C-BGP User's Guide

Bruno Quoitin
CSE Department, UCL, Belgium[1][2]

24th November 2004

[1]This work was supported by the European Commission within the ATRIUM project (www.alcatel.be/atrium) and is now being continued under the TOTEM project (totem.info.ucl.ac.be)

[2]This document refers to version 1.1.18 of C-BGP

Abstract

C-BGP is an efficient BGP decision process simulator. C-BGP can be used to experiment with modified decision processes and additional BGP attributes. It can also be used to evaluate the impact of input/output policies on the routing tables of other ASes. Thanks to its efficiency, it can be used with large topologies with sizes of the same order of magnitude than the Internet.

C-BGP is open source, written in C language and has been tested on various platforms like Linux, FreeBSD and Solaris. The BGP model implemented in C-BGP is not hindered by the transmission of BGP messages on simulated TCP connections as in packet-level simulators such as SSFNet or JavaSim. The simulator supports the complete BGP decision process, import and export filters, redistribution communities and route-reflectors. It is easily configurable through a CISCO-like command-line interface. C-BGP does not model the various BGP timers (MRAI, dampening, ...) and has a simplified session management.

To make large simulations easier to perform, C-BGP is able to load interdomain topologies produced by University of Berkeley and is also able to output all the exchanged BGP messages in MRTD format so that existing analysis scripts may be re-used. The simulator can load real BGP routing tables in MRTD format and can also save the routing tables resulting from a simulation in MRTD format.

Published: 2004

Computer Science and Engineering Department
University of Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

Contents

Chapter 1

Installation

1.1. LIBGDS installation

In order to build **C-BGP**, you will need to build and install the **libgds** library. This library is freely available from the **C-BGP** web site. In order to install the **libgds** library, download its sources `libgds-x.y.z.tar.gz` (where `x.y.z` denotes the version of the library) and follow the installation procedure described below. Note that some steps of this installation will require root privileges on the host where you install the library. If you have not such privilege, please read the section "non-standard installation" below.

First, untar the archive to a temporary directory on your host:

```
[tmp]$ tar xvzf libgds-x.y.z.tar.gz
```

This should create a new directory, `libgds-x.y.z`. Move to this directory and then run **./configure**.

```
[tmp]$ cd libgds-x.y.z
[libgds-x.y.z]$ ./configure
```

The configure script should run without any problem. When it is finished, simply type **make**. That will actually build the library.

```
[libgds-x.y.z]$ make
```

Once the build process is done, you must install the library. This step often requires that you have root privileges. To proceed with the installation, login as root and type **make install**. The default installation prefix is `/usr/local`. The installation process will install the library file under `<prefix>/lib`. It will also install the library headers under `<prefix>/include/libgds`.

```
[libgds-x.y.z]$ su
Password:
[libgds-x.y.z]# make install
```

Note: under the Linux operating system, you will need to run **/sbin/ldconfig** in order to update the linker's database of shared libraries. Running **ldconfig** requires root privilege. If you have installed the library under a non default path, you will probably need to update the configuration of the linker which is located in `/etc/ld.so.config`. If you have not the required privilege, you can use the alternative environment variable `LD_LIBRARY_PATH`. Please refer to the documentation of your operating system.

1.2. C-BGP installation

Once you have successfully setup the **libgds** library, you can start with the **C-BGP** build process. You must own the sources archive `cbgp-x.y.z.tar.gz` freely available from the **C-BGP** web site. The

procedure that you must use to build and install **C-BGP** is fairly similar to the above procedure used for the **libgds** library. First, untar the archive in a temporary directory, this will create a new directory `cbgp-x.y.z`. Move to that directory and run the **./configure** script. Once the configuration script is done, run **make**.

```
[tmp]$ tar xvzf cbgp-x.y.z.tar.gz
[tmp]$ cd cbgp-x.y.z
[cbgp-x.y.z]$ ./configure
[cbgp-x.y.z]$ make
[cbgp-x.y.z]$ make install
```

1.3. Installation in another directory

In certain cases, you will want to install the **libgds** library under another directory than the default `/usr/local`. This will be the case if you have not the required privileges to install under the default prefix. In this case, you must use the **./configure** script with an additional parameter *–prefix=<directory>*. For instance, in order to install under your own directory `/home/user/projects`:

```
[libgds-x.y.z]$ ./configure --prefix=/home/user/projects
[libgds-x.y.z]$ make
[libgds-x.y.z]$ make install
```

If you have installed the **libgds** library in a non-standard directory, will most probably encounter problems during the **C-BGP** build process. The **./configure** will probably complain because it is not able to find the **libgds** library or headers. To fix this problem, you must tell the **./configure** script about the special installation path of **libgds**. This is done with the `LDFLAGS` and `CPPFLAGS` environment variables. The `LDFLAGS` variable allows you to tell the linker where the library is installed by adding the following parameter: *-L<lib-dir>*. The `CPPFLAGS` variable tells the C preprocessor and the C compiler that the headers can be found in the directory specified with the following parameter: *-I<include-dir>*. For instance, to inform the **./configure** script that you have installed the library under `/home/user/projects`:

```
[cbgp-x.y.z]$ LDFLAGS=-L/home/user/projects/lib \
              CPPFLAGS=-I/home/user/projects/include \
              ./configure --prefix=/home/user/projects
[cbgp-x.y.z] make
[cbgp-x.y.z] make install
```

1.4. Additional requirements

You must install the **GNU readline** library and its headers prior to building **C-BGP** if you want to use its interactive mode. The **GNU readline** library and headers are freely available from `http://www.gnu.org`

Chapter 2

Commands reference

2.1. Command-line usage

The C-BGP simulator usage is as follows:

cbgp [**-h**] [**-l** *log*] [**-i**] [**-c** *script*]

With the options being

- **-h** display the command-line options of C-BGP.
- **-i** run the simulator in interactive mode. In this mode, commands can be given from the console. If a command fails, the simulator does not stop but reports a complete error message. Note: the simulator must be compiled with readline to support this mode.
- **-l** *LOGFILE* output the log messages to *LOGFILE* instead of stderr.
- **-c** *SCRIPT* load and execute *SCRIPT* file. (without this option, commands are taken from stdin)

2.2. Environment variables

Since version 1.1.18, **C-BGP** supports two environment variables. These environment variables control how the history of the command-line interface is stored in a file. If the `CBGP_HISTFILE` is set, **C-BGP** will load the historic of commands from a file named `~/.cbgp_history`. If `CBGP_HISTFILE` is not empty, the default file name is replaced by the environment variable's value.

In addition, the value of the `CBGP_HISTFILESIZE` can be set in order to limit the number of lines that will be loaded from the history file. The value of `CBGP_HISTFILESIZE` must be a positive integer value.

