CS 3700 - Networks and Distributed Systems

Project 2: Simple BGP Router

The milestone for this project is due at 11:59 pm on Sunday, February 2, 2019.
The final project is due at 11:59 pm on Friday, February 14, 2019.

Description

In this project, you will implement a simple BGP router. There are several educational goals of this project:

- To give you a sense of how core Internet infrastructure works, by giving you first-hand experience building and managing forwarding tables, generating route announcements, and forwarding data packets from Internet users.
- To give you more experience manipulating IP addresses, to make sure you understand CIDR notation and the purpose of network masks.
- To introduce the select() and poll() API functions for managing multiple sockets in a single program.

Your BGP router will be run inside a simulator provided by us. Inside the simulator, your router will need to accomplish all of the essential functions of a real router, i.e. accepting route announcements from simulated peer routers, generating new route announcements for your peer routers, managing and compressing a forwarding table, and forwarding data packets that arrive from simulated Internet users.

Your router will be tested for both correctness and performance. Correctness means generating the correct routing announcements, and forwarding data packets to the correct destination. Performance means generating a succinct (i.e. compressed) forwarding table by aggregating forwarding entries when possible. Along with the simulator, you will be provided 16 test cases that assess the functionality of your router.

Small groups are allowed when working on this project. *We strongly recommend working with a partner.* **We will not accept submissions from groups of 3 or larger.**

Language

You can write your code in whatever language you choose, as long as your code compiles and runs on **unmodified** Khoury College Linux machines or college vagrant machine **on the command line**. Students in previous terms using Java have reported that there is no readily available support either in the standard library or open sources projects for the particular socket variant [AF_UNIX SOCK_SEQPACKET] we use for this project. As such, *we will not support Java or any other language that does not have an implementation of AF_UNIX SOCK_SEQPACKET* for this project.

You may use an IDE during development, but do not turn in your IDE project without a Makefile, and make sure your code has **no dependencies** on your IDE.

BGP Router Overview

In reality, a BGP router is a hardware box that has a bunch of fancy, high-speed jacks (or *ports*) on it. First, a network administrator would plug cables into these ports that connect to neighboring BGP routers, either from the same Autonomous System (AS) or another AS. Second, the administrator would manually configure each of these ports by:

1. Choosing the IP address that the router will use on this port, since each port will have a different IP
2. Choosing whether this port leads to a provider, a peer, or a customer (i.e. what kind of BGP relationship does the router have with each neighbor?)

3. Possibly manually configuring some specific routes via each neighbor

Third, once this manual configuration is complete, the administrator would turn the router on, at which point it will contact its neighbors and establish BGP sessions. At this point, the neighboring routers can pass BGP protocol messages to each other, as well as pass data packets from Internet users. The routers job is to:

1. Keep its forwarding table up-to-date, based on the BGP protocol messages it gets from its neighbors
2. Help keep its neighbors' forwarding tables up to date, by sending BGP protocol messages to them
3. Make a best-effort attempt to forward data packets to their correct destination

Your Program

For this project, you will submit one program named **router** that implements a BGP router. You may use any language of your choice. We will provide very basic starter code in Python. You may not use any libraries that implement routing logic; you must implement all of the routing logic yourself. If you have any questions about a specific library, post on Piazza.

Conceptually, your program is going to act like a router. When your program is executed, it will open several Unix domain sockets, each of which corresponds to one "port" on your router. Thus, **your program will have multiple open sockets**. This contrasts with Project 1, where your client had only a single open socket. Your router will receive messages on these sockets. Each message will either be a BGP command from the given neighbor, or a data packet that your router must forward to the correct destination. The simulator will take care of setting up the domain sockets, running your program, and closing your program at the end of each simulation. **Only one copy of your router will be running at any given time**, i.e. this project does not require you to manage concurrency or multiple parallel versions of your program.

If you use C or any other compiled language, your executable should be named *router*. If you use an interpreted language, your script should be called *router* and be marked as executable. If you use a virtual machine-based language (like Java or C#), you must write a brief Bash shell script, named *router*, that conforms to the input syntax below and then launches your program using whatever incantations are necessary. See Project 1 or Lab 2 for more information.

