Project 3: BGP Router

Manage multiple sockets in a simulated network

`The final project is due at 11:59pm on Tuesday, February 20, 2024.`

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 subnet mask notation and the purpose of network masks.
- To get more experience with 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.

(Il)Legal Libraries

Part of the challenge of this assignment is that **all code for manipulating IP addresses must be written by the student, from scratch**. For example, if you write your router in Python3, you may not use the *ipaddress* module.

If students have any questions about the legality of any libraries please post them to Piazza. It is much safer to ask ahead of time, rather than turn in code that uses a questionable library and receive points off for the assignment after the fact.

Language

You can write your code in whatever language you choose, as long as your code compiles and runs on Gradescope. Do not use libraries that are disallowed for this project. Similarly, your code must compile and run on the command line. You may use IDEs (e.g., Eclipse) during development, but do not turn in your project without a Makefile. Make sure you code has **no dependencies** on your IDE. We provide starter code in Python; you are welcome to use this, but if you decide use another language, you will need to port the starter code to your language yourself.

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 pass data packets into the routers. The routers job is to (1) keep its forwarding tables 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, and (3) make a best-effort attempt to forward data packets to their correct destination.

Starter code

Very basic starter code for the assignment in python is available on the Khoury GitHub server. Provided is a simple implementation of a router that simply connects to its peers, sends a handshake message, and prints out every other message it receives. You may use this code as a basis for your project if you wish. To get started, you should create a copy of the starter code on your local machine (or on the Khoury Linux machines if you wish):

`git clone https://github.khoury.northeastern.edu/cbw/bgp-router-starter-code`

Simulator

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 `configs/` 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 (in addition to a few additional configurations we do not reveal). Note that you do not need to parse the configuration files; the simulator is responsible for this functionality.

The simulator is implemented in a script named `run` that is available as part of the starter code.

At no point during this project should you modify `run`. Furthermore, when we grade your code, we will use the original versions of `run` and the configurations in `configs/`, not your (possibly modified) versions.

The simulator accepts the following command line spec:

```
$ ./run [config-file]
```

The simulator will read in the config file, start up your router with the appropriate arguments, and then feed your router messages from your neighbors. The simulator will record the messages your router sends, and will compare that output to what is expected. The simulator will tell you whether your router passed the test case, printing out a summary of the error if your router did not do what was expected.

Your Program

For this project, you will submit one program named `4700router` that implements a BGP router. You may use any language of your choice, and we will give you very basic starter code in Python 3. **You may not use any libraries that implement routing logic or IP address/subnet mask manipulation; you must implement all of the routing logic and IP address/subnet mask manipulation 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 UDP sockets, each of which corresponds to one "port" on your router. Thus, **your program will have multiple open sockets**. 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 UDP 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 `4700router` If you use an interpreted language, your script should be called `4700router` and be marked as executable. It should have an

```
#!/usr/bin/env -S python3 -u
```

If you use a virtual machine-based language (like Java or C#), you must write a brief Bash shell script, named `4700router`, that conforms to the input syntax below and then launches your program using whatever incantations are necessary. For example, if you write your solution in Java, your Bash script might resemble

```
#!/usr/bin/perl -w $args = join(' ', @ARGV); print 'java -jar 4700router.jar $args';
```

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 routing table cache so that it can be checked for correctness
- Your program must be called `4700router`

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.

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 UDP port you should use to communicate with the neighboring router, (2) what IP address of the neighboring router on this port, and (3) the type of relationship your router has with this neighboring router:

```
./4700router <asn> <port-ip.add.re.ss-[peer,prov,cust]> [port-ip.add.re.ss-[peer,prov,cust]] ...[port-ip
```

Your router will always be passed at least one neighbor.