2.3. Scripting

In **C-BGP**, simulations are configured through scripts. A script is a sequence of **C-BGP** commands that are used to build the topology by adding nodes and links, to setup BGP sessions and to record routing information. The available **C-BGP** commands are shortly described in the following section. Before writing scripts, let's learn some particular features of the **C-BGP** scripting interface.

First, commands are grouped into functionnal classes. The **net** class contains commands related to the network and the IP layer. The commands in this class are used to build a topology of nodes and links but also to change the IP routing table of nodes, to trace the route from one node to another or even to add IP-in-IP tunnels. The **bgp** class contains commands related to the BGP protocol. The commands in this class are used to enable the BGP protocol on a particular node, to advertise local networks, to configure

BGP peerings, and so on. The **sim** class contains the simulator related commands, that is commands that are used to run/stop the simulator. Finally, some commands do not belong to any of the above classes because they are general purpose commands that serve to print a message or to include a subscript.

Second, some commands are composed of a context part. That is a part of the command can be used alone to change the current command context. Let's clarify this with an example. The **bgp router X add network Y** is composed of the context part **bgp router X** which changes the command context to the commands available in router X. If the context part of the command is executed alone, the only available commands will be commands that start with the current context.

```
bgp router X
        add network Y
        add peer Z1 Z2
        ...
```

is thus equivalent to the command **bgp router X add network Y** followed by the command **bgp router X add peer Z1 Z2**.

In order to exit the current context, type the **exit** command. The parent context is restored. It is also possible to exit all the nested contexts by typing an empty command line.

2.4. Commands

This section describes all the available C-BGP commands. The commands are grouped into four main groups: **net**, **bgp**, **sim** and a group with miscellaneous commands.

2.4.1. Network related commands

2.4.1.1. **net add node** *address*

This command adds a new node to the topology. The node is identified by its IP address. This address must be unique. When created, a new node only supports IP routing as well as a simplified ICMP protocol. If you want to add support for the BGP protocol, consider using the **bgp add router** command.

2.4.1.2. **net node** *address* **ipip-enable**

This command enables the support for the IP-in-IP protocol. That means that this node can behave as a tunnel end-point. If it receives encapsulated packets with the destination address of the encapsulatin header being itself, it will decapsulate the packet and deliver it locally or try to forward it depending on the encapsulated header content.

2.4.1.3. **net node** *address* **ping** *destination*

2.4.1.4. **net node** *address* **record-route** *destination*

This command records the addresses of the nodes that packet sent from the source *address* traverse to reach the *destination* address.

▶ Output format

```
<source> <destination> <result> <list of hops>
```

where *result* is one of

SUCCESS	The destination was reachable. In this case, the list of hops is the list of the IP adresses of the traversed nodes.
UNREACH	The destination was not reachable. In this case, the list of hops is the list of IP addresses of the nodes traversed until no route was available.
TOO_LONG	The path towards the destination was too long (i.e. longer than 30 hops). This is often the symptom of a routing loop.
TUNNEL_UNREACH	The path went through a tunnel but the tunnel end-point does not support the IP-in-IP protocol. Consider using the **net node** *X* **ip-ip enable**. The last node in the list of hops is the address of the faulty node.
TUNNEL_BROKEN	The path went through a tunnel but at a point the tunnel end-point was not reachable. The last node in the list of hops is the address of the faulty node.

▶ **Example**

```
cbgp> net node 0.1.0.1 record-route 0.2.0.2
0.1.0.1 0.2.0.2 SUCCESS 0.1.0.1 0.1.0.2 0.2.0.1 0.2.0.2
```

2.4.1.5. **net node** *address* **route add** *prefix next-hop metric*

This command is used to add a route towards a *prefix* into the node identified by *address*. The command specifies the route's *next-hop* and the route's *metric*.

Note. It is often more convenient to use the **net node** *X* **spf-prefix** command which computes for each node within a given prefix the shortest route according to the used metric.

2.4.1.6. **net node** *address* **route del** *prefix* **{** *next-hop* | ***** **}**

This command removes from the node identified by *address* a route previously added with the above command. The route to be removed is identified by the destination *prefix* as well as its *next-hop*. A wildcard can be used in place of the *next-hop*. In this case, all the routes that match the *prefix* will be removed.

2.4.1.7. **net node** *address* **show links**

This command shows all the links connected to this node.

▶ **Output format**

```
<prefix> <delay> <metric> <state> <type> [<IGP option>]
```

where the *state* can be either **UP** or **DOWN**. The *type* is one of

DIRECT	The link is a direct link towards the destination, i.e. the destination is adjacent to this node.
TUNNEL	The link is a tunnel to the destination, i.e. messages that traverse this link will be encapsulated, then routed towards the tunnel end-point and hopefully decapsulated there.

and the *IGP option*, if present, can contain the **IGP_ADV** flag which means that this link is used by the "IGP", i.e. it is used in the shortest-path computation performed by the **net node** *X* **spf-prefix** command.

▶ **Example**

```
cbgp> net node 0.0.0.1 show links
0.2.0.1/32      444     444     UP      DIRECT  IGP_ADV
0.2.0.2/32      370     370     UP      DIRECT  IGP_ADV
```

2.4.1.8. **net node** *address* **show rt {** *address* | *prefix* | * **}**

This command shows the content of the routing table of node *address*. The command takes one parameter to filter the output. If the filter parameter is *, all the routes are shown. If the filter parameter is an IP address, the best route that matches the given address is shown. If the filter parameter is an IP prefix, the exact route that matches the given prefix is shown.

▶ **Output format**

```
<prefix> <next-hop> <metric> <type>
```

where the route's *type* can be one of

STATIC	The route was statically installed with the **net node** *X* **route add** command.
IGP	The route was automatically computed by the **net node** *X* **spf-prefix** command.
BGP	The route was learned by BGP and selected as best.

▶ **Example**

```
cbgp> net node 0.0.0.1 show rt *
0.1.0.1/32      0.2.0.2 0       BGP
0.2.0.1/32      0.2.0.2 0       BGP
0.2.0.2/32      0.2.0.2 0       BGP
```

2.4.1.9. **net node** *address* **spf-prefix** *prefix*

This command computes the shortest paths from the node identified by *address* towards all the nodes which are in the given *prefix*. The metric of the computed shortest paths is equal to the sum of the IGP weights of the traversed links. The command also adds in the node's routing table an entry for each computed path. These routing entries are of type IGP.

SPT computation. The shortest paths will only be composed of links whose end-points are in the given *prefix* and which have the **IGP_ADV** flag set (see the **net node** *X* **show links** command for more information).

Note that the behaviour of this command can be slightly modified with the **igp-inter** option. See the **net options igp-inter** statement.