Requirements

To simplify the project, instead of using real packet formats, we will be sending our data across the wire in JSON format (many languages have utilities to encode and decode JSON, and you are welcome to use these libraries). Your router must meet the following requirements:

- Accept route update messages from the BGP neighbors, and forward updates as appropriate
- Accept route revocation messages from the BGP neighbors, and forward revocations as appropriate
- Forward data packets towards their correct destination
- Return error messages in cases where a data packet cannot be delivered
- Coalesce forwarding table entries for networks that are adjacent and on the same port
- Serialize your forwarding table so that it can be checked for correctness
- Your program must be called *router*

We will test your router by running it in a variety of simulations, each of which will stress different aspects of the BGP protocol and router logic. Your router should handle all message types and never crash. We encourage you to write debugging messages to a file while implementing your router.

Simulator and Starter Code

Rather than testing your router on a real network, we will test it in a simulator. The simulator takes care of create neighboring routers and domain sockets to connect to them, run your router program with the appropriate command line arguments, sending various messages, asking your router to "dump" its forwarding table, and finally closing

your router. The simulator comes with a suite of configuration files in the directory *tests/* that define situations your router must be able to handle. These are the same configuration files that we will use when evaluating everyone's code. Note that you do not need to parse the configuration files; the simulator is responsible for this functionality. The simulator itself is implemented in an executable file named *sim*. **At no point during this project should you modify *sim*.** Furthermore, when we grade your code, we will use the original versions of *sim* and the configurations in *tests/*, not your (possibly modified) versions. The simulator accepts the following command line spec:

```
$ ./sim [-h] [--test-dir <dir>] [--router <router>] <all|[milestone] file [...]>
```

At its simplest, the simulator can be run by providing a list of config files to test `router` against. Running the simulator with "`all`"" will run all of the test files in `<test-dir>`. Calling the simulator with "`all`" cannot be use in conjunction with a list of other config files. Running the simulator with "`milestone`" will add the config files we will be using to evaluate milestone submissions. "`milestone`" *can* be used in conjunction with a list of other config files. The simulator will tell you whether your router passed the test cases encoded in the list of config files provided, printing out error messages when something unexpected occurs.

Very basic starter code for the assignment in Python is provided and can be downloaded from the course website at the link below. The starter code provides a bare-bones skeleton of a router along with some helpful constants. You may use this code as a basis for your project if you wish, but it is strongly recommended that you only do so if you are comfortable with Python.

You should download and extract at least the simulator from its archive and into your development environment. The easiest way of doing so is with the following Linux commands:

```
$ curl http://course.khoury.neu.edu/cs3700sp20/archive/<file-name>
$ tar xzf <file-name>
```

Packages

- The simulator: bgp-sim.tar.gz
- Starter code in python: bgp-python-starter.tar.gz

Command Line Specification

The command line syntax for your router is given below. The router program takes one argument, representing the AS number for your router, followed by several arguments representing the "ports" that connect to its neighboring routers. For each port, the respective command line argument informs your router (1) what IP address it should use on this port, and (2) the type of relationship your router has with this neighboring router:

```
./router <asn> <ip_address>-<peer|prov|cust> [...]
```

Your router will always be passed at least one neighbor. For example, your router might be invoked by the simulator with following command line:

```
./router 7 1.2.3.2-cust 192.168.0.2-peer 67.32.9.2-prov
```

This command line tells your router several things:

1. Your router is part of AS 7
2. Your router has three ports, connected to three neighboring routers. Thus, your router will need to open three domain sockets.
3. The IP address of the neighboring router on the first port is 1.2.3.2, and so the IP address your router should use on this port is 1.2.3.1. The IP address of the neighboring router on the second port is 192.168.0.2, and so

the IP address your router should use on this port is 192.168.0.1. The IP address of the neighboring router on the second port is 67.32.9.2, and so the IP address your router should use on this port is 67.32.9.1.

4. The first neighbor belongs to a BGP customer. The second neighbor is a BGP peer. The third neighbor is a BGP provider.

