASN*: MRT module will process this update message without any changes
 - *Second message is update and has 4 bytes ASN*: MRT module will print a error and wont process this message
- *First message is RIB-IN and has 4 bytes ASN*
 - *Second message is update and has 2 bytes ASN*: MRT module will convert internal BGPmon routing table to 2 byte ASPath length

- *Second message is update and has 4 bytes ASN:* MRT module will process this update message without any changes
- *First message is update and has 2 bytes ASN*
 - *Second message is RIB-IN and has 2 bytes ASN:* MRT module will process this update message without any changes
 - *Second message is RIB-IN and has 4 bytes ASN:* MRT module will convert ASpath of RIB-IN message to 2 byte ASN
- *First message is update and has 4 bytes ASN*
 - *Second message is RIB-IN and has 2 bytes ASN:* MRT module will print error and wont process this message
 - *Second message is RIB-IN and has 4 bytes ASN:* MRT module will process this update message without any changes

5.5 Design Philosophy

The important design decision here is that BGPmon MRT module can receive data from any kind of Routing Collector which supports MRT format specification. For example routing software such as Zebra or Quagga needs a small patch to support MRT format. This patch is available in BGPmon distribution.

In the design of MRT module, one of important issues is how to handle order of update or RIB-IN messages with 2 or 4 bytes AS Path length. Today's peering routers rarely use 4-bytes AS Path length and they trying to avoid problem with 4 to 2 bytes ASPath conversion. MRT module is using simple solution described in previous subsection.

6. LABELING MODULE

Labeling module manages one RIB-IN table associated with a peer if configured and uses the tables to assign labels to updates received from the peer. In this module, the only configuration is about how to process the BGP updates from peers. It is specified by "labelAction" in peer configuration as shown in Figure 5. "labelAction" could be one of these three options:

- *None:* Don't process the updates at all.
- *RibStore:* Store the updates in RIB-IN tables on a per-peer basis. The peer has its own RIB-IN table if this option is set.
- *Label:* Store the updates in RIB-IN tables and label the updates based on how they change RIB-IN tables. This option implicitly stores RIB-IN for the peer.

Since the RIB-IN tables are the major memory consumption of BGPmon, one might want to set "labelAction" as "None" if the memory is the major concern.

Labeling module has a single thread which is a reader of peer queue and a writer of label queue as shown in Figure 1. Main flow has three steps:

- Read the BMF messages from peer queue
- If the BMF message is a update, then process it based on the "labelAction" configuration. Otherwise, do nothing.
- Write the processed BMF messages into labeling queue

The detail of main flow logic will be discussed in section 6.2.

6.1 Data Structure

Labeling Module uses 2 main data structures(PrefixTable and AttrTable) to maintain the RIB-IN table. As the RIB-IN table is maintained on a per-peer basis, it is natural to make these 2 structures as components of the "Session" structure as we mentioned in section 4.2.1. "PrefixTable" and "AttrTable" are used to store prefixes and attributes in BGP updates respectively. And these 2 tables are inter-linked to make up one RIB-IN table.

6.1.1 PrefixTable Structure

In our design it is a hash table that consists of multiple entries. Each entry is a link list of nodes and each node contains a prefix. It has the following six parts.

- *prefixCount:* is the number of prefixes are contained in the prefix hash table.
- *tableSize:* is the number of entries in the hash table.
- *occupiedSize:* is the number of occupied entries in the hash table. Occupied entry means it has at least one node.
- *maxNodeCount:* is the max length among all entries in the hash table. The length of one entry indicates how many nodes it contains.
- *maxCollision:* is the max number of nodes one entry is allowed to contain. If one entry contains too many nodes, we need to increase the number of entries in a hash table in order to improve the performance. Basically if maxNodeCount is larger than maxCollision, we need to enlarge the hash table.
- *emphprefixEntries:* is an array of PrefixEntry structures. It contains all the entries of the prefix hash table. For the details of PrefixEntry structure, see Figure 9.

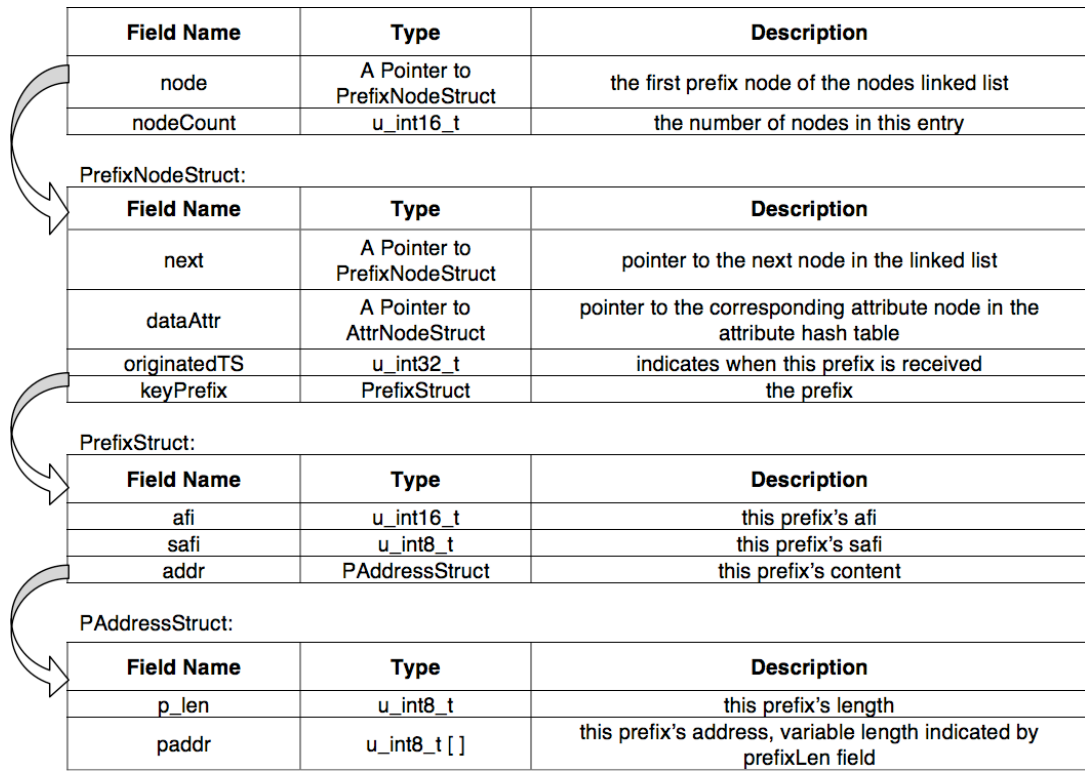


Figure 9: PrefixEntry Structure

Each prefix in the NLRI of a BGP update will be one node of a particular entry in the prefix table. The index of this entry is calculated by hash value of the prefix. 2 same prefixes will be hashed to the same entry and stored in the same node. Because of hash collision, 2 different prefixes could also be hashed to the same entry. But they will be stored in difference nodes of this entry.

6.1.2 AttributeTable Structure

Attribute hash table is similar to the prefix hash table mentioned in the previous section. AttributeTable structure is used to implement a hash table to store the attributes of BGP updates. It has the following six parts.

- *attributeCount*: is the number of attributes are contained in the attribute hash table.
- *tableSize*: is the number of entries in the hash table.
- *occupiedSize*: is the number of occupied entries in the hash table.
- *maxNodeCount*: is the max length of all entries in the hash table.
- *maxCollision*: is the max number of nodes one entry is allowed to contain.