2.4.1.10. **net node** *address* **tunnel add** *end-point*

This command adds a tunnel from the given node towards the tunnel *end-point*. Messages that will be routed through this tunnel will be encapsulated, then routed to the *end-point* and decapsulated at the *end-point*. Consider using the **net node** *X* **ip-ip enable** on the destination node to enable the decapsulation of received messages.

2.4.1.11. **net options igp-inter [on | off]**

This command changes the destination nodes that are considered by the SPT computation performed by the **spf-prefix** statement. If the option value is **off** (default), the SPT computation only considers as destinations the nodes which strictly match the given prefix. If the option is **on**, the SPT computation also considers as destinations the nodes which are tail-ends of links that leaves the nodes which match the given prefix.

This is illustrated in Fig. 2.1. In the left part of the figure, the **igp-inter** option is **off** and the nodes which are tail-ends of the links that go outside of the considered prefix are not considered by the SPT-computation. In the right part, those destinations are taken into account during the SPT computation and routes are setup for these destinations in the nodes that belong to the prefix.

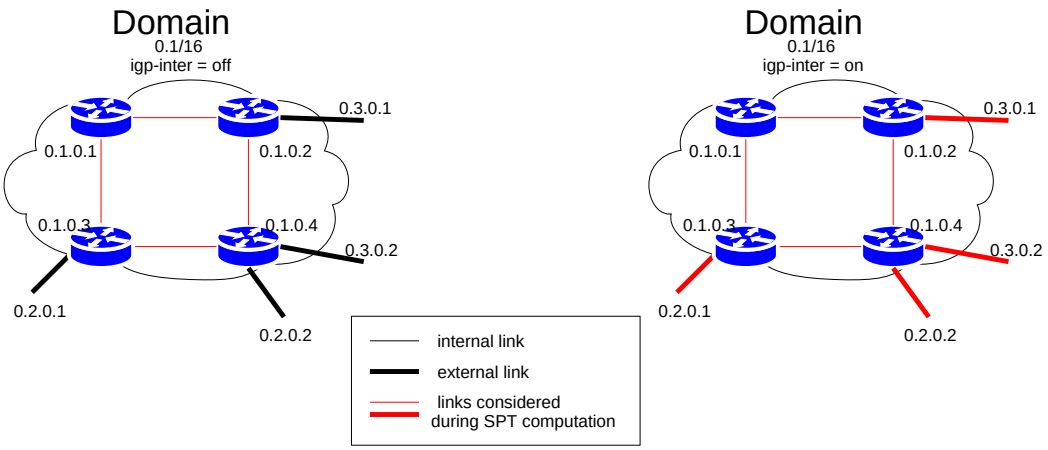

Figure 2.1: Effect of the **igp-inter** option on SPT computation.

2.4.1.12. **net options max-hops** *max-hops*

This command changes the maximum number of hops used by the record-route command. The default value is 15 hops.

2.4.1.13. **net add link** *address1 address2 delay*

This command adds a new link between two existing nodes whose addresses are *address1* and *address2* in the topology. The new link is bidirectional. The propagation delay of the link is specified by the *delay* parameter. Note also that by default, the IGP-cost of the link is fixed at the same value

2.4.1.14. **net link** *address1 address2* **[down | up]**

2.4.1.15. **net link** *address1 address2* **igp-weight** *weight*

This command changes the IGP weight of the link identified by the two end-points *address1* and *address2*.

Warning. This command changes the IGP weight of the link in one direction only. If you use different weights for both directions and if you use the **net node** *X* **spf-prefix** command, routing loops may be created.

2.4.2. BGP related commands

2.4.2.1. **bgp assert peerings-ok**

This command checks that all the peerings defined in all the routers are valid, i.e. for each peering, the existence of the peer is checked as well as the existence of a similar peering definition in the peer.

2.4.2.2. **bgp assert reachability-ok**

This command checks that all the prefixes announced by BGP are reachable by all the BGP routers.

2.4.2.3. **bgp add router** *as-num address*

This command adds BGP support into the node identified by *address*. The node thus becomes a BGP router. The command also configures this router as a member of the domain identified by *as-num*.

2.4.2.4. **bgp options local-pref** *pref*

This command changes the default preference given to routes that enter a domain. The default preference is 0, but can be changed to *local-pref* using this command.

2.4.2.5. **bgp options med** *"deterministic"* | *"always-compare"*

This command changes the behaviour of the MED-based rule of the decision process. If the argument is "deterministic", then the rule will only compare the MED of routes received from the same AS. If the argument is "always-compare", the rule will compare the MED of all routes whatever the neighbor AS is.

2.4.2.6. **bgp options msg-monitor** *out-file*

This command enables the BGP message monitoring. All the BGP messages will be written in the given *out-file*. If the file does not exist, it will be created. If it already exists, the file will be overwritten. The BGP messages will be written in MRTD format, prefixed by the IP address of the destination router.

▶ **Output format (UPDATE)**

```
dest-ip|BGP4|time|A|peer-ip|peer-as|prefix|
as-path|origin|next-hop|local-pref|med|communities
```

▶ **Output format (WITHDRAW)**

```
dest-ip|BGP4|time|W|peer-ip|peer-as|prefix
```

It should be easy to extract messages sent to a specific destination on the basis of the first field. Then, existing analysis script can be used with the MRTD output.

▶ **Example**

```
0.2.0.0|BGP4|0.00|A|0.1.0.0|1|0.1.0.0/16|1|IGP|0.1.0.0|100|0|
```

2.4.2.7. **bgp router** *address* **add network** *prefix*

This command adds a local network that will be advertised by this router. The given network will be originated by this router.

2.4.2.8. **bgp router** *address* **del network** *prefix*

This command removes a local network previously added with the above command.

2.4.2.9. **bgp router** *address* **add peer** *as-num peer-address*

This command adds a new BGP neighbor to the router identified by *address*. The peer belongs to the domain identified by *as-num* and is identified by *peer-address*. This command also configures for this neighbor default input and output filters that will accept any route. See the **bgp router** *X* **peer** *Y* **filter** commands for more information about the route filters.

2.4.2.10. **bgp router** *address* **peer** *peer-address* **[down | up]**

This command starts the establishment of the BGP session with the peer identified by *peer-address* (when used with **up**) or shuts down the previously established session (when used with **down**).

2.4.2.11. **bgp router** *address* **peer** *peer-address* **filter [in | out] add-rule**

This command adds a rule to the input (**in**) or output (**out**) filter of the peer identified by *peer-address*. See chapter 3 for more information about filters.

2.4.2.12. **bgp router** *address* **peer** *peer-address* **filter [in | out] insert-rule** *index*

This command inserts a rule at position *index* to the input (**in**) or output (**out**) filter of the peer identified by *peer-address*. See chapter 3 for more information about filters.

2.4.2.13. **bgp router** *address* **peer** *peer-address* **filter [in | out] remove-rule** *index*

This command removes the rule at position *index* from the input (**in**) or output (**out**) filter of the peer identified by *peer-address*. See chapter 3 for more information about filters.