Essentially, you can think of the command line arguments as the manual configuration that the router administrator would perform before turning the router on. **For the sake of simplicity, all of the neighboring routers will have IP addresses of the form *.*.*.2 and all of the IP addresses used by your router will be of the form *.*.*.1.**

Connecting to Your Neighbors

You will be using UNIX domain sockets to connect to your neighboring (simulated) routers, with one domain socket per neighbor. You do not need to be intimately familiar with how domain sockets work, but essentially they are socket-like objects that you can read or write. However, rather than sending and receiving packets over the Internet, the packets are instead passed between programs on the local machine. In other words, this is how your program will send and receive data from our simulator, which is just another program running locally on the machine. You should constantly be reading from your domain sockets to make sure you receive all messages (they will be buffered if you don't read immediately).

Exactly how to connect to a UNIX domain socket depends on your programming language. For example, if you were using python to complete the project, your code for connecting would look like:

```
from socket import socket, SOCK_SEQPACKET, AF_UNIX
s = socket (AF_UNIX, SOCK_SEQPACKET)
s.connect('ip.add.re.ss')
```

Where "ip.add.re.ss" is one of the command line parameters passed to your program sans relationship information. In golang, your code would look like:

```
c, err := net.Dial("unixpacket", "ip.add.re.ss")
if err != nil { panic(err) }
```

and produce similar results.

You may notice files named "ip.add.re.ss" sitting in your project directory. These files represent the domain sockets. Feel free to *manually* delete these files after the simulator terminates.

Handling Multiple Sockets

We encourage you to write your code in an event-driven style using select() or poll() on all of the domain sockets that your router is connected to. This will keep your code single-threaded and will make debugging significantly easier. Alternatively, you can implement your router in a threaded model with one thread handling each socket, but expect it to be significantly more difficult to debug.

Messages Format

To simplify the development and debugging of this project, we use JSON (JavaScript Object Notation) to format all messages sent on the wire. Most common programming languages have built-in support for encode and decoding JSON messages, and you should use these when sending and receiving messages. *You do not have to create or parse JSON messages yourself*. All messages will have the same basic form:

```
{
```

```
  "src":  "<ip_addr>",
  "dst":  "<ip_addr>",
  "type": "<update|revoke|data|no route|dump|table>",
  "msg":  {...}
}
```

Just like packets on the Internet, every message in our simulations comes from a source, and has a destination. These will always be IP addresses in dotted quad notation. How your program interprets the source and destination for a given message depend on the messages "type". The six message types are described below. The interpretation of the "msg" also depends on the messages "type".

Route Update Messages

The most basic and essential message that your router will receive from neighbors are route announcements. These messages tell your router how to forward data packets to far-flung destination on the Internet. Whenever your router receives a route announcement, it should (1) save a copy of the announcement in case you need it later, (2) add an entry to your forwarding table, and (3) potentially send copies of the announcement to neighboring routers. Route announcement messages have the following form:

```
{
  "src":  "<ip_addr>",                      # Example: 172.65.0.2
  "dst":  "<ip_addr>",                      # Example: 172.65.0.1
  "type": "update",
  "msg":
  {
    "network":    "<ip_addr>",              # Example: 12.0.0.0
    "netmask":    "<ip_addr_netmaks>",      # Example: 255.0.0.0
    "localpref":  <integer>,                # Example: 100
    "selfOrigin": <true|false>,
    "ASPath":     \[<asn>, [...]\]          # Examples: [1] or [3, 4] or [1, 4, 3]
    "origin":     "<IGP|EGP|UNK>",
  }
}
```

Using the example above as a guide, this route announcement is telling your router that your neighbor (172.65.0.2) knows how to forward data packets to the 12.0.0.0/8 network. In the future, if your router were to receive a data packet whose destination IP was in this network (e.g. 12.4.66.13), then your router should forward the data packet to this neighbor.

The "network" and "netmask" fields describe the network that is routable. The "localpref" field is the "weight" assigned to this route, where higher weights are better. The "selfOrigin" field describes whether this route was added by the local administrator (true) or not (false), where "true" routes are preferred. The "ASPath" field is the list of Autonomous Systems that the packets along this route will traverse, where preference is given to routes with shorter ASPaths. The "origin" field describes whether this route originated from a router within the local AS (IGP), a remote AS (EGP), or an unknown origin (UNK), where the preference order is IGP > EGP > UNK. The last fields of the message are important for breaking ties, when multiple paths to a given destination network are available; see the Data Forwarding section below.

