BGP Simulator Documentation

Group 8
Niko Hellgren, 505174

Protocol Processing, Spring 2017

Contents

1 Introduction

The simulator software is implemented in Java programming language using just the base libraries. For visualization purposes, a graph plotting library, GraphStream [1], was utilized. A high-level overview of the program structure is presented in Section2. The code is available in `https://github.com/nipehe/BGP-simulator` for the time being.

The simulation has the following basic technical features and limitations:

- The simulation is strongly threaded, with each router and client working as its own thread and `Timer` threads utilized in testing and `KEEPALIVE` messages.

- Layer 1 and Layer 2 functionalities are simulated by Java's `PipedInputStream` and `PipedOutputStream` that are made to transit `byte[]` arrays between threads. The output streams are `synchronized` to make sure the sent packets do not mix up.

- Layer 3 functionality is implemented according to the IPv4 protocol. Although the constructed packets contain and transmit all the fields defined in [2], fields *Type of Service*, *Identification*, *Flags*, *Fragment Offset*, and *Protocol* are filled with default values in all cases, and they are not used for anything. This is partly enabled by the simulation not supporting packet fragmentation.

- A globally available static class provides the simulated routers and clients with DNS-like functionality, since implementing an actual DNS functionality is not essential in this simulation.

- To provide packets for the simulation, clients are attached to the router objects and they have functionality to send data to each other over the network.

- The simulated network uses BGP-4 messages to transmit information between routers.

- Since the simulation is focused on BGP, the idea of Autonomous Systems (AS) is abstracted into each AS being represented by one router, and the clients attached to this AS being connected straight to the router. This removes some features concerning, for example, the `NEXT_HOP` attribute, since the hops done are always to another AS/router.

- TCP functionality is not implemented, and BGP packets are transmitted as ordinary IP packet payloads. This removes the possibility to test or simulate security and stability issues caused by TCP, and simplifies the connection process, but streamlines the implementation due to the complicated specification and functionality of TCP [3].

- Routing information is transmitted using `UPDATE` messages after connection initialization and all routing table changes.

- Additional BGP message type, `TRUST` has been added for trust voting between routers. To avoid Man-in-the-Middle attacks (usually the router being voted on is on the transmission path of the vote), the trust votes are encrypted using 1024-bit RSA and signed using RSA with SHA-1. To avoid making key transmission overly complicated, a PKI-like

```
bgp
├── client .................... Clients of simulator's routers
│   └── messages .............. Messages transmitted between clients
├── core ...................... Router implementation
│   ├── messages .............. BGP message implementations
│   │   ├── notificationexceptions . Exception classes used to notify about situations that
│   │   │                            should raise a NOTIFICATION
│   │   └── pathattributes .... Path attributes used in UPDATE messages
│   ├── network ............... Network adapter simulation, L1/L2 functionality
│   │   ├── fsm ............... Finite state machine used by the inter-router connections
│   │   └── packet ............ Interfaces defining behaviour for actors handling packets
│   ├── routing ............... Routing table engine
│   └── trust ................. Inter-router trust table
├── tests ..................... JUnit tests to ensure correct working of the program
├── ui ........................ User interface components
└── utils ..................... Functionalities needed by various classes (e.g. IP addresses)
```

Figure 1: Package structure of the code

functionality was made available as a globally available class, providing routers with other routers public keys. The trust implementation is discussed in more detail in Section 5.

- Behaviour in error situations has been implemented comprehensively, and both L1/L2 breakage, missing `KEEPALIVE` messages, and erroneous BGP messages cause the routers to drop the link and inform their neighbourhood of this. Possible issues effect the trust rate of the misbehaving router.

- Since the simulation is targeted at demonstrating the behaviour of BGP, logging has not been implemented.

2 Code overview

This section discusses the code-level decisions and structure in the project. Smaller details are overlooked, and the focus is on main functional parts of the program. A package-level visualization of the project can be found in Figure 1.

`bgp.core.SimulatorState` Global state to transmit information between parts of the simulation that is either not possible or relevant to transmit via the network. This contains registry of the routers and clients (with checks for duplicate ID's), DNS-like functionality that provides information about client's addresses for data transmission purposes, and PKI-like functionality that provides routers' public keys used in trust voting.