- *attrEntries*: is an array of AttrEntry structures. For the details of AttrEntry structure, see Figure 10.

In the attribute table, the attribute set in a BGP update will be stored in one node. And which entry this node belongs to is determined only by the hash value of AS path. That means if 2 attribute sets have the same AS path, they will be hashed to the same entry. If other attributes of the 2 sets are also same, they will be stored in the same node. If 2 attribute sets have the same AS path but other attributes are different, they will still be stored in the same entry but 2 different nodes. Note in the case the AS path is only stored once and the 2 different nodes will link to the same AS path. Again it is possible that 2 attribute sets with different AS paths are hashed to the same entry because of hash collision.

6.2 Main Flow Logic

After introducing the data structure, we describe the main flow logic of labeling module here. As we mentioned, labeling module reads the BMF messages from peer queue. For the BMF messages with type other than 2 (see Figure 3), the labeling module simply writes them into labeling queue without any processing. For each BMF message with type 2, labeling module processes it as follows:

- If it doesn't contain a BGP update message, sim-

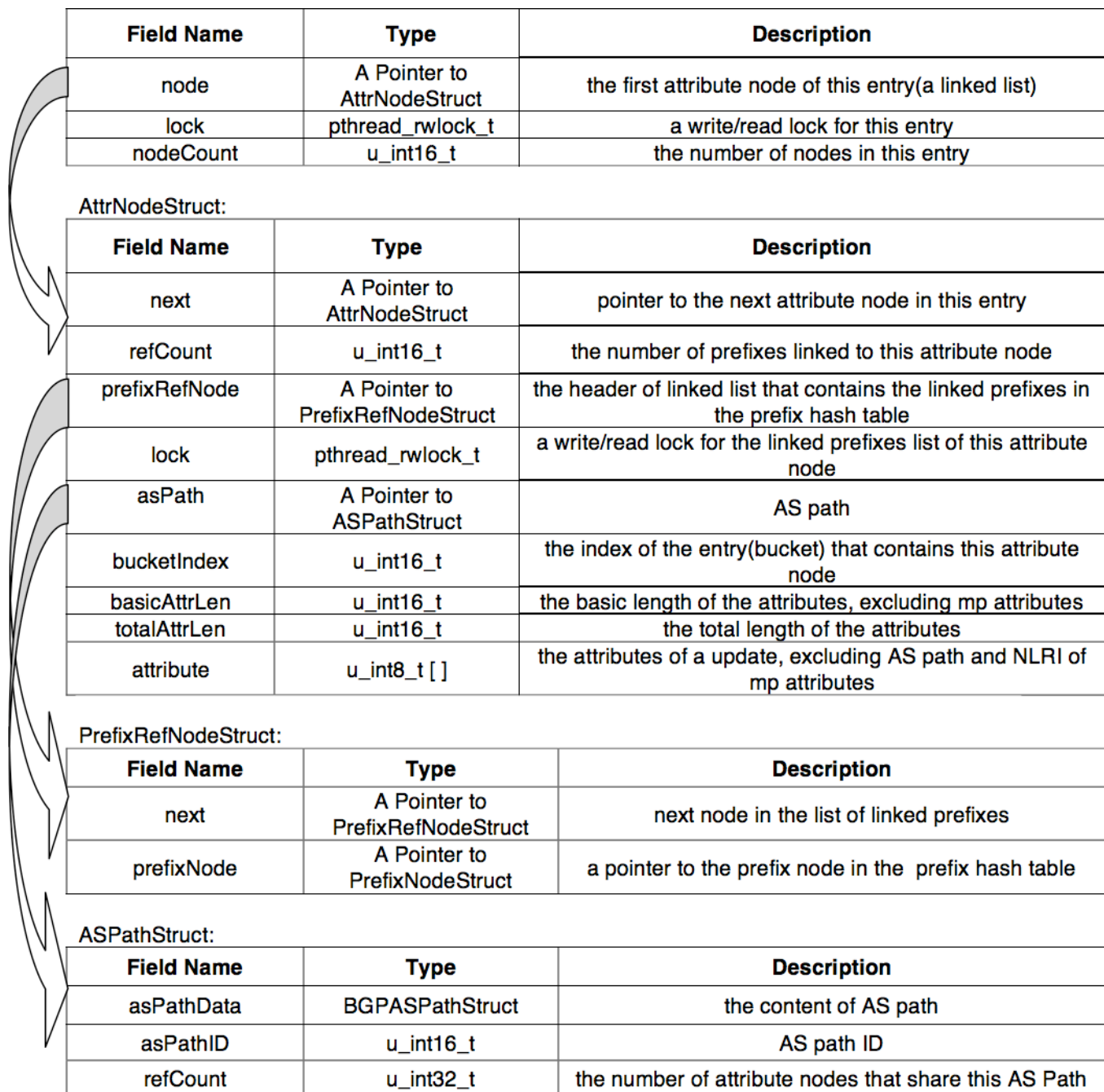


Figure 10: AttributeEntry Structure

ply writes it into labeling queue without any processing.

- If it contains a BGP update message, extract the session ID from the BMF message. With the sessionID the labeling module finds the peering session structure and checks the 'labelAction' in it (see Figure 5).
 - If the field 'labelAction' is set to "None", simply writes the BMF message into label queue without any processing.
 - If the field 'labelAction' is set to "RibStore", updates the RIB-IN table of the corresponding peering session and then writes the the BMF message into label queue. In section 6.3 we will discuss the detail about how to update RIB-IN tables given a BGP update message.
 - If the field 'labelAction' is set to "label", updates the RIB-IN table of the corresponding peering session and labels this BMF message. Specifically, one label for each prefix in NLRI will be attached to the BMF message and its message type will be changed to 3. Finally the new BMF message with type 3 will be written into label queue. In section 6.4 we will discuss the detail about how to label a BGP update message.

6.3 Store RIB-IN Tables

The labeling module stores RIB-IN tables on a per-peer basis. For each incoming BGP update message from a peer, the labeling module stores it in the peer's RIB-IN table as follows:

1. Parse the BGP update message into the following components:
 - IPv4 unicast reachable NLRI and unreach NLRI
 - multiple protocol(mp) reachable NLRI and unreach NLRI
 - the attribute set excluding AS path, mp unreachable attributes and NLRI of mp reachable attributes
 - AS path
2. Hash the AS path and find the entry in the attribute table based on the hash value. Then check each node of this entry against the AS path and attribute set from above as follows:
 - If the current node's AS path and attribute set are the same, return this attribute node.
 - If the current node's AS path is same but attribute set is not, return a new created attribute node that has the new attribute set

and is linked to the existing AS path. Recall the same AS path shared by multiple nodes is only stored once.