2.4.2.14. **bgp router** *address* **peer** *peer-address* **filter [in | out] show**

This command shows the rules that compose the input (**in**) or output (**out**) filter of the peer identified by *peer-address*. See chapter 3 for more information about filters.

2.4.2.15. **bgp router** *address* **peer** *peer-address* **next-hop-self**

2.4.2.16. **bgp router** *address* **peer** *peer-address* **recv** *message*

This command can be used on virtual peers to feed real BGP messages. The messages must be provided in MRT format. Only UPDATE and WITHDRAW messages are accepted.

2.4.2.17. **bgp router** *address* **peer** *peer-address* **reset**

2.4.2.18. **bgp router** *address* **peer** *peer-address* **rr-client**

This command configures the peer identified by *peer-address* as a route-reflector client. By the way, the router identified by *address* becomes a route-reflector.

2.4.2.19. **bgp router** *address* **peer** *peer-address* **virtual**

This command changes the peer into a virtual peer. A virtual peer is used to feed the router with real BGP messages in MRT format. The router will not maintain a BGP session with the virtual peer. Moreover, the router will not send BGP messages to the virtual peer. The virtual peer will only accept UPDATE and WITHDRAW messages with the help of the "recv" statement.

2.4.2.20. **bgp router** *address* **record-route** *prefix*

This command records the AS-level route from the given router identified by *address* towards the given *prefix*.

▶ **Output format**

```
<source> <destination> <status> <AS-Path>
```

▶ **Example**

```
cbgp> bgp router 0.0.0.1 record-route 0.1.0.1/32
0.0.0.1 0.1.0.1/32      SUCCESS 0 2 1
```

2.4.2.21. **bgp router** *address* **set cluster-id** *id*

2.4.2.22. **bgp router** *address* **show peers**

This command shows the list of peers of the router identified by *address*. For each peer, the configured AS-number and session state are shown.

▶ **Output format**

```
<peer> <as-num> <status>
```

▶ **Example**

```
cbgp> bgp router 0.0.0.1 show peers
0.2.0.1 AS2     ESTABLISHED
0.2.0.2 AS2     ESTABLISHED
```

2.4.2.23. **bgp router** *address* **show rib {** *address* | *prefix* | * **}**

This command shows the routes installed into the Loc-RIB of the router identified by *address*. The command takes one parameter to filter the output. If the filter parameter is *, all the routes are shown. If the filter parameter is an IP address, the best route that matches the given address is shown. If the filter parameter is an IP prefix, the exact route that matches the given prefix is shown.

▶ **Output format**

```
<flags> <prefix> <peer> <pref> <metric> <AS-Path> <origin>
```

where the *flags* can contain

*****	If the route's next-hop is reachable.
>	If the route is a best route and is installed in the Loc-RIB.
i	If the route is local, i.e. installed with the **add network** command.

moreover, the *origin* is one of

i	If the origin router learned this route through a **add network** statement.
e	If the origin router learned this route from an EGP protocol.
?	If the origin router learned this route through redistribution from another protocol.

Since the only way to learn a BGP route in **C-BGP** is through the **add network** statement, the *origin* will always be **i**.

▶ **Example**

```
cbgp> bgp router 0.0.0.1 show rib *
i> 0.0.0.1/32   0.0.0.1 0       0       null    i
*> 0.1.0.1/32   0.2.0.2 0       0       2 1     i
*> 0.2.0.1/32   0.2.0.2 0       0       2       i
*> 0.2.0.2/32   0.2.0.2 0       0       2       i
```

2.4.2.24. **bgp router** *address* **show rib-in {** *peer* | * **} {** *address* | *prefix* | * **}**

This command shows the routes received from the selected peers of the router identified by *address*. Thus, this command shows the content of the Adj-RIB-Ins of the given router. The command takes two parameters to filter the output. The first parameter filters the peers whose Adj-RIB-Ins are shown. The parameter can be the IP address of the peer or * to show all the Adj-RIB-Ins. The second parameter can be * to show all the routes. It can also be an IP address to show only the best route that matches the given address. Finally, it can be an IP prefix. In this case, the exact route that matches the given prefix is shown.

▶ **Output format**

```
<flags> <prefix> <peer> <pref> <metric> <AS-Path> <origin>
```

(see the **show rib** command for more information on the fields).

▶ **Example**

```
cbgp> bgp router 0.0.0.1 show rib-in * *
*  0.1.0.1/32   0.2.0.1 0       0       2 1     i
*  0.2.0.1/32   0.2.0.1 0       0       2       i
*  0.2.0.2/32   0.2.0.1 0       0       2       i
*> 0.1.0.1/32   0.2.0.2 0       0       2 1     i
*> 0.2.0.1/32   0.2.0.2 0       0       2       i
*> 0.2.0.2/32   0.2.0.2 0       0       2       i
```

2.4.2.25. **bgp topology load** *file*

This command loads a topology from the specified *file*. The format of the file is similar to the AS pair relationships file specified at this address. That is each line of the file specifies a relationship between two Internet domains. Based on this file, C-BGP builds a network where each domain is composed of a unique router having the IP address equal to the domain's number (AS-NUM) multiplied by 65536. For instance, the IP address of the router which composes the domain AS7018 would be 27.106.0.0. C-BGP also configures the BGP sessions between the network's routers.

▶ **Input format**

```
<domain-1> <domain-2> <relationship> [<delay>]
```

The relationship can be *0* for a peer-to-peer relationship or *1* for a provider-to-customer relationship. The optional *delay* parameter specifies the delay on the network link between the routers in the given domains.

2.4.2.26. **bgp topology policies**

This command configures the filters of the routers according to the relationships specified in the topology loaded by the **bgp topology load** command.

2.4.2.27. **bgp topology record-route** *prefix in-file out-file*

This command records the paths towards the given *prefix* from each router specified in the *in-file* and writes those paths in the *out-file*. The *in-file* has the following format: each line contains the identifier of a domain (i.e. its AS-NUM) from which the path has to be computed or an asterisk (*) which means that the paths from all the routers have to be computed.

▶ **Output format**

```
<src-as-num> <prefix> <as-path>
```

▶ **Example** For instance, here is the result of the EBGP_2_ROUTERS example. The path from AS1 is "1" because it has advertised the prefix 0.1/16. The path from AS2 is "2 1" because it has received a BGP message with the prefix 0.1/16 from AS1.

```
1 0.1.0.0/16 1
2 0.1.0.0/16 2 1
```

2.4.2.28. **bgp topology run**

This command establishes the BGP sessions between all the routers loaded by the bgp topology load command.