`bgp.core.BGPRouter` Objects from this class represent the routers that make up the network. Each router is given an ID (AS ID) and a specified subnet. Routers have a list of `ASConnections` that represent the connections to the router's neighbours. Each router also has their own `RoutingEngine` and `TrustEngine`, responsible for routing table upkeep and neighbour trust calculations, respectively. The routers implement a DHCP-like behaviour, where they provide clients with IP addresses. To avoid maintenance packets blocking the routing, there are two threads in each router: one for making packet routing decisions and one for processing BGP packets designated to current router. Despite this separation, BGP messages are transmitted via the same media as ordinary routed packets. `BGPRouter` also has functionalities that bind the features of routing and trust engine and various `ASConnection`s together.

`bgp.core.ASConnection` Class responsible for storing information concerning the state of a connection from one router to another. Contains the BGP finite state machine, information about this connection's IP address, and a reference to its `InterRouterInterface`. Is responsible for sending `OPEN` and `KEEPALIVE` messages and handling events that require sending a `NOTIFICATION`.

`bgp.core.network.InterRouterInterface` Represents the L1/L2 functionalities responsible for transferring data between routers. Contains a `PipedInputStream` and `PipedOutputStream` for transferring `byte[]` arrays with the connected neighbour, and a thread responsible for listening the input stream. Sending a packet consists of sending 10 bytes (0x00, 0x00, ..., 0x00, 0xFF) for synchronization and marking the beginning of transmission (not necessary in simulation environment), 2 bytes representing the length of the upcoming packet (MTU is therefore 64 kilobytes), and finally the IP packet as bytes.

`bgp.core.routing.RoutingEngine` Handles `UPDATE` messages, decides the routing paths and keeps up the routing table. The routing engine behaviour is discussed in more detail in Section 4.

`bgp.core.trust.TrustEngine` Calculates trust values for neighbouring nodes based on received `TRUST` messages and user-defined direct trust, creates `TRUST` requests and votes with cryptographic protections in place. Trust functionality is presented in Section 5.

`bgp.core.messages.BGPMessage` This class, together with its subclasses `OpenMessage`, `KeepaliveMessage`, etc., is used to construct BGP messages and serialize them to bit-level representation, and deserialize them to be used by the receiving end. De-serialization phase is also responsible for message validation.

`bgp.client.BGPClient` Class representing a client of the network, connected through one of the routers. Clients have their own IP addresses and are capable of receiving IP packets. To test network functionality, a ping request and response have been implemented.

`bgp.utils.Address` `Address`, together with its subclass, `Subnet`, represent IP addresses and and subnets, respectively, and can be converted into both 4-`byte` array representation and a 64-bit `long` representation (`long` was used instead of 32-bit `int` to avoid the 1st bit, sign, affecting possible comparisons of addresses). Subnets also store the length of the prefix in question as a bitmask.

`bgp.utils.PacketEngine` Functionalities regarding the construction, validation and modification of IP packets. Packet constructions and verifications, checksum calculations, TTL modifications and field extractions are done here. As mentioned in Section 1, Version, IHL, DSCP, ECN, protocol and fragmentation fields are used, but filled with default values.

3 Connection initialization process

4 Routing engine

This section presents the routing engine used to make routing decisions and handle `UPDATE` messages. The decision process on receiving an `UPDATE` message is discussed first, with behaviour on link breakage and route selection presented after that.

4.1 Decision process

Upon receiving a new `UPDATE` message, the routing engine validates it and then follows the following algorithm:

1. Iterate over the withdrawn routes (if present):
    - If exactly matching routing information is not found, do nothing.
    - Else, if the first hop on the `UPDATE` message matches the current best path for the subnet, remove the routing information for the subnet and store the subnet to be sent to other neighbours (if currently revoking neighbour was not on the best path, it is not necessary to inform neighbours of this, since it is not on the best path from this router).
    - Else store the information about the subnet and selected best path to it. This collection of non-removed subnets will be replied to the revoking peer to inform it of a possible alternative route through this router.
2. Iterate over the Network-layer reachability information (if present):
    - If no exact match to the subnet in question is found from the current routing table, create a new node and insert it into the correct location on the subnet tree, with first hop, path length and local preference information based on the path attributes.
    - Else if the local preference values of the current and possible new path differ; store/keep the better selection in the tree.
    - Else, store the shorter path scaled with trust values of the neighbours (presented in detail in Subsection 5.1).
    - If the selected path was changed, store it for sending to other neighbours.