3. If none of the nodes matches the above 2 rules, return a new created attribute node with the new AS path and attribute set.
4. Extract all prefixes from IPv4 unicast reachable NLRI and mp reachable NLRI. Each prefix is processed as follows:
 - If there is a existing prefix node which contains the same prefix content such as such as afi, safi, mask length and address in the prefix hash table:
 - If the existing prefix node's linked attribute node('dataAttr' field in Figure9) is same as the attribute node returned from step 2, do nothing.
 - Otherwise:
 - * Delete this prefix node from its current attribute node's linked prefix nodes list. If there isn't any prefix node linked to this attribute node after this deletion, remove this attribute node from the attribute hash table.
 - * Add this prefix node to the linked prefix nodes list in the attribute node returned from step 2.
 - Otherwise:
 - Create a prefix node(PrefixNodeStruct, see Figure9) based on the prefix's content.
 - The created prefix node is placed into an entry(PrefixEntryStruct, see Figure9) in the prefix hash table based on the hash value of the prefix's content.
 - Update its linked attribute node field('dataAttr' field in Figure9) with the attribute node returned from step 2.
 - As each attribute node maintains a list of all linked prefix nodes, we also need to add this new created prefix node to that list of the attribute node returned from step 2
5. Extract all prefixes from the IPv4 unicast unreachable NLRI and multiprotocol unreachable NLRI. Each prefix is processed as follows:
 - If there is a existing prefix node which contains the same prefix content such as such as afi, safi, mask length and address in the prefix hash table:
 - Delete this prefix node from its corresponding attribute node's linked prefix nodes

list. If there isn't any prefix node linked to this attribute node after this deletion, remove this attribute node from the attribute hash table.

- Remove this prefix node from the prefix hash table.

- Otherwise: Do nothing.

6.4 Labels the BGP updates

The labeling module labels all the prefixes in a BGP update message base on how they change RIB-IN tables. More specifically, the label classifies the prefixes into six categories: 'new announcement' versus 'duplicate announcement', 'same path' versus 'different path' and 'withdraw' versus 'duplicate withdrawal'. In BGP one update consists of multiple prefixes which share the same set of attributes and these prefixes might change the RIB-IN tables in different ways. As a result, the multiple prefixes in one BGP updates have to be labeled separately. That's why labeling has to be done on a per-prefix basis, not a per-update basis.

Labeling is done as follows:

1. Return a existing or new created attribute node according the step 1 and 2 in the preview subsection.
2. Extract all prefixes from IPv4 unicast reachable NLRI and multiprotocol reachable NLRI. Each prefix is labeled as follows:

- If there is a existing prefix node which contains the same prefix content such as such as afi, safi, mask length and address in the prefix hash table:
 - If the existing prefix node's linked attribute node ('attributeNode' field in Figure9 is same as the attribute node return from step 1, label it as 'duplicate announcement'.
 - Otherwise, compare the 'AS Path' in this prefix node's current linked attribute node to the 'AS Path' in the attribute node returned from step 1.
 - * If they are same, label it as 'same path'.
 - * Otherwise, label it as 'different path'.
- Otherwise, label it as 'new announcement'.

3. Extract the IPv4 unicast unreachable NLRI and multiprotocol unreachable NLRI. Each prefix is labeled as follows:

- If there is a existing prefix node which contains the same prefix content such as such as afi, safi, mask length and address in the prefix hash table, label it as 'withdraw'
- Otherwise: label it as 'duplicate withdraw'.

6.5 Design Philosophy

The first design issue here is how to organize a RIB table in memory. Logically a RIB table can be viewed as prefixes and attributes that are interlinked together. And another observation is that a set of attributes is typically shared by multiple prefixes as they are received in one BGP update. Based on these observations, we decided to store a RIB by using 2 interlinked hash tables: prefix hash table and attribute hash table. As we discussed in subsections 6.1.1 and 6.1.2, these 2 hash tables has the same structure in a high level. Specifically both of them consists of multiple entries and each entry is a linked list that has multiple nodes. In the prefix hash table each node represents a prefix and in attribute hash table each node holds a set of attributes. And the prefixes and the set of attributes are linked together if they are received in the same BGP update. Note one prefix node representing a single prefix can only be linked to one attribute node holding a set of attributes. But one attribute node could be linked to multiple prefix nodes. In other words, the relationship between prefix node and attribute node is many to one.

The second design issue is how to store the prefixes and the set of attributes in a BGP update efficiently. For a prefix it is straightforward to hash the prefix to an entry(bucket) in the prefix hash table. Then all the nodes of this entry are checked in order to know if this prefix is existing or not. For the set of attributes, we have 2 options here:

- *Option1*: Hash the entire set of attributes to an entry of attribute hash table.
- *Option2*: Extract the AS path form the set of attributes and then hash the AS path to find an entry for the set of attributes.

Each option has its own pros and cons. Option1 basically hashes each unique set of attributes to a different entry(bucket) if ignoring the hash collision. That will make the attribute hash table too long in terms of the number of entries. In option2 each set of attributes is assigned to a entry based on the hash value of its AS path. The implication is that all sets of attributes with the same AS path will be assigned to the same entry. As a result, option2 will take fewer entries than needed by option1. But each entry will have more nodes in option2 than option1. So it is a tradeoff between the number of entries and the length of entries.

But besides that option2 is better in comparing the AS paths in 2 sets of attributes. In option2 2 sets of attributes that have different AS paths must be assigned to the different entry. But 2 sets of attributes assigned to the same entry don't necessarily have the same AS path because of hash collision. So when attribute node gets created we tag its AS path with ID in order to differentiate the different AS paths that are assigned to

the same entry because of hash collision. As a result, we can compare the AS paths in 2 sets of attributes as follows:

- If they are assigned to different entries, we know they have different AS paths.
- If they are assigned to the same entry, continue to compare their AS Paths' IDs.
 - If they are the same, we know they have the same AS path.
 - Otherwise, we know they have different AS paths.

Contrast to option2, comparing the AS paths in 2 sets of attributes would be more complex in option1 because one first needs to extract the 2 AS paths from them then compare the 2 AS paths in bitwise. That could be a big flaw of option1 as this operation is done for every receiving prefix in BGPmon.

Another advantage of option2 is that all the sets of attributes have the same AS path will share the same AS path structure in memory. In option1 each set of attributes stores the AS path respectively even some of them have the same AS path. So option2 is also better in terms of memory consumption. Based on all of these, we decided to use option2 in our current design.

The last design issue here is about configuration changes. As we mentioned, the only related configure item here is "labelAction" which has 3 values: None, RibStore and Label. All the possible changes are listed:

- If "labelAction" changes from Label to RibStore, nothing needs to be done. The labeling module will automatically turn off the labeling function.
- If "labelAction" changes from RibStore to Label, nothing needs to be done. The labeling module will automatically turn on the labeling function.
- If "labelAction" changes from RibStore to None, the RIB table needs to be freed.
- If "labelAction" changes from Label to None, the RIB table needs to be freed.
- If "labelAction" changes from None to Label, this change cannot be applied immediately. If we apply this change immediately, the labeling will misbehave as we don't have the RIB. For instance, right after the change all the updates labelled as 'new announcement' are actually not new.
- If "labelAction" changes from None to RibStore, this change cannot be applied immediately either. If we start to store RIB immediately and right after that user further changes it from RibStore to Label, we will run into the same problem as above.

The only way to make the last 2 changes applied is to reset the session. When a new session starts, the changes will be applied automatically and labeling will behave correctly.

7. PERIODIC EVENT HANDLING MODULE

Periodic event Handling Module(periodic module in short) manages periodic events such as route refresh requests and periodic status messages. This module has two threads:

- *Periodic Status Messages Thread*: It periodically writes session status messages(BMF type 5), queues status messages(BMF type 6) and chains status messages(BMF type 7) into label queue if configured.
- *Route Refresh Request Thread*: It centralized schedules and executes the route refreshes for all the peers if configured. For each peer, there are two possibilities.
 - If this peer supports route refresh, periodic module will notify peering module to send route refresh request to the peer.
 - If this peer doesn't support route refresh, periodic module will simulate route refresh by sending local stored RIB-IN out.