Your router should store all of the information contained in route announcement. Additionally, your router may need to send copies of the route announcement to its neighbors. Who you send updates to is a function of (1) your relationship with the source of the update, and (2) your relationship with each neighbor. Your route announcements must obey the following rules:

- Update received from a customer: send updates to all other neighbors
- Update received from a peer or a provider: only send updates to your customers

Route Revoke Messages

Sometimes, a neighboring router may need to revoke an announcement. This typically occurs when there is some problem with the route, i.e. it doesn't exist anymore or there has been a hardware failure, so data packets can no longer be delivered. In this case, the neighbor will send a revocation message to your router. Your router must: (1) save a copy of the revocation, in case you need it later, and (2) remove the dead entry from the forwarding table Route revocation messages have the following form:

```
{
  "src":  "<ip_addr>",        # Example: 172.65.0.2
  "dst":  "<ip_addr>",        # Example: 172.65.0.1
  "type": "revoke",
  "msg": \[
    {"network": "<ip_addr>", "netmask": "<ip_addr_netmask>"},
    {"network": "<ip_addr>", "netmask": "<ip_addr_netmask>"},
    ...
  \]
}
```

Using the example above as a guide, this route revocation is telling your router that your neighbor (172.65.0.2) can no longer forward data to the two given networks. Note that for revoke messages, the "msg" field contains a list, i.e. it may contain multiple networks that should be removed from the forwarding table.

As with update messages, your router may need to send copies of the route revocation to its neighbors. This follows the same set of rules as update messages, see above.

Data Messages

Once your router has received some updates, it will have a forwarding table that it can use to try and deliver data messages to their final destination. Data messages have the following format:

```
{
  "src":  "<ip_addr>",        # Example: 134.0.88.77
  "dst":  "<ip_addr>",        # Example: 12.4.66.13
  "type": "data",
  "msg":  "<opaque>"
}
```

Note that the source and destination of data messages are not your router, or your neighboring routers. Rather, this is data that some Internet user is trying to send to some other Internet user (for example, a person trying to request a webpage). As such, **your router does not care about the "msg" or its contents.** Your router only cares about the destination IP address (and possibly the source IP address...). Your router's job is to determine: (1) which route (if any) in the forwarding table is the best route to use for the given destination IP, and (2) whether the data packet is being forwarded legally.

Lets handle the various possibilities from least to most complex. The easiest scenario is that your router does not have a route to the given destination network. In this case, your router should return a "no route" message back to the source that sent you the data. This message has the format:

```
{
  "src":  "<ip_addr>",          # Example: 172.65.0.1
  "dst":  "<ip_addr>",          # Example: 134.0.88.77
  "type": "no route",
  "msg":  {}
```

```
}
```

The next easiest scenario is that your router knows exactly one possible route to the destination network. In this case, your router should forward the data packet on the appropriate port. Your router does not need to modify the data message in any way.

The next scenario is more challenging. It is possible that your forwarding table will include multiple destination networks that all match the destination of the data packet. For example, suppose your forwarding table contains two entries: 172.0.0.0/8 and 172.128.0.0/9. These two ranges overlap. Now suppose a data message arrives with destination IP 172.128.88.99: which of the two paths should you choose? The answer is **the longest prefix match**, which in this case would be 172.128.0.0/9, since it has a 9-bit netmask, versus 172.0.0.0/8 which only has an 8-bit netmask.

The final scenario also concerns multiple possible routes. It is possible that your forwarding table will include multiple destination networks that all match the destination of the data packet, and that these matches will themselves be identical networks. For example, it is possible for your forwarding table to contain multiple entries for a given network, such as 172.0.0.0/8. In this case, there are specific rules your router should follow to break the tie, and decide which path to use:

```
1. The path with the highest "localpref" wins. If the "localpref"s are equal...
2. The path with "selfOrigin" = true wins. If all selfOrigins are the equal...
3. The path with the shortest "ASPath" wins. If multiple routes have the shortest length...
4. The path with the best "origin" wins, were IGP > EGP > UNK.
   If multiple routes have the best origin...
5. The path from the neighbor router with the lowest IP address.
```

Using these rules (plus the longest prefix match rule above), all ties between paths can be resolved.