2.4.3. Simulation related commands

2.4.3.1. **sim options log-level <level>**

This command changes the verbosity of the simulator log. The available log levels are

everything	Every log message is written.
debug	Debug and error messages are written.
warning	Error messages are written.
severe	Only severe warning and fatal error messages are written.
fatal	Only fatal error messages are written.

2.4.3.2. **sim options scheduler <scheduler>**

2.4.3.3. **sim run**

This command starts the simulator, i.e. its starts processing the queued events until no more event is available or the simulator is stopped.

2.4.3.4. sim stop at <time>

2.4.4. General purpose commands

2.4.4.1. include <file>

This command processes the commands found in the given *file*.

2.4.4.2. print <message>

This command prints a *message* on the current output. The default output is stdout.

Chapter 3

Filters

3.1. Introduction

This chapter describes the route filtering features of **C-BGP**. Route filtering is an important part of the simulator since it is used to implement the policies of interdomain routing. **C-BGP** provides an easy to use interface to filters similar to what can be found in real routers.

In **C-BGP**, the filters can be configured differently in each router. Moreover, for each BGP router different filters can be associated to each neighbor. We also distinguish input and output filters. The first ones are used to filter the routes that are learned from neighbors while the second ones are used to filter the routes that are redistributed to neighbors.

A typical filter in **C-BGP** is a sequence of rules. Each rule being composed of two parts. The first part of one rule is a logical combination of predicates used to check if the rule applies to a route. For instance, a predicate can check is the tested route contains a given community. The second part of one rule is a set of actions that are applied to the routes matching the rule's predicates. A typical action would be to change the route's local preference.

In **C-BGP**, the filters of one router are configured using the **peer** *X* **filter [in | out]** familly of commands. The list of these commands is given in section 2.4.2. The **add-rule** sub-command allows to add a new rule to the given filter. Once the rule is added, the predicates and actions can be specified with the commands described in the sections hereafter. Another sub-command makes possible to insert a new rule in the sequence of rules of one filter. It is of course always possible to show the sequence of rules of one filter with the **show** sub-commands.

3.2. Predicates

Once a new rule is added or inserted, the predicates can be specified with the **match** statement. This statement takes a single parameter which is the expression of the logical combination of predicates. The syntax of this expression is described in Fig. 3.1:

Predicates	::=	Predicate
	|	Predicates '|' Predicates
	|	Predicates '&' Predicates
	|	'(' Predicates ')'
	|	'!' Predicates

Figure 3.1: Syntax of predicates.

The atomic predicates that are currently available in **C-BGP** are described below:

3.2.0.3. **any**

This predicate matches any route.

3.2.0.4. **community is** *community*

This predicate matches only the routes that contain the given *community*. The community can be written in two forms. The first form is as an integer in the range $[0, 2^{32} - 1]$. The second form is as a couple of integers in the range $[0, 2^{16} - 1]$ separated by a colon (':').

3.2.0.5. **prefix is** *prefix*

This predicate matches only the routes with a prefix which is equal to the given *prefix*.

3.2.0.6. **prefix in** *prefix*

This predicate matches only the routes with a prefix which is more specific than the given *prefix*.

3.3. Actions

In order to specify the second part of one rule, **C-BGP** also provides the **actions** statement. This statement takes a single parameter which is a set of atomic actions separated by a comma. The actions that are currently available in **C-BGP** are described below:

3.3.0.7. **accept**

This action accepts the route.

3.3.0.8. **as-path prepend** *amount*

This action prepends the AS-number of the router to the AS-Path of the route. The number of times prepending is performed is given by *amount*.

3.3.0.9. **community add** *community*

This action adds the given community to the list of communities of the route. The *community* can be specified either as a single 32-bits integer or as a couple of two 16-bits integers separated by a colon.

Special standard communities may also be specified with special identifiers: *no-export* or *no-advertise*.

3.3.0.10. **community strip**

This command clears the list of communities attached to the route.

3.3.0.11. **deny**

This commands deny the route. If this action occurs in an input filter, the route will not be considered as feasible. If this action occurs in an output filter, the route will not be redistributed to the peer concerned by the filter.

3.3.0.12. **local-pref** *pref*

This command changes the local-preference of the route. The new value of the LOCAL-PREF attribute is set to *pref*. Note that this command should only be used in input filters associated with peers that are in another domain (i.e. that have a different AS-number).

3.3.0.13. **metric** *med* | *"internal"*

This command changes the MED of the route. The value can be specified explicitly by passing it as an integer value to the command. Another way to set the MED value is to specify the special-value "internal". In this case, the MED value is set to the cost of the IGP path towards the route's next-hop.

3.3.0.14. **red-community add prepend** *amount target*

This command attaches a redistribution community to the route. This redistribution community requests that the neighbor domain perform prepending when it redistributes the route to the domain identified by *target*.

3.3.0.15. **red-community add ignore** *target*

This command attaches a redistribution community to the route. This redistribution community requests that the neighbor domain does not redistributes the route to the domain identified by *target*.

3.4. Example

In order to illustrate the above obscure explanations, this section presents an example of filters. The purpose of this example is to define the filters required to enforce the Internet policies, that is the customer-provider and peer-to-peer relationships.

These policies are composed of two parts. First, the filters must control the provision of transit. The usual rule is that a domain will not provide transit to its providers, a limited transit to its peers and full connectivity to its customers. This is known as the *valley-free* property.

Second, the domains usually prefer the routes learned from customers over the routes learned from providers. The last routes being also prefered over routes learned from providers. One of the reason is that such an ordering assures the convergence of BGP. Another reason is that domains get paid for the traffic that transit on the links with their customers while they must pay for the traffic that transit over links with their providers.

Such policies are easy to setup within **C-BGP**. Let's take the example topology shown in Fig. 3.2. The topology is composed of 4 domains. Domain AS1 is composed of 3 routers, R11, R12 and R13. The first router, R11, is connected to R21, the router of its provider, AS2. R12 is connected to R31, the router of its peer, AS3. Finally, R13 is connected to R41, the router of its customer, AS4.

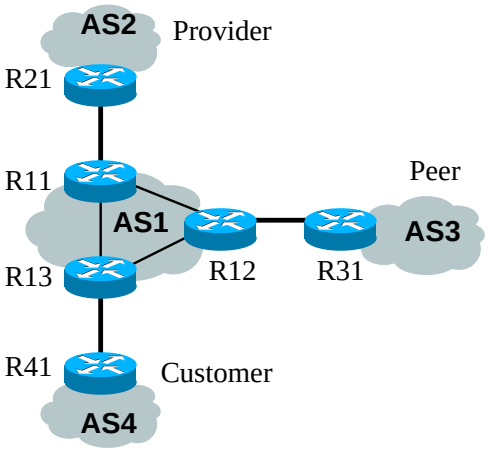

Figure 3.2: Example topology with business relationships.