For example, your router might be invoked by the simulator with following command line:

```
./4700router 7 7833-1.2.3.2-cust 2374-192.168.0.2-peer 1293-67.32.9.2-prov
```

This command line tells your router several things:

- Your router is part of AS 7.
- Your router has three ports, connected to three neighboring routers. Thus, for each neighbor (on ports 7833, 2374, and 1293, respectively), your router will need to:
  - create a UDP (SOCK_DGRAM) socket,
  - `bind()` the socket to `localhost:0`, where `localhost` is an alias for `127.0.0.1` and the `0` tells the OS to select an available source port

- send a *handshake* message to the neighbor on its port.

the neighboring router on the first port is 1.2.3.2, and 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 the IP address your router should use on this port is 192.168.0.1. Etc.
- 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 router router will be of the form* ...1.**

Connecting to Your Neighbors

You will be using UDP sockets to connect to your neighboring (simulated) routers, with one socket per neighbor. You do not need to be intimately familiar with how UDP sockets work, but essentially they are 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. For example, to send to a message to a specific neighbor, send the message in packet using a UDP socket to the UDP port the neighbor is listening on. You should constantly be reading from your sockets to make sure you receive all messages (they will be buffered if you don't read immediately). .

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 (see the starter code for an example). This will keep your code single-threaded and will make debugging your code 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.

Message Format

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

```
{
  "src":  "<source IP>",
  "dst":  "<destination IP>",
  "type": "<update|withdraw|data|no route|dump|table|handshake>",
  "msg":  {...}
}
```

Just like packets on the internet, every message in our simulations comes from a source `src`, and has a destination `dst`. 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 seven message types are described below. The interpretation of the `msg` also depends on the messages `type`.

Handshake Messages

When your router starts up, it needs to send a "handshake" message to each of your neighboring routers to let the neighbor know it is up. To do so, you can send:

```
{
  "src":  "<source IP>",
  "dst":  "<destination IP>",
  "type": "handshake",
```

For example, in the configuration above, your router would send the following handshake message to localhost port 7833 to connect to the first neighboring router:

```
{
  "src":  "1.2.3.1",
  "dst":  "1.2.3.2",
  "type": "handshake",
  "msg":  {}
}
```

And it would send similar messages to the other neighboring routers.

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":  "<source IP>",        # Example: 172.65.0.2
  "dst":  "<destination IP>",   # Example: 172.65.0.1  Notice the final one, this will typically be the
  "type": "update",
  "msg":
  {
    "network":    "<network prefix>",           # Example: 12.0.0.0
    "netmask":    "<associated subnet netmask>",# Example: 255.0.0.0
    "localpref":  "<integer>",                  # Example: 100
    "selfOrigin": "<true|false>",
    "ASPath":     "{<nid>, [nid], ...}",        # 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” is the “weight” assigned to this route, where higher weights are better. “selfOrigin” describes whether this route was added by the local administrator (true) or not (false), where “true” routes are preferred. “ASPath” is the list of Autonomous Systems that the packets along this route will traverse, where preference is given to routes with shorter ASPaths. “origin” 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

When you send a copy of your route announcement to a neighbor, you should *not* include any of the "private" fields (no the f, s gin, or origin). Your announcement sho only inclu e network, netmask, and ASPath.

Route Withdraw Messages

Sometimes, a neighboring router may need to withdraw 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, (2) remove the dead entry from the forwarding table, and (3) possibly send copies of the revocation to other neighboring routers. Route revocation messages have the following form:

```
{
  "src":  "<source IP>",        # Example: 172.65.0.2
  "dst":  "<destination IP>",   # Example: 172.65.0.1
  "type": "withdraw",
  "msg":
  [
    {"network": "<network prefix>", "netmask": "<associated subnet mask>"},
    {"network": "<network prefix>", "netmask": "<associated subnet mask>"},
    ...
  ]
}
```

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 withdraw 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":  "<source IP>",        # Example: 134.0.88.77
  "dst":  "<destination IP>",   # Example: 12.4.66.13
  "type": "data",
  "msg":  "<some data>"
}
```

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":  "<source IP>",        # Example: 172.65.0.1, i.e., the router's IP on the given port
  "dst":  "<destination IP>",   # Example: 134.0.88.77
```

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 entries 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 entry to use:

1. The entry with the highest `localpref` wins. If the `localpref` s are equal...
2. The entry with `selfOrigin` as true wins. If all `selfOrigin` s are the equal...
3. The entry with the shortest `ASPath` wins. If multiple entries have the shortest length...
4. The entry with the best `origin` wins, where IGP > EGP > UNK. If multiple entries have the best origin...
5. The entry from the neighbor router (i.e., the `src` of the update message) with the lowest IP address.