The main data structure of periodic event handling module has five fields:

- *StatusMessageInterval*: It indicates the interval of sending periodic status messages for peering sessions, queues and chains. It is configured by user.
- *RouteRefreshInterval*: It indicates the interval of requesting route refresh for every peer that is configured for route refresh. It is configured by user.
- *labelQueueWriter*: It is used to write messages into the label queue.
- *shutDownFlag*: It is used to signal periodic module to exit. Specifically both threads keep checking this flag, they will quit if it becomes TRUE.

Next we will discuss the detail of the two threads.

7.1 Periodic Status Messages Thread

This thread writes one status message for each peering session, one status message for all queues and one status message for all chains into the label queue every 'StatusMessageInterval' interval. Peering session status message (BMF type 5) only includes a session ID instead of including the detail information. Queues' status message(BMF type 6) and chains' status message(BMF type 7) are not associated with a particular peering session.

Then when XML module reads a session status message from label queue, it will extract the session ID from it and retrieve the detail information based on the session ID. Finally the detail information of a peering session will be converted to XML and write into a XML queue. The detail is built from peering session structure(see section 4.2.1). More specifically, it consists of the follow fields: When XML module reads a queues status message from label queue, it will ignore the session ID from it and retrieve the status information all queues. Finally the status information of all peers will be converted to XML and write into a XML queue. Similarly when XML module reads a chains status message from label queue, it will ignore the session ID from it and retrieve the status information all chains. For the XML format of these periodic status messages, please refer to the XML specification.

7.2 Route Refresh Request Thread

This thread distributes the route refresh requests for all established peering session evenly over time in order to prevent the queues being overwhelmed.

If a route refresh request is for one peer that supports route refresh, the periodic module just needs to notify peering module to issue a route refresh request message. Otherwise the periodic module has to simulate a route refresh by sending out this peer's local RIB-IN. The detail of handling the route refresh request for a establish peering session is as follows:

- Check if route refresh is enabled for this session. If not, do nothing. Otherwise, continue.
- Check the 'routeRefreshType' field of session structure(see section 4.2.1)..
 - if it is 0, that means this peer doesn't support route refresh. The periodic module needs to send out this peer's RIB-IN table as follows:
 - * The 'prefixTable' and 'attributeTable' fields of the found RL structure compose this peer's RIB-IN. For each node in the attribute table, do the following things:
 - Find all the related prefix nodes in the prefix table.
 - Build a BMF message(type 4) that contains a BGP update that is built based on the attribute node and all related prefix nodes.
 - Write this BMF message into label queue.
 - Otherwise, that means this peer supports route refresh. The periodic module just needs to set the 'routeRefreshFlag' field of session structure to TRUE in order to signal the peering module to send out route refresh request to the peer.

7.3 Design Philosophy

Actually in the previous design we used to put the logic of sending route refresh requests and periodic status messages in the peering module. More specifically each peering thread decides when to send its own route refresh requests and periodic status messages. But it turns out we lost the ability to schedule these events from a global view. Specially for route refresh requests, it is important to schedule them carefully as each of them will trigger hundreds of thousands of messages that could be a big burden for the entire system. If all the peers' route refresh request are scheduled at the same time, the queues in BGPmon might not be able to hold the huge amount of messages triggered by them.

In the current design, we have a dedicated module to schedule these events. The route refresh requests of all peers are scheduled evenly over the time based based on "RouteRefreshInterval" and the number of peers in order to prevent the queues from being overwhelmed. In contrast to route refresh requests, the periodic status messages of all peers are simply written into the label queue every "StatusMessageInterval" as the size of periodic status message is pretty small. But by decoupling the scheduling from peering module it is easy to plug in some sophisticated scheduling algorithms for the periodic status message as needed.

8. XML GENERATION MODULE

The XML generation module manages the conversion of all BGPmon received and generated messages into XML. It has one single thread which consists of three main steps.

- It reads messages from the label queue. These messages can be any of all eight types in Figure 3.
- It converts all types of messages into XML format according to our XML specification.
- It writes messages in XML format to the XML queue for processing by the client threads.

8.1 XML Format Overview

The XML module converts all the messages from BMF to XFB, a XML-based format for BGP routing information. XML is a general-purpose markup language; its primary purpose is to facilitate the sharing of data across different information systems, particularly via the Internet. Using XML as the base for our XFB markup provides the following advantages:

- XFB is human and machine-readable. By using CSS or XSL, XFB can be easily displayed on websites. Because XFB is based on XML which is a common interface to many applications, XFB can be processed by a variety of existing tools.

- XFB can easily be extended with additional information based on the raw BGP routing information. The BGP data is simply annotated with additional attributes and/or elements; programs which are not looking for this new information will simply ignore it. This allows us to easily modify XFB in general (or particular BGPmons) to allow for newly required information. We include guidelines for adding new standard elements in each section.
- XFB messages can be used to reconstruct the raw BGP messages, if needed.

Though XFB pays a storage cost since a compact binary message is (usually) expanded into ASCII text with additional tags, the results of our experiments using the default compression parameters for bzip2 on XFB data are promising. Currently there are two types of BGP routing information which are included in XFB: BGP messages which come "over the wire" and may or may not have additional "helper" information appended, and BGP control information that originates with the BGPmon. For the details about XFB, please refer to the BGPmon XFB specification.

8.2 Design Philosophy

There are two issues when we design how to convert messages to xml.

- How to convert the fields which are not defined in xml specification? The answer to this question is each unknown field is represented by the a 'Octets' element. The 'Octets' element looks like: `<octets length = '3'>2E3A4D</octets>`. In this way, we avoid any information loss even for the information we don't know.
- How to convert the xml message back to binary? Similar to the previous one, the solution is we piggyback a 'Octets' field which represent the entire BGP raw message from wire in the end of xml message. In this way, we can easily replay some BGP raw messages to routers by extracting the last 'Octets' field.

9. CLIENTS CONTROL MODULE

This module is used to manage the BGPmon clients. Clients control module consists of a single server thread and multiple client threads.

- *Server Thread*: It is a TCP server that listens on a specific port and spawns one client thread for each allowed client.
- *Client Thread*: Each client thread reads the messages from the XML queue and sends them to the client via a TCP connection.

During the initial BGPmon startup, the server thread needs to start first to allow clients to connect before the peering threads begin. This allows the clients to receive the complete set of messages which is particularly important for logging client. Some messages will be skipped when a client becomes unresponsive or is unable to keep up with the messages stream. This addresses the need for support of real-time support in BGPmon as slow clients cannot affect the entire system.

9.1 Data Structure

The main data structure of clients control module consists of the following fields:

- *listenAddr*: The listening socket of server thread binds to this address. It is a string which could be a IPv4/IPv6 address or one of four keywords(ipv4loopback, ipv4any, ipv6loopback and ipv6any). After it is initialized from configuration, it could be set via command line interface(CLI) at runtime.
- *listenPort*: It is the port on which server thread listens. It is an integer. It also could be set via command line interface(CLI) at runtime after initialization.
- *enabled*: It indicates the status of server thread. If it is false, server thread stops listening but the existing clients still run. Otherwise server thread listens on 'listenPort' and accepts allowed clients. It could be set via CLI after initialization.
- *maxClients*: It is the max number of clients. It could be set via CLI after initialization.
- *activeClients*: It is the number of connected clients.
- *nextClientID*: It is the ID for the next client to connect.
- *rebindFlag*: If listenAddr or listenPort changes, this flag will be set to TRUE. That means the listening socket of server thread will bind to the new address or port. It is set by CLI.
- *shutdown*: It is a flag to indicate whether to stop the server thread.
- *lastAction*: It is a timestamp to indicate the last time the thread was active.
- *firstNode*: It is the header of a linked list which maintains the information of all connected clients. Figure 11 shows the details of client structure of this linked list.
- *clientLock*: It is a pthread mutex lock used to lock the clients info linked list when clients are added or deleted.