The following scripts show how the various peerings are setup. Note that for convenience, the script does not contain the IP addresses of the router but their names.

```
bgp router R11
        peer 1 R12
        peer 1 R13
        peer 2 R21
        peer R21
                filter in
                        add-rule
                                match any
                                action "local-pref 60, community add 1"
                                exit
                        exit
                filter out
                        add-rule
                                match "community is 1"
                                action deny
                                exit
                        add-rule
                                match any
                                action "community strip"
                                exit
                        exit
                exit

bgp router R12
        peer 1 R11
        peer 1 R13
        peer 3 R31
        peer R31
                filter in
                        add-rule
                                match any
                                action "local-pref 80, community add 1"
                                exit
                        exit
                filter out
                        add-rule
                                match "community is 1"
                                action deny
                                exit
                        add-rule
                                match any
                                action "community strip"
                                exit
                        exit
                exit

bgp router R13
        peer 1 R11
        peer 1 R12
        peer 4 R41
        peer R41
```

```
filter in
        add-rule
                match any
                action "local-pref 100"
                exit
        exit
filter out
        add-rule
                match any
                action "community strip"
                exit
        exit
exit
```

Chapter 4

Examples

4.1. Simple two-routers topology

This example describes how to build a simple topology composed of two BGP routers in two different domains. The first step is to create the nodes which correspond to the routers and the link which connects them together.

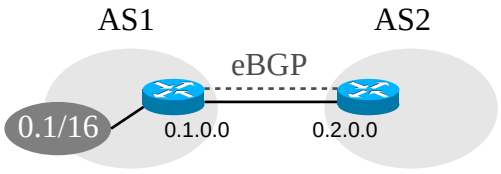

Figure 4.1: Simple two-routers topology

Building the topology is done by using the **net add** commands as explained below. The **net node X route add** statement adds static routes over the interdomain link.

```
# Setup the topology
net add node 0.1.0.0
net add node 0.2.0.0
net add link 0.1.0.0 0.2.0.0 5
net node 0.1.0.0 route add 0.2.0.0/32 0.2.0.0/32 5
net node 0.2.0.0 route add 0.1.0.0/32 0.1.0.0/32 5
```

Then, BGP has to be enabled on both nodes with the **bgp add** command. Moreover, each router has its neighbors configured and finally router 0.1.0.0 will announce a local network with BGP.

```
# Setup BGP in router 0.1.0.0
bgp add router 1 0.1.0.0
bgp router 0.1.0.0
add network 0.1/16
```

```
add peer 0.2.0.0
peer 0.2.0.0 up

# Setup BGP in router 0.2.0.0
bgp add router 2 0.2.0.0
bgp router 0.2.0.0
add peer 0.1.0.0
peer 0.1.0.0 up
```

Finally, the simulation is started with the **sim run** command. After the simulation has converged, the BGP routing tables of both routers can be dumped.

```
# Run the simulation
sim run

# Dump both router's routing table
print "Routing table of router 0.1.0.0\n"
bgp router 0.1.0.0 show rib *
print "Routing table of router 0.2.0.0\n"
bgp router 0.2.0.0 show rib *
```

4.2. eBGP and iBGP sessions

This example describes a somewhat more complicated configuration where 4 routers are involved. A first domain, AS1, contains a single router, 0.1.0.0 which advertises a single network 0.1/16. The second domain, AS2, contains 3 routers, 0.2.0.0, 0.2.0.1 and 0.2.0.2. There is an iBGP session between routers 0.2.0.0 and 0.2.0.2. Since there is no direct physical link between 0.2.0.0 and 0.2.0.2, we will add static routes in both routers.

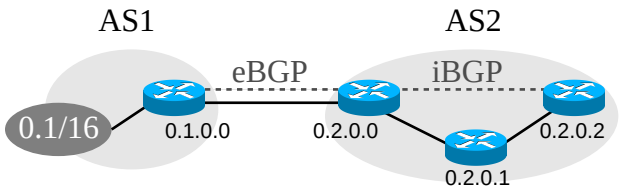

Figure 4.2: Simple topology with eBGP and iBGP sessions

So, the first step consists in building the topology:

```
# Build the topology
net add node 0.1.0.0
net add node 0.2.0.0
net add node 0.2.0.1
net add node 0.2.0.2
net add link 0.1.0.0 0.2.0.0 20
net node 0.1.0.0 route add 0.2.0.0/32 0.2.0.0 20
```

```
net node 0.2.0.0 route add 0.1.0.0/32 0.1.0.0 20
net add link 0.2.0.0 0.2.0.1 5
net node 0.2.0.0 route add 0.2.0.1/32 0.2.0.1 5
net node 0.2.0.1 route add 0.2.0.0/32 0.2.0.0 5
net add link 0.2.0.1 0.2.0.2 5
net node 0.2.0.1 route add 0.2.0.2/32 0.2.0.2 5
net node 0.2.0.2 route add 0.2.0.1/32 0.2.0.1 5
```

Then, we must add in nodes 0.2.0.0 and 0.2.0.2 a route to each other that goes through node 0.2.0.1. This is done with the **net node X route add** command.

```
# Add static routes between 0.2.0.0 and 0.2.0.2
net node 0.2.0.0 route add 0.2.0.2/32 0.2.0.1 10
net node 0.2.0.2 route add 0.2.0.0/32 0.2.0.1 10
```

Finally, the BGP protocol is enabled in routers 0.1.0.0, 0.2.0.0 and 0.2.0.2. The BGP peerings are configured and a single network is advertised by 0.1.0.0.

```
# Setup BGP in router 0.1.0.0
bgp add router 1 0.1.0.0
bgp router 0.1.0.0
add network 0.1/16
add peer 2 0.2.0.0
peer 0.2.0.0 up

# Setup BGP in router 0.2.0.0
bgp add router 2 0.2.0.0
bgp router 0.2.0.0
add peer 1 0.1.0.0
peer 0.1.0.0 next-hop-self
add peer 2 0.2.0.2
peer 0.1.0.0 up
peer 0.2.0.0 up

# Setup BGP in router 0.2.0.2
bgp add router 2 0.2.0.2
bgp router 0.2.0.2
add peer 2 0.2.0.0
peer 0.2.0.0 up
```

4.3. Domains and SPT computation

In the above example, we have added two static routes between node 0.2.0.0 and node 0.2.0.2. These routes were easy to add but when the topology becomes large, configuring static routes can become tedious. Fortunately, **C-BGP** provides an alternative to a manual route setup: a shortest path tree (SPT) computation. It is possible to compute the shortest-path from one node to a group of other nodes and automatically setup the required routes. Today, the only way to specify the group of destination nodes is through a network prefix, that is a prefix specifies the group of all nodes which have an IP address that matches the prefix. Usually, the prefix will cover the whole domain to which the SPT root node belongs. Indeed, an hierarchical addressing scheme must be used in order to be able to use this facility.