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

Assuming that your router was able to find a entry 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 routing announcement cache in your router. Dump messages use the following format:

```
{
  "src":  "<source IP>",        # Example: 72.65.0.2, i.e., a neighboring router
  "dst":  "<destination IP>",   # Example: 72.65.0.1, i.e., your router
  "type": "dump",
  "msg":  {}
}
```

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

```
{
  "src":  "<source IP>",        # Example: 72.65.0.1, i.e., your router
  "dst":  "<destination IP>",   # Example: 72.65.0.2, i.e., the neighboring router
```

```
    "type": "table",
    [
      {"network" : "<network.in.quad.notation>", "netmask" : "<netmask.in.quad.notation>", "peer" : "<peer
      {"network" : "<network.in.quad.notation>", "netmask" : "<netmask.in.quad.notation>", "peer" : "<peer
      ...
    ]
  }
```

Each object in the `msg` is a cached route announcement 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 subnet mask; 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); the other fields are the other properties of the route announcement. **Also note that you will need to perform aggregation (see next section) on these announcements before you send your response.**

Aggregation

An important function in real BGP routers is aggregation: if there are two or more entries 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 entries can be aggregated into a single entry. 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 when possible. 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 however that "withdraw" 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 announcement was withdrawn. The simplest way to handle this is to simply throw away the entire forwarding table and rebuild it using the saved "update" and "withdraw" messages, although there are certainly more performant ways of dealing with disaggregation.

Testing Your Router

In order for you to test your code, we have provided a simulator that will create neighboring routers and domain sockets to connect to them, run your router program with the appropriate command line arguments, send various types of messages, ask your router to "dump" its forwarding table, and finally close your router. You should copy the simulator and the associated *tests/* directory into your local directory. The simulator can be executed using either of the following commands:

```
user@host$ ./run configs/<config-file>
```

This will run a single test of your choice from the `configs/` directory. Note that you do not need to parse the configuration files in the `configs/` directory; the simulator will read and parse these files. The simulator will tell you whether your router passed the test, as well as print out error messages if specific unexpected things happen, like messages being delivered to the wrong destination.

Config File Format

You will notice that the `configs/` 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.

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" … "dump" messages that will be sent during the simulation. Eac… these includes a summary of what the expected messages that your neighboring routers should receive are.

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:

- 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?
- 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 deserializing the JSON and access fields of the objects.
- 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 can be simplified at this point. Implement the logic for adding entries to your forwarding table, as well as sending "update" messages to other neighboring routers as necessary. 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.
- Implement support for "dump" messages, and implement the "table" response message. At this point, your router should be able to passed the level-1 test configurations.
- Expand your forwarding table to include the various attributes (e.g., localpref), 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.
- Implement support for "withdraw" messages. This means being able to remove paths from your forwarding table, and sending "withdraw" 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.
- Implement the various rules for enforcing peering and provider/customer relationships. The means restricting which neighbors receive "update" and "withdraw" 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.
- 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.
- The final challenge: implement route aggregation and disaggregation and pass the level-6 test cases.

Submitting your project

To turn-in your project, you should submit the following three things:

1. Your thoroughly documented source code that implements your router program.
2. A `Makefile` that compiles your code.
3. 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.

Your `README.md`, `Makefile`, source code, etc. should all be placed in the root of a compressed archive (e.g., a .zip) and then uploaded to Gradescope. Do not use the compress or archive option on a Mac – Gradescope does not parse these archives. Instead, use the `zip` program on the command line. Alternatively, you can check **all** these items into Github, download a zip file from Github, and submit that to Gradescope.

Grading

The grading in this project will be broken down as follows:

Item	Percentage of Grade
Program correctness	85%
Style and documentation	15%

The final grading in this project will be based on the number of test configurations that your router successfully completes. 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. All student code will be scanned by plagiarism detection software to ensure that students are not copying code from the internet or each other.

← Project 2: FTP Client

Project 4: Reliable Transport Protocol →