Assuming that your router was able to find a path for the given data message, the last step before sending it along is to make sure that the packet is being forwarded legally. The relationship between the source router that sent the data to you, and the destination router is crucial here. Specifically:

- If the source router or destination router is a customer, then your router should forward the data. You always forward data for your customers (its one reason they are paying you).
- If the source router is a peer or a provider, and the destination is a peer or a provider, then drop the data message. Your router does not forward data for free, and in these cases, you would not be making any money.

If your router drops a data message due to these restrictions, it should send a "no route" message back to the source.

Dump and Table Messages

The final message type that your router must support is the "dump" message. This is not based on real BGP protocol message. Instead this is a message that the simulator needs in order to test your router for correctness. When your router receives a "dump" message, it must respond with a "table" message that contains a copy of the current forwarding table in your router. Dump messages use the following format:

```
{
  "src":  "<ip_addr>",          # Example: 72.65.0.2
  "dst":  "<ip_addr>",          # Example: 72.65.0.1
  "type": "dump",
  "msg":  {}
}
```

Your router must respond to the given source with a "table" message in the following format:

```
{
  "src":  "<ip_addr>",          # Example: 72.65.0.1
  "dst":  "<ip_addr>",          # Example: 72.65.0.2
  "type": "table",
  "msg": \[
    {"network" : "<ip_addr>", "netmask" : "<ip_addr_netmask>", "peer" : "<ip_addr>"},
    {"network" : "<ip_addr>", "netmask" : "<ip_addr_netmask>", "peer" : "<ip_addr>"},
    ...
  \]
}
```

Each object in the "msg" is a forwarding table entry from your router. Note again that in this case, "msg" contains a list, not an object. The "network" and "netmask" are the network prefix and associated CIDR netmask; the "peer" is the IP address of the router that is the next-hop for this path (i.e. the source IP of the router that sent the corresponding "update" message).

Path Aggregation

An important function in real BGP routers is path aggregation: if there are two or more paths in the forwarding table that are (1) adjacent numerically, (2) forward to the same next-hop router, and (3) have the same attributes (e.g. localpref, origin, etc.) then the two paths can be aggregated into a single path. For example, the networks 192.168.0.0/24 and 192.168.1.0/24 are numerically adjacent. Assuming the next-hop router and attributes are the same, these can be combined into 192.168.0.0/23. Notice that the netmask has gotten one bit shorter.

Your router must implement aggregation. The simulator will check to make sure you have implemented it correctly by asking your router to "dump" its forwarding table. In practice, aggregation should be triggered after each "update" has been received, i.e to compress the table. Note that "revoke" messages may require your router to disaggregate its table! For example, in the above case, consider what would happen if the 192.168.1.0/24 path was revoked. The simplest way to handle this is to simply throw away the entire forwarding table and rebuild it using the saved "update" and "revoke" messages. However, there are certainly more performant ways of dealing with disaggregation.

Config File Format

The configuration files specify the routers that will neighbor your router in each simulation, as well as the messages that will be sent. The list of "networks" are the neighboring networks, along with their AS number, network prefix, netmask, and type (customer, provider, or peer). These end up being the neighboring routers. The list of "messages" corresponds to the "update", "revoke", "dat", and "dump" messages that will be sent during the simulation.

You will notice that the *tests/* directory contains 16 test configurations, numbered by difficulty. Each configuration corresponds to a single simulation, and there are six levels of difficulty in all. Below, we outline a suggested order in which we think you should implement the features of your router that correspond with the difficulty of the tests.

Implementing Your Router

Although implementing your router may seem like a daunting task, it is manageable if you carefully plan the order in which you implement features. We recommend implementing your router using the following steps:

1. Before you write any code, look at some of the test cases and understand what kind of network topologies they create. Translate the network prefixes and netmasks into binary, and then look at the destination of the data messages. Which path should be chosen for each data message?
2. Start by writing the code to open the Unix domain sockets, and listen to all of them using select() or poll(). At this point, just print out messages you receive, and experiment with de-serializing the JSON and access fields of the objects.