The command to use for the purpose of computing the shortest path tree is **net node X spf-prefix P**. The command computes the SPT rooted at node X to all the nodes in prefix P. The statements used in the above example (4.2) for the purpose of setting up static routes between each pair of nodes in the same domain, can thus be replaced by the following statements. In the case of large domains, it is a far more straightforward manner to configure the intradomain routes.

```
net node 0.2.0.0 spf-prefix 0.2/16
net node 0.2.0.1 spf-prefix 0.2/16
net node 0.2.0.2 spf-prefix 0.2/16
```

The behaviour of the **spf-prefix** statement can be slightly altered with the use of the **igp-inter** option. This option can take the values **on** and **off** (default) with the **net options igp-inter** statement. If the option value is **off**, then the SPT computation will only consider as destinations the internal nodes. If the option value is **on**, then the SPT computation will also consider as destinations the nodes which are tail-ends of interdomain links.

To clarify this, let's take the example network shown in Fig. 4.3. The network contains three domains (which can be different ASes). A typical configuration of this topology will require running an IGP inside each domain, which is modelled by the SPT computation, and BGP to provide reachability between the domains. However, in order for BGP to be run, it is required for border routers to know a route towards the tail-end of interdomain links. There are two ways to do this. The first possibility relies on the setup of static routes on the external links. This is implemented in **C-BGP** with the help of the **net node X route add** statement. In addition, eBGP sessions must be configured with the **next-hop-self** statement.

The second possibility is to insert the tail-ends of the external links in the SPT computation. It is implemented with the help of the SPT computation and the **igp-inter** option being **on**. In this case, using **next-hop-self** on eBGP sessions is useless.

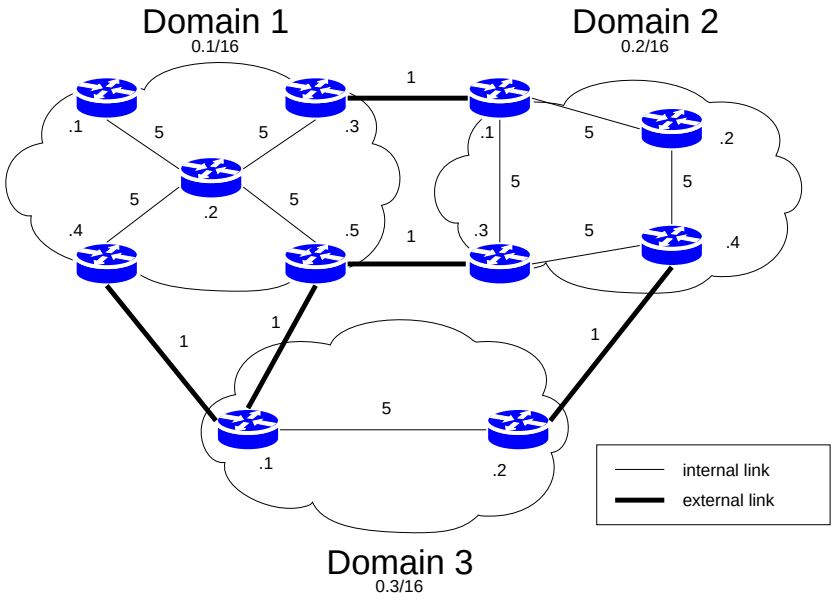

Figure 4.3: Three domains interconnected by external links.

4.4. Multi-Exit-Discriminator

In this example, we show how an AS can be configured in order to advertise routes with the MED value set based on the IGP cost to the next-hop. We also illustrate how the MED-based rule of the decision process can be changed in order to allow the comparison of the MED value of routes received from different neighbor ASes, an option found in commercial routers under the name *always-compare*.

The example topology we use is shown in Fig. 4.4. The topology contains 4 different ASes. Each AS is associated with a network with a /16 mask. For instance, AS1 is the network 0.1/16; AS2 is the network 0.2/16 and so on. There are multiple peerings between those 4 ASes. A single prefix 0.1/16 is advertised

by AS1 and we observe how it is propagated until AS4, and in particular, until router 0.4.0.5 (the lowest router in the Fig. 4.4). We do not show in this example the script required to setup the different networks and BGP sessions. However, the IGP weights are shown beside the links in Fig. 4.4.

Let's now focus on the MED utilization. In our example, AS2 and AS3 advertises to AS4 routes towards AS1's prefix, 0.1/16, with the MED value set to the IGP cost towards the next-hop. In order to achieve this, we must configure output filters in the border routers of AS2 and AS3. Those output filters will contain a single rule which matches any route and whose action is to change the MED value of a matched route to the IGP cost to its next-hop. The following script shows how it is done in router 0.2.0.3 in AS2 which has a BGP session with router 0.4.0.3 in AS4. A similar configuration is made in routers 0.2.0.1, 0.3.0.1 and 0.3.0.2.

```
bgp router 0.2.0.3
        peer 0.4.0.3
                filter out
                        add-rule
                                match any
                                action "metric internal"
                                exit
                        exit
                exit
exit
```

On the other hand, we must also configure how the MED values received by routers of AS4 will be treated. The default behaviour is to only compare the MED values of two different routes if these routes have been received from the same neighboring AS. This is configured with the BGP option **bgp options med deterministic**. This statement must be issued before any decision is made by BGP, that is, before the statement **sim run** is issued to the simulator.

Another behaviour is possible for the MED-based rule with the statement **bgp options med always-compare** which means that the MED values of the routes are compared whatever the announcing AS was.

On Fig. 4.4, we show in green the route used by router 0.4.0.5 to reach destination prefix 0.1/16 when the MED-based rule only compares the MED of routes received from the same neighbor AS. When the *deterministic* mode is used (green route), the egress is 0.4.0.1. The reason for this is that routes are grouped by neighboring AS before their MED values are compared. When comparing the MED values of routes received from AS2, the one from received from 0.2.0.3 is prefered (the MED value is 1 while the MED value of the route received from 0.2.0.1 is 15). When comparing the MED values of routes received from AS3, both routes are kept because they have the same MED value, that is 5. There are thus 3 routes remaining and the decision process then relies on the router-ID. The lowest router ID is 0.4.0.1 an the route received from this router is thus prefered.

We show in brown the route used by router 0.4.0.5 when the MED-based rule compares all routes. When the *always-compare* mode is used, the MED values of the 4 routes received from both AS2 and AS3 are compared and the route received from router 0.2.0.3, which has the lowest MED value, is selected as best.