- *lastAction*: It is a timestamp to indicate the last time the thread was active. This structure is mainly maintained by server thread.

9.2 Server Thread

Server thread has 2 main tasks:

- Listen on "listenAddr" and "listenPort" and periodically every `THREAD_CHECK_INTERVAL` (60s by default) check the values of following three fields. These three fields could be changed by command line interface (CLI).
 - * *shutdown*: If it is TRUE, the server thread will be closed.
 - * *enabled*: If it is FALSE and "shutdown" is FALSE, close the current listening socket if any. If it is TRUE and "shutdown" is FALSE, open a new listening socket if there is no listening socket.
 - * *rebindFlag*: If it is TRUE and "shutdown" is FALSE, we need to close the current listening socket and open a new listening socket based on the current "listenPort" and "listenAddr". This flag is typically set TRUE after changing "listenPort" and "listenAddr".
- Accept the new clients and check them against Access Control List(ACL).
 - * If a new client pass the ACL check, a new thread will be spawned for it and a new node will be added to the linked list. Then server thread goes back to listen.
 - * If a new client doesn't pass the ACL check, the client socket(returned by accept system call) will be close and server thread keeps listening.

9.3 Client Thread

Client thread just reads the messages from XML queue and then writes the messages to the client via socket. Each client thread is associated with the following client structure(See Figure 11).

- *id*: identification number of the client.
- *addr*: client's address.
- *port*: client's port.
- *socket*: client's socket for writing messages to the client.
- *connectedTime*: client's connected time in seconds.
- *lastAction*: client's last action timestamp.
- *qReader*: This is a queue reader(see section 11). It is used to read messages from XML queue.

- *deleteClient*: flag to indicate to close the client thread and cleanup client structure.

Note the client thread might exit by itself because of the following 2 reasons:

- This client thread fails to read the messages from XML queue. In this case, the client thread needs to exit.
- This client thread fails to write messages to "socket".

The client thread also might be deleted(call `deleteClient`) by command line interface(CLI) at running time. Note deleting a client thread is asynchronous action. In other words, "deleteClient" function only sets the "deleteClient" flag as TRUE. This deletion will be deferred to the next time the child thread checks the "deleteClient" flag. At that time the client thread will exit and the client info in the linked list will be removed.

9.4 Design Philosophy

The important design decision here is that we should let each client specify what they want to receive from BGPmon or we just make BGPmon blindly send everything to the every client. In the previous design, we did let clients submit their own filters to specify what kind of data they want to receive from BGPmon. But then we realized it would be a huge burden if hundreds of clients all specify their own complex filters. As our main design principle is to let BGPmon provide a real-time event stream to a large number of clients, we should relieve BGPmon from the huge burden of handling clients' filters. As a result, in current design BGPmon simply sends all data to all clients without any processing.

10. CHAIN MODULE

This module is used to allow BGPmon to scale out through chaining multiple BGPmons. BGPmons are chained together via tcp connections. Conguration of BGPmon chains is manual so care must be taken not to create loops in the topology. One BGPmon could initialize a chain to another BGPmon or accept a chain from another BGPmon. From the perspective of BGPmon accepting a chain, the BGPmon intializing the chain is same as a typical client. As a result, the logic of accepting a chain and serving data is already handled by clients control module(see section9).

On another side, Initializing a chain and processing data are implemented in this chain module. At the beginning of BGPmon, for each configured chain its data structure is populated and its threads is launched if it is enabled. After that, chains can be created, deleted, disabled and enabled via command line interface(CLI).

10.1 Data Structure

The main data structure of a chain has the following fields:

ClientNode:

Field Name	Type	Description
id	int	Client's identification number
addr	Char []	Client's addr
port	int	Client's port
socket	int	Client's socket
connectedTime	time_t	Indicates how long the client has connected to BGPmon
lastAction	time_t	Client's last action timestamp
qReader	QueueReader	XMLqueue's reader
deleteChain	int	flag to indicate to close the client thread and cleanup client structure
Next	a pointer to ClientNode	pointer to the next node in the linked list

Figure 11: Client Structure

- *chainID*: It is the unique identifier of a chain. It is an integer starting from 0 and automatically assigned when a new chain is created.
- *addr*: It is the address of remote BGPmon. It is a string which could be a IPv4/IPv6 address or one of two keywords(ipv4loopback and ipv6loopback). After it is initialized from configuration, it could be set via command line interface(CLI) at runtime.
- *port*: It is the listening port of remote BGPmon. It is an integer. It also could be set via command line interface(CLI) at runtime after initialization.
- *enabled*: It is a boolean value indicating the status of a chain. If it is FALSE, the chain thread will exit. It could be set via command line interface(CLI) at runtime after initialization
- *connectRetryInterval*: It is the tcp connection retry interval in seconds. It could be set via command line interface(CLI) at runtime after initialization
- *deleteChain*: This flag will be checked after a chain is disabled. If it is TRUE, the chain's data structure will be freed. It is set by via command line interface(CLI).
- *reconnectFlag*: This flag will be checked every time a message is received or periodic check timer expires. Any changes of "addr" or "port" will set this flag to TRUE. If it is TRUE, the existing tcp connection will be torn down and a new tcp connection(with the latest "addr" and "port") will be initialized.
- *lastAction*: It is a timestamp to indicate when is the last action of this chain. It is used to infer the liveness of the chain thread by thread management module. The chain thread keeps update this timestamp when it is alive. If this field hasn't been updated for a while, the thread management module can infer the chain thread is dead.
- *runningFlag*: It is a flag to indicate if the chain thread is running or not. It should be set to FALSE when the chain thread normally exits.
- *socket*: It is the socket of a chain.
- *errno*: It is socket error code.
- *connectRetryCounter*: It is the number of times of retrying a tcp connection.
- *connectionState*: It is current connection state of a chain. It could be one of these: chainStateIdle, chainStateConnecting and chainStateConnected.
- *msgHeaderBuf*: It is used to buffer the header(first 100 bytes) of a XML message. With the length field of header, we can figure out how long the XML messages. Then the complete XML message can be read from the socket and written into the XML queue. Basically every message written into XML queue must be a complete XML message with open tag and close tag, not a partial message. Otherwise the XML messages from different chains will be mangled.
- *establishedTime*: It is a timestamp indicating when a chain got connected to remote BGPmon.
- *lastDownTime*: It indicates when is the last down time of tcp connection.
- *resetCounter*: It is the number of tcp connection resets.

- *messageRcvd*: It is the number of received XML messages via a chain.
- *periodicCheckInt*: It indicates how often the periodic check timer expires.
- *xmlQueueWriter*: It is used to write xml messages to XML queue.

This data structure is attached to each chain thread.

10.2 Chain Thread

Each chain is a separate thread and has the following tasks:

- Initialize a tcp connection to a configured remote BGPmon instance(sending side of the chain).
- Read the XML stream via the tcp connection, cut the stream into messages and write the messages into XML queue.
- Check the 2 flags: "enabled" and "reconnectFlag" every time a XML message is received or periodic check timer expires .
 - The "enabled" flag is set directly by CLI. When it is TRUE, the client thread will exit by itself and if the "deleteChain" flag is also TRUE its corresponding data structure will be freed.
 - The "reconnectFlag" is also set by CLI. Any changes of "addr" and "port" will set this flag TRUE. If it is TRUE, the existing tcp connection will be torn down and a new tcp connection will be initialized with the latest "addr" and "port".