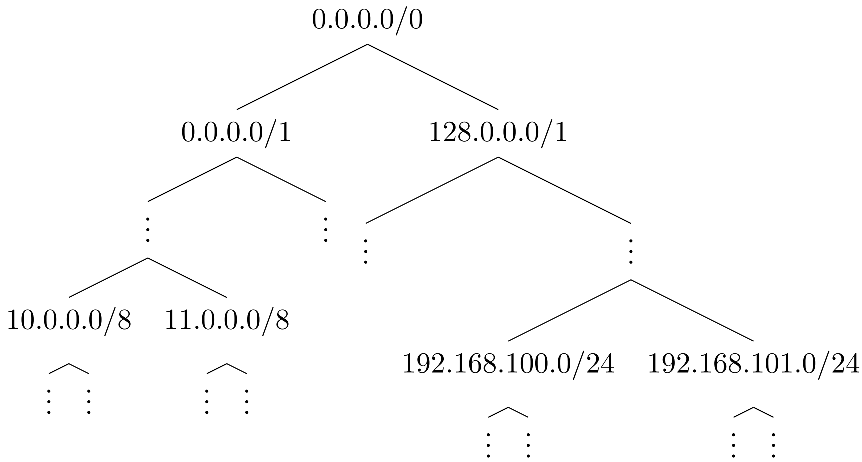

Figure 2: Tree structure used to categorize the subnets

3. Modify the received `UPDATE` message:
   - Add own ID to the beginning of `AS_PATH`.[1]
   - Remove routes that were not used from Withdrawn routes list.
   - Remove NLRI entries that were not utilized by this router.
4. Send the modified message to each neighbour not in the `AS_PATH`.[2]
5. Send information about revoked subnets reachable via this router to the revoking neighbour.

4.2 Route withdrawals

If a connection to a neighbour is broken, an `UPDATE` message with all subnets that were originally reached via that neighbour attached to the Withdrawn routes list. This message is then handled according to the algorithm defined in 4.1, thus removing the routing information and sending the information to all neighbours. Due to the reply feature defined in the decision process algorithm, if an alternative route exists, information about that should reach the router originally revoking connections, after the information has reached a router that has a differing best path.

4.3 Subnet tree

Subnets in the 32-bit IPv4 address space can be categorized into a binary tree, with each level having prefix length one longer than the previous one (see Figure 2). For the purposes of the routing engine, the binary tree was not complete, but was instead populated with all the prefixes received as Network-layer reachability information (NLRI), placing each new node in the graph as a child of the longest prefix containing the new subnet. Each node contains information about the first hop to be taken to get on the currently selected path to the target subnet, local

[1] `NEXT_HOP` value is changed before serializing the message for each neighbour.

[2] Since changes not affecting routing decisions are not propagated, the original `UPDATE` message content diminishes as the information travels further in the network.

preference of the currently selected first hop and the length of the path to the goal node. The graph is started with a single node with the prefix 0.0.0.0/0, with first hop set to be the default route.

4.4 Route selection

Route selection is achieved by first finding the longest matching subnet by recursively looking through the subnet tree, continuing deeper into the tree as long as a child subnet node that contains the address being looked for is found, and then extracting the first hop ID from the node representing the longest matching prefix. `BGPRouter` has a map from neighbour's ID to the corresponding `ASConnection`, which can then be used to transmit the packet. If the longest matching prefix is the current router's subnet, local routing is done to either one of the clients or the router itself.

5 Trust implementation

5.1 Calculating trust

6 Miscellaneous notes from the project

- When Java interprets bytes as integers, they are shifted to range -128..127. Doing bitwise shifts also easily converts the values to a larger data size (e.g. 32-bit integers) instead of dropping the overflowing bits. This caused multiple errors in initial implementations of the bit-level manipulations, and adding bitmasking (e.g. `<value>&0xFF` to force a value to 8 bits) was necessary in most of the places to avoid issues caused by this. Lack of unsigned numbers also made value comparisons difficult at some points, since the interpretations of bytes easily flowed over to the negative numbers.

- Great built-in support for threads, timers and task executors in Java 8 made some otherwise difficult tasks (e.g. `KEEPALIVE` message sending and checking) really easy.

- Following object-oriented paradigm in development was both intuitive and helpful, due to the software being a simulator.

- Since the definition of BGP [4] only specifies the limitations and the requirements of the protocol, information concerning the practices that the BGP speaker should follow was hard to come by.

Bibliography

[1] GraphStream Team, "Graphstream - a dynamic graph library," Mar. 2017.

[2] J. Postel, "Internet Protocol," STD 5, RFC Editor, September 1981. `http://www.rfc-editor.org/rfc/rfc791.txt`.

[3] J. Postel, "Transmission Control Protocol," STD 7, RFC Editor, September 1981. `http://www.rfc-editor.org/rfc/rfc793.txt`.

[4] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)," RFC 4271, RFC Editor, January 2006. `http://www.rfc-editor.org/rfc/rfc4271.txt`.