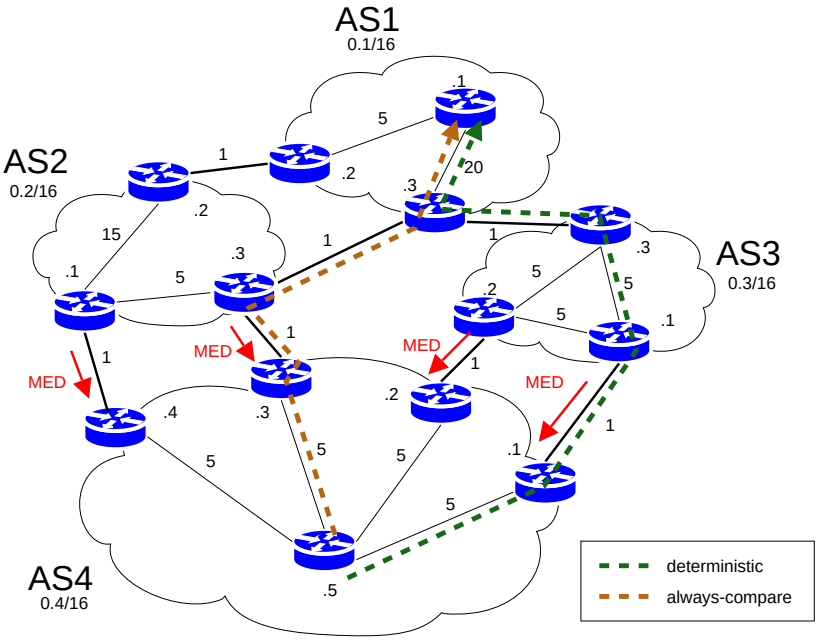

Figure 4.4: Example topology where MED is advertised.

Appendix A

Perl interface

In order to ease the development of applications or scripts that interact with **C-BGP**, a **Perl** interface is provided. This interface handles the dialog between a **Perl** script and a **C-BGP** instance. The **Perl** interface comes as a module that has to be imported in the **Perl** script. A small set of methods is defined in the module in order to establish the dialog between the **Perl** script and the **C-BGP** instance as well as to send and receive messages to and from the **C-BGP** instance.

The **Perl** interface module works as follows. First, it runs the **C-BGP** simulator and sets up new file descriptors in order to be able to write to **C-BGP**'s standard input and read from its standard output. The **Perl** module relies on a separate thread to manage the communication between the **Perl** script and the **C-BGP** instance for the purpose of avoiding buffering problems. The **Perl** script developper has thus little time to spend in order to tackle these issues. By using the methods provided by the **Perl** module, handling the interaction with **C-BGP** only takes a few lines of code.

A.1. Initialization

The following script illustrates how a basic **Perl-C-BGP** interaction can be setup. The first line imports the CBGP module with the version being at least 0.3 (the version we are talking about in this section). Note that in order to use this version of the CBGP module, we recommend that you use a version of **Perl** that is no older than 5.8.3 since previous versions suffer from various bugs related to multi-threading.

```
use CBGP 0.3;

my $cbgp= new CBGP;
$cbgp->spawn();
$cbgp->send(''set autoflush on\n'');
...
# Interaction with the C-BGP instance
...
$cbgp->finalize();
```

The second line creates an instance of the CBGP object which will be responsible for handling the interaction with the **C-BGP** instance. Then, the **spawn** method launches the **C-BGP** simulator and establishes the dialog. The third line configures **C-BGP** so that it will flush most of it output messages. This is required in order to function in an interactive manner. At this point, the script can use the **send** and **expect** methods to send/receive commands to/from the **C-BGP** instance. Finally, the last line finalize the interaction. It shuts down the thread and closes the input descriptor of the **C-BGP** instance which, as a consequence, terminates the simulator.

A.2. Interaction

Once the CBGP module has been initialized and a CBGP object has been created, the developper can send and receive messages to/from the **C-BGP** simulator. The commands that can be sent to **C-BGP** are the same commands as in **C-BGP** scripts (see Ch. 2). For instance, the following script creates a tiny topology composed of two nodes and one link. It then traces the route from one node to the other.

```
$cbgp->send(‘‘net add node 0.1.0.1\n’’);
$cbgp->send(‘‘net add node 0.1.0.2\n’’);
$cbgp->send(‘‘net add link 0.1.0.1 0.1.0.2 5\n’’);
$sbgp->send(‘‘net node 0.1.0.1 spf-prefix 0.1/16\n’’);
$sbgp->send(‘‘net node 0.1.0.2 spf-prefix 0.1/16\n’’);
$cbgp->send(‘‘net node 0.1.0.1 record-route 0.1.0.2\n’’);
my $answer= $cbgp->expect(1);
my @fields= split /\s+/, $answer, 4;
if ($fields[1] eq ’SUCCESS’) {
  print "Route: ".$fields[3]."\n";
} else {
  print STDERR "Error: could not trace the route\n";
}
```

The first five lines create two nodes, one links and initialize the routing tables of both nodes. The sixth line requests the simulator to trace the route from the first node to the other. The **Perl** script then waits for the answer with the help of the **expect** method. The parameter ’1’ given to the **expect** method means that the call is blocking, i.e. this call will block until something has been received from **C-BGP**. Once the answer has been received, the **Perl** script prints the route to the standard output. Note that the format of the answer to a **record-route** statement is described in Ch. 2.

A.3. Checkpoints

It will often be required during a dialog with the **C-BGP** simulator to receive the answer from a statement that produces an answer composed of an unknown number of lines. This is the case with commands such as **show rt** or **show rib**. The problem with these commands is that you cannot use a blocking call to the **expect** method since you do not know when the simulator has sent the whole answer.

Fortunately, a simple solution exists that makes possible to circumvent this issue, the use of checkpoints. Checkpoints are places in the dialog where a synchronization is required. For instance, when you need to know that the simulator has completed a request or when you need that the simulator signals that it has finished producing its multiple lines answer, you will use checkpoints.

The simplest way to implement checkpoints with the CBGP module is to request **C-BGP** to write to its output a message that you will be able to catch. For instance, after you have requested **C-BGP** to dump the routing table of a node, you ask it to write a message “DONE” to its output. This example is illustrated in the following excerpt of a **Perl** script.

```
$cbgp->send("net node 0.1.0.1 show rt *\n");
$sbgp->send("print \"DONE\\n\"\n");
while ((my $result= $cbgp->except(1)) ne "DONE") {
  ...
  # Process $result...
  ...
}
```

A.4. Logging

Since it will sometimes be difficult to debug the interaction between your **Perl** scripts and **C-BGP**, the module provides a simple way to log all the commands that were sent to the **C-BGP** instance. In this way, you are able to load the log file afterwards into **C-BGP** using the interactive mode in order to debug your scripts.

By default, this option is turned off. To turn it on, use the following **Perl** command: **$cbgp->log=1**. The consequence is that every subsequent command that is sent to the **C-BGP** instance will be written in the file .CBGP.pm.log in the working directory. Note that this file is removed each time the command **$cbgp->new()** is used.