10.3 Chain Management

Chains management is done via command line interface(CLI). There are 4 possible chain operations.

10.3.1 Create a Chain

Creating a chain is a synchronous operation. It will occur immediately after function "createChain" is called by CLI. It consists of 2 steps:

- Populate the new chain's data structure.
- Launch a thread for the new chain if its initial 'enabled' flag is TRUE.

10.3.2 Enable a New Chain

Enabling a chain is a synchronous operation. A new thread will be immediately launched after function "enableChain" is called by CLI. Note the chain's data structure must be existing when the function "enableChain" is called.

10.3.3 Disable a New Chain

Disabling a chain is a asynchronous operation. It will NOT occur immediately by calling function "disableChain" by CLI. The function "disableChain" only sets the flag "enabled" to FALSE. The chain thread will actually exit when a new XML message is received or periodic check timer expires. The difference between disabling a chain and deleting a chain is that disabling a chain will not free the chain's data structure.

10.3.4 Delete a Chain

Deleting a chain is a asynchronous operation. It will NOT occur immediately by calling function "deleteChain" by CLI. Inside the function "deleteChain", both "enabled" flag and "deleteChain" flag are set to TRUE. The actual actions of exiting chain thread and freeing chain data structure are deferred to the next time a new XML message is received or periodic check timer expires.

10.4 Design Philosophy

If one downstream BGPmon is chained to multiple upstream BGPmons, the fundamental design issue is about how downstream BGPmon avoids to mingle the XML streams from upstream BGPmons. Remember all the XML streams from upstream BGPmons are mixed together into one stream at the downstream BGPmon by writing them into XML queue. That means the downstream BGPmon has to first divide the streams from upstream BGPmons into messages and then write all the messages into the XML queue.

We add a length field for each XML message in order to help BGPmon divide stream into messages by giving it a hint about how long the message is. More specifically, downstream BGPmon repeats the following steps to process a stream:

- Reads the first a few bytes from stream and figures out the length of the current message
- Extracts the message from the stream based on the length from the previous step
- Move the cursor to the end of the current message.

11. QUEUE MODULE

Queue is a utility module which is used by other modules to send messages from one to another. As shown in Figure 1, there are 3 queues in BGPmon.

- *Peer Queue*: It is used by peering module to send internal messages with type 1,2,4 and 6 to the rib and labeling module.
- *Label Queue*: It is used by rib and labeling module, periodic module and main module to send all eight types of internal messages to XML generation module.

- *XML Queue*: It is used by XML generation module to send XML messages to client management module which in turn sends them to client applications.

Each of them is a running instance of queue module. They are created by main module during the initial phase of BGPmon and then used by other modules.

The queue module is build on a circular array and each item in this array contains a generic pointer which points to the real message. In this way we can make queue module generic enough to hold all kinds of messages. It also keeps track all the readers and writers. The main data structure of queue module will be introduced in subsection 11.1.

The queue module implements a Readers/Writers pattern where multiple threads may access the same queue simultaneously, some reading and some writing. We call the reading thread reader and the writing thread writer. Each message written by a writer is available to all the readers and a message can be deleted from the queue only after all the readers read it. A key design issue is the locking mechanism to synchronize access to the share data structure in the queue among all threads. The details of thread synchronization will be discussed in subsection 11.2.

The biggest challenge in the design of queue module is to avoid being overwhelmed with limited queue length. There are two situations we need to address: 1) writers write too fast, and 2) Readers read too slow. We will discuss how to address these two situations in subsection 11.4.

11.1 Main Data Structure

The main data structure of queue module is called 'Queue'. It consists of four parts:

- *General Substructure*: It holds the general information for this queue such as the queue's name, its mutex lock and some logging information. See the details in 11.1.1.
- *Items Substructure*: It contains all the data of the queue. It implements a circular array. See the details in 11.1.2.
- *Readers Substructure*: It contains the information of all readers of the queue. For example, the sequence number of the next unread item for each reader needs to be maintained here. See the details in 11.1.3.
- *Writer and Pacing Substructure*: It is used to track all the writers in order to pace them when needed. And some other information needed for pacing are also included here such as pacing on/off threshold. See the details in 11.1.4.

11.1.1 General Substructure

The general substructure has the following fields:

- *name*: It is the name of the queue. In BGPmon, the name of peer queue is 'PeerQueue', the name of lable queue is 'LabelQueue' and the name of XML queue is 'XMLQueue'.
- *queueLock*: It is a pthread mutex lock which is used for thread synchronization.
- *queuecond*: It is a pthread condition variable which is used to notify a reader when the new item gets wrote.
- *logging related fields*: A group of logging related fields such as the historical max number of messages, the historical max number of readers and the historical max number of writers.

Figure 12 shows the details.

11.1.2 Items Substructure

Items substructure maintains a circular array of items which is the heart of queue module. Each item is defined as a 'QueueEntry' structure which has the following two fields:

- *count*: It indicates how many readers haven't read it. This reference count decrements by one after one reader reads this item. If the reference count of a item is zero, this item can be reclaimed by the queue module. Otherwise this item cannot be reclaimed.
- *messageBuf(void *)*: It points to the real data buffer of this item. The data buffer must be created by writers and be passed into queue module.

Items substructure has the following five fields:

- *head*: It is the sequence number of the oldest item in the queue. It means at least one reader hasn't read this item. The 'head' is incremented after the oldest data item is read by the last reader.
- *tail*: It is the sequence number of the next available item in the queue. It is incremented by writing a new message into the queue.
- *items*: It is an array of items. Each item is a 'QueueEntry' structure.
- *copy*: It is a callback function. If a reader reads a item and it is not the last reader for this item, the callback function will be called to return a copy of this item to the reader. If a reader is the last one of a item, this item will be returned directly. This allows the reader to always free the returned item after processing it without knowing the details of queue.

Field Name	Type	Description
name	char []	the name of the queue, set by configuration module
queueLock	pthread_mutex_t	the pthread mutex lock used to synchorize the threads
queueCond	pthread_cond_t	the pthread condition variable used to let readers wait when needed
lastLogTime	time_t	when was the last time of outputting logging information
logMaxMessages	int	the historical max number of messages in this queue
logMaxReaders	int	the historical max number of readers in this queue
logMaxWriters	int	the historical max number of writers in this queue
logPacingCount	int	the accumulative number of times of pacing

Figure 12: General Substructure of Queue

- *sizeof*: It is a callback function. Based on 'QueueEntry' structure, we know the size of a item depends the size of its data buffer. This function is used to get the size of a data buffer in bytes. As the data format is queue specific, the sizeof function is provided by the queue creator.

In order to avoid drop any messages in the queue, the different between 'head' and 'tail' must be smaller than 'max' field. Note 'head' and 'tail' are all sequence numbers, not subscripts of array. A sequence number(seq) is long integer and is a logical subscript of the queue. The subscript to access the physical array can be calculated by $seq \% max$. Figure 13 shows the details of this substructure.

11.1.3 Readers Substructure

This substructure is used to track all the readers. It has the following three fields:

- *readerCount*: It is the current number of readers.
- *nextItem*: It is an array of sequence numbers for all the readers. This array is indexed by reader ID and each sequence number indicates the next unread item in the queue for the corresponding reader. For example, `nextItem[6]` is the sequence number of the next unread item for the reader with ID 6. The length of this array equals the max number of readers.
- *itemsRead*: This array is indexed by reader ID and each element indicates total items read by the corresponding reader. For example, `itemsRead[6]` is the number of total read items by the reader with ID 6. The length of this array also equals the max number of readers.