3. Implement basic support for "update" and "data" messages. At this point, you can assume all of the neighboring routers are customers, and that the network prefixes will all be disjoint, so there will not be cases where paths overlap. Thus, your forwarding table implementation can be simplified for the time being:
    i. Implement logic for adding entries to your forwarding table, as well as sending "update" messages to other neighboring routers as necessary.
    ii. Implement forwarding of "data" messages to your neighbors by looking up the appropriate route in your forwarding table. At this point, you can assume that all "data" messages will be valid and legal.
4. Implement support for "dump" messages, and implement the "table" response message. At this point, your router should be able to pass the level-1 test configurations.
5. Expand your forwarding table implementation to include the various attributes (localpref, etc) and implement the five rules for selecting a path when multiple options are available. When you have implemented this correctly, your router should be able to pass all level-2 test configurations.
6. Implement support for "revoke" messages. This means being able to remove paths from your forwarding table, and sending "revoke" messages to neighboring routers as necessary. Also, add support for "no route" messages in cases where your router has no path to the destination of a "data" message. At this point, your router should be passing the level-3 test cases.
7. Implement the various rules for enforcing peering and provider/customer relationships. The means restricting which neighbors receive "update" and "revoke" messages, as well as dropping "data" messages when the transit relationship is not profitable. When done, your router should pass the level-4 test cases.
8. This is a major step: modify your forwarding table and associated logic to implement longest prefix matching. This will require encoding IP addresses as numbers and using bitwise logic to determine (1) whether two addresses match and (2) the length of the match in bits. When implemented correctly, your router will be able to pass the level-5 test cases.
9. The final challenge: implement route aggregation and disaggregation and pass the level-6 test cases.

Submitting Your Milestone

This is a very challenging project. To make sure everyone starts early, we require submission of a **milestone**. To complete the milestone, you must turn in a router program that is able to pass the level-1 test cases. The milestone is worth 1% of your final grade. To submit the milestone, follow the turn-in instructions below, using the *project2-milestone* variants instead of *project2*.

Submitting Your Project

Before turning in your project, you and your partner(s) must register your group. To register yourself in a group, execute the following command:

```
$ /course/cs3700sp20/bin/register <project2-milestone|project2> <team name>
```

This command registers you for the milestone submission and the final submission. This command will either report back success or will give you an error message. If you have trouble registering, please contact the course staff. **You and your partner must run this script with the same team name**. This is how we know you are part of the same group.

When submitting your project, you must provide a directory containing:

- Your thoroughly documented code.
- A Makefile to optionally compile your code. If you don't need a Makefile, it may be blank, but it must exist.
- A plain-text (no Word or PDF) README.md file. In this file, you should briefly describe your high-level approach, any challenges you faced, and an overview of how you tested your code.

You submit your project by running the turn-in script on `login.khoury.neu.edu` as follows:

```
$ /course/cs3700sp20/bin/turnin <project2-milestone|project2> <project directory>
```

The first parameter determines if you are turning in the milestone or the final submission. <project directory> is the name of the directory with your submission. The script will print out every file that you are submitting, so make sure that it prints out all of the files you wish to submit! The turn-in script will not accept submissions that are missing a README or a Makefile. *Only one group member needs to submit your project*. Your group may submit as many times as you wish; only the last submission will be graded, and the time of the last submission will determine whether your assignment is late.

Double Checking Your Submission

To try and make sure that your submission is (1) complete and (2) will work with our grading scripts, we provide a simple script that checks the formatting of your submission. This script is available on `login.khoury.neu.edu` and can be executed using the following command:

```
/course/cs3700sp20/code/project2/project2_format_check.py [path to your project directory]
```

This script will attempt to make sure that the correct files (e.g. README.md and Makefile) are available in the given directory, that your Makefile will run without errors (or is empty), and that after running the Makefile a program named router exists in the directory. The script will also try to determine if your files use Windows-style line endings (\r\n) as opposed to Unix-style line endings (\n). If your files are Windows-encoded, you should convert them to Unix-encoding using the `dos2unix` utility before turning in.

Grading

This project is worth 12% of your final grade in total. 1% comes from the milestone and 11% comes from the rest of the project. Grading for this project will be based on the number of test configurations that your router successfully completes, with more weight will be given to more difficult configurations (e.g. level-5 and level-6). At a minimum, your code must pass the test suite without errors or crashes, and it must obey the requirements specified above. We reserve the right to make your final project grade dependent upon a code review. All student code will be scanned by plagiarism detection software to ensure that students are not copying code from the Internet or each other.

You can see your grades for this course at any time by using the gradesheet program that is available on `login.khoury.neu.edu`.

```
$ /course/cs3700sp20/bin/gradesheet
```