Note the readers of the same queue might have different next item.

Figure 14 shows an example of the queue which can help us understand the items substructure and readers substructure. In this example, there are a queue with

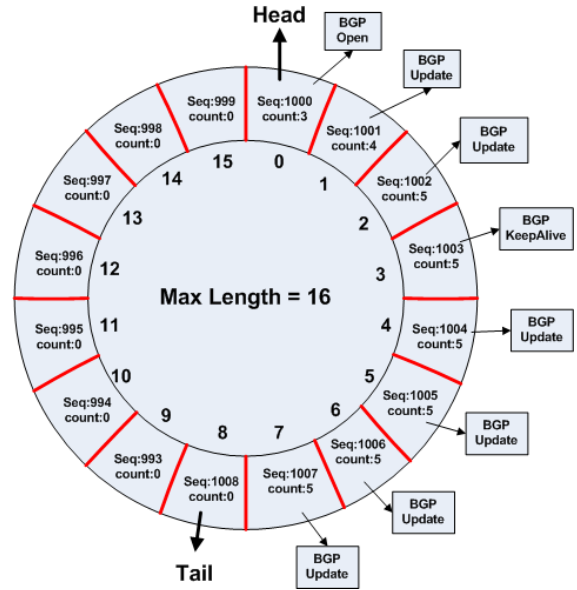



Figure 14: An Example of Queue

max length 16 and 5 readers. The items substructure looks like this:

- *head*: is 1000 which is sequence number of the oldest item. The count of that item is 3 which means there are three readers haven't read it.
- *tail*: is 1008 which is sequence number of the next available item for the new message.

Field Name	Type	Description
head	long	the sequence number of the oldest item in the queue
tail	long	the sequence number of the next available slot in the queue
Items	an array of QueueEntry structure	an array of all items in the queue
copy	a function pointer	a callback called when readers read a item from the queue
sizeof	a function pointer	a callback called to get the size of a item



QueueEntry Structure		
Field Name	Type	Description
count	int	a reference count which indicates how many readers haven't read it
messageBuf	void *	the pointer to the real data buffer

Figure 13: Items Substructure of Queue

- *items*: It is an array of 16 items. 8 of them are in use.
 - The item with sequence number 1000 has a reference count as 3 which means 3 readers haven't read it.
 - The item with sequence number 1001 has a reference count as 4 which means 4 readers haven't read it.
 - All the others item with sequence number from 1002 to 1007 has a reference count as 5 which means all 5 readers haven't read it.

The corresponding readers substructure is as follows:

- *readerCount*: is 5 which means there are 5 readers.
- *nextItem[0]*: is 1000 which means the next item for the reader 0 is 1000 .
- *nextItem[1]*: is 1000 which means the next item for the reader 1 is 1000 .
- *nextItem[2]*: is 1000 which means the next item for the reader 2 is 1000 .
- *nextItem[3]*: is 1001 which means the next item for the reader 3 is 1001 .
- *nextItem[4]*: is 1002 which means the next item for the reader 4 is 1002 .
- *itemsRead[0]*: is 0 which means reader 0 hasn't read anything .
- *itemsRead[1]*: is 0 which means reader 1 hasn't read anything.

- *itemsRead[2]*: is 0 which means reader 2 hasn't read anything.
- *itemsRead[3]*: is 1 which means reader 3 has read 1 items.
- *itemsRead[4]*: is 2 which means reader 4 has read 2 items.

11.1.4 Writers and Pacing Substructure

This structure is used to track the writing rate of all the writers and pace them when needed. It has the following fields:

- *writerCount*: It is the number of current writers.
- *tick*: It is the start time of the current pacing interval.
- *readCount*: It is used together with 'tick' to count how many messages are read in one interval by all the readers.
- *writeCounts*: It is an array of counts for all the writers. One count is for each writer which is used together with 'tick' to count how many messages are written in one interval. The length of this array equals the max number of writers.
- *writesLimit*: It is how many messages are allowed to be written in the queue by each writer in one interval when pacing is turned on.
- *pacingFlag*: It is set to TRUE if pacing is turned on. It is set to FALSE if pacing is turned off.

Field Name	Type	Description
writercount	int	the number of current writers
tick	time_t	the start time of the current pacing interval
writeCounts	Int[]	an array of counts for all the writers which are used together with 'tick' to count how many messages are written in one interval
readCount	int	it is used together with 'tick' to count how many messages are read in one interval across all the readers
writesLimit	int	how many messages are allowed to be written in the queue by each writer in one interval when pacing is on
pacingFlag	int	1 if pacing is turned on 0 if pacing is turned off

Figure 15: Writers and Pacing Substructure of Queue

Figure 15 shows the details of this substructure. In the subsection 11.4, we will discuss how pacing works by using this substructure in detail.

11.2 Thread Synchronization

In BGPmon, there are multiple writers that write messages into one queue and multiple readers that read messages from the same queue. As each read/writer is a separate thread, the thread synchronization is very important for the queue. In our design we use mutex lock and condition variable to manage the thread synchronization. Basically the writer needs to lock the queue by obtaining 'queuelock' in general substructure before it writes a message and unlock the queue by release 'queuelock' after it writes a message. Similarly the reader needs to lock the queue before it reads a message and unlock the queue after it reads a message.

When a reader exhausts the messages in the queue, it must wait for new messages while other readers or writers still need to continue their reading or writing. In our design, if a reader wants to read a message from the queue and successfully locks the queue by obtaining 'queuelock', it will do the following checks:

- If there are some new messages available for it to read, it will read the oldest one and unlock the queue.
- Otherwise, it unlocks the queue and waits on the condition variable 'queueCond' to lock the queue again. If any writer writes any new messages into the queue, the queue will broadcast this condition to all the waiting readers.

11.3 Interface Functions

In the queue module, there are only two interface functions which can be used by other modules.

- *readQueue*: It is used to read a message from

queue by giving a queue ID, reader ID and a pointer which points to the result data buffer. The return value of 'readQueue' is the number of remaining messages of this reader. It may return `READER_SLOT_AVAILABLE` if this reader has ceased. See details in 11.4.2.

- *writeQueue*: It is used to write a message into queue by giving a queue ID, writer ID and a pointer which points to the written message. It returns 0 if successes.

11.4 Stream Control

In our design, the queue removes a message until all the readers read it. As a result there are two things we can do in order to prevent a queue being overwhelmed.

- *Pacing writers*: The queue paces the writers according to the average reading rate across all the readers. For example the queue is almost full and there are 4 readers and 2 writers. If each of reader can read 8 messages per second averagely, we should limit the writing rate of each writer to $8/2 = 4$ in order to avoid overwhelm the queue. See details in 11.4.1.
- *Adjust slow readers*: In some cases pacing writers doesn't work, the queue needs to adjust the slow readers. For example, the queue is almost full and there are 4 readers and 2 writers. If 3 of the 4 readers can read 8 messages per second but one of them can only read 2 message per second. As a result, the average reading rate across the 4 readers is 6.5 messages per second. In this case, even pacing is turned on and each writer is limited to write $6.5/2 = 3.25$ messages per second the queue will still be overwhelmed because the slowest reader cannot read 3.25 messages per second.

That's why in this case the queue has to adjust the slowest reader by skipping its unread items. See details in 11.4.2.

11.4.1 Pacing Writers

In our design, the queue utilization (from 0% to 100%) is used to determine when to turn on pacing. When the queue utilization exceeds a configurable threshold (PacingOnThreshold), pacing is turned on until the queue utilization drops below a configurable threshold (PacingOffThreshold). The reason why we have two thresholds here is this allows the queue to reach a steady state rather than oscillate in and out of pacing. During the pacing period, the objective is to make the writers write at a pace that matches the average reader. More specifically, in each configurable interval (PacingInterval) the queue limits the number of writes from each writer according to the average number of reads by all readers. In order to do this, when a new interval starts we need to predict the number of reads by all readers in this new interval and then use this value to pace the writers in this new interval. The pacing related logic related to the 'readQueue' function is as follows:

1. Check if a new interval starts
 - (a) If yes, update the 'writesLimit' using exponential weighted moving average(EWMA):
 - i. Calculate the new 'writesLimit' by this formula. Note 'alpha' is configurable, 'writesLimit' is the number of writes allowed per writer in the new interval, 'averageReads' is the average number of reads by all readers in the past interval and 'writerCount' is the number of writers.

$$writesLimit = (1 - alpha) * writesLimit + alpha * \frac{averageReads}{writerCount}$$
 - ii. If new 'writesLimit' is larger than half of the remaining queue, use half of the remaining queue as the new 'writesLimit'.
 - iii. If new 'writesLimit' is smaller than a configurable 'minWritesLimit', use 'minWritesLimit' as the new 'writesLimit'.
 - (b) Otherwise, do nothing.
2. Check if needs to turn off pacing by compare the queue utilization with the configurable threshold (PacingOffThreshold)
 - (a) If the queue utilization is smaller, set the 'pacingFlag' field as FALSE.
 - (b) Otherwise, do nothing.

Note this logic will be executed every time a reader calls 'readQueue'.

The 'writeQueue' function starts with the same pacing related logic as 'readQueue'. But after that, it needs to limit the number of writes per interval according to the 'writersLimit' field when pacing is enabled. This additional step inside the 'writeQueue' function is as follows:

1. Check if needs to turn on pacing by compare the queue utilization with the configurable threshold (PacingOnThreshold)
 - (a) If the queue utilization is larger, set the 'pacingFlag' field as TRUE.
 - (b) Otherwise, do nothing.
2. Check if 'pacingFlag' is set to TRUE.
 - (a) If yes, do the following checks:
 - i. If 'writeCount[writerID]' is larger than 'writesLimit', sleep until a new interval starts.
 - ii. Otherwise, do nothing.
 - (b) Otherwise, do nothing.

This logic will be executed every time a writer calls 'writeQueue'.

11.4.2 Adjust Slow Readers

Pacing prevents the queue from being overwhelmed if the readers are uniformly able to read the messages at the same rate. In the case of a slow reader, the queue utilization will still continue to grow. When the queue utilization reaches the maximum, the responsible reader is adjusted by skipping all its unread items. This logic is mainly implemented in the 'writeQueue' function.

1. Check if the queue is full.
 - (a) If it is full, find the slowest reader and adjust it by skipping all its unread items. For each reader, check the nextItem[readerID] as follows:
 - i. If nextItem[readerID] equals to 'head', adjust this reader as mentioned.

In the 'readQueue' function, suppose the queueID, readerID and a pointer is passed in from a caller.

1. If head \neq nextItem[readerID] \neq tail, pass the oldest message to the caller via the pointer and return the number of remaining messages.
2. If nextItem[readerID] \neq tail, block the caller until a new message is written into the queue.
3. If nextItem[readerID] = READER_SLOT_AVAILABLE, it means the reader has ceased and return READER_SLOT to the caller.

If a caller receives READER_SLOT_AVAILABLE from 'readQueue', that means the corresponding reader has ceased. Then the caller may need to close thread and release resources in this case.

11.5 Design Philosophy

The most important design issue here is how to handle slow readers. As we discussed, it is essential that some action be taken to address the problem of slow readers. If no action is taken, a slow reader can cause the queue to overflow and eventually data would be dropped. This is particularly problematic if most readers could read at a high rate and receive all the data, but a few slow readers fill the queues and cause data loss.

In the previous design, our solution is to identify and then terminate the slow readers. However, by doing that a potential problem is that the slow reader may simply re-connect and thus drive the overall system into a state of persistent oscillation. The system runs well until the slow reader joins. The slow reader then causes queues to build up and the reader is eventually killed. The queue then quickly drains when the slow reader is killed. Note that the queue contains at least one update that has been read by everyone except the slow reader. When the slow reader is killed, that update can be discarded. In our experiments thus far, a typical slow reader has hundreds of updates that are waiting only for the slow reader; killing the slow reader immediately removes these updates and frees hundreds of slots in the queue. But oscillation occurs if the slow reader immediately connects. The queue of unread updates begins to build again as soon as the slow reader joins and the cycle repeats. One can easily imagine a poorly written slow reader that automatically reconnects anytime it is disconnected.

An alternate approach is to better manage, but not kill the slow readers. In current design, the slow reader is not deleted from the system. Instead, slow readers are forced to skip messages. From a queuing standpoint, the effect is similar to killing the slow reader and works as follows. When BGPmon determines a reader is reading messages too slowly, all messages that have yet to be read by that slow reader are immediately marked as read. In this case the slow reader misses several messages, but it is allowed to continue.

12. LOGIN MODULE

The login module handles the Cisco-like commands typed by logged users and calls the corresponding functions provided by other modules. It is made up of several pieces.

The first piece is a listener that listens on a specific address and port for new Command Line Interface (CLI) connections. Once a connection has been established then the next major piece, the command tree structure, is used. This structure is a tree ADT that contains a complete mapping of all the commands that can be called from the CLI. Each command is broken down into parts and each part is added to the tree. For

instance the command 'show running' is broken into two pieces. The 'show' command is a child of the root node and the 'running' command is child of the 'show' command. When a command is executed from the CLI, the command is sent from the client to the server then broken down and mapped into the command structure. If the entire command can be mapped onto the tree then the final node will contain a function pointer for the command. Every node in the structure also has an associated mask with it that controls whether the command is visible or executable to a given security level. All the commands have been organized in a way that will help make maintenance of them easier. In `commandprompt.c` there are a series of functions that contain the necessary code to create the commands and associations to the necessary functions. Then, there is a series of files that end with `'_commands.c'` which contain the functions for each module that the commands reference.

13. ACKNOWLEDGEMENTS

Many of the design insights would not have been possible without the help of the Oregon RouteViews team, the UCLA Internet Research Lab, and the Networking Research Lab at University of Memphis, and the many contributors from the Colorado State Network Security Group.

14. REFERENCES

- [1] D. Matthews, N. Parrish, H. Yan, , and D. Massey. BGPmon: A real-time, scalable, extensible monitoring system. *Proceedings of the ACM SIGCOMM Internet Measurement Conference (IMC)*, 2008.
- [2] D. Matthews, H. Yan, , and D. Massey. BGPmon Administrator's Reference Manual, 2008.
- [3] unknown. A border gateway protocol 4 (bgp-4). <http://www.ietf.org/rfc/rfc4271.txt>.
- [4] unknown. Mrt routing information export format. <http://www.ietf.org/internet-drafts/draft-ietf-grow-mrt-11.txt>.
- [5] unknown. Ripe (rseaux ip europens) routing information service. <http://www.ripe.net/projects/ris/>.
- [6] unknown. University of oregon route views project. <http://www.routeviews.org/>.
- [7] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM.