Project 2: Routing Process Simulator

Introduction to Data Communication Networks (2023 Fall)

Instructor: Prof. Kyunghan Lee

TAs: Jongseok Park, Dongsu Kwak

Deadline: December 7th, 2023

Project Overview

Nowadays, packet switching is a popular method to transmit data due to efficiency of network resources. For packet switching, each router or switch should decide next hop of each packet using routing table. In this project, we will simulate routing process in network.

In this document, we will first give you a general introduction about routing. We will then give details on what behavior you should expect from your implementation, and what you will have to implement to complete the simulator. Make sure to read them all carefully.

Details on grading and submission will be presented in the “Grading” and “Submission” sections separately.

Routing Protocol

A. Introduction to Routing Protocol

Switches or routers decide where the packet goes to. If we know the whole graph of the network, routers will easily decide the route. However, router cannot know this information at the beginning. We use the routing tables to help the decision process. Routing tables notice the IP address of gateway depending on IP prefix of the destination. Then, how to make routing table is a good question for efficient packet delivery.

Human experts can write the routing table of every router. However, as the Internet grows, it is not affordable to update all the routers in the world. outing protocol that automatically updates routing tables is proposed. To achieve this goal, routers communicate with each other to change information that enables to choose paths. We can classify routing protocols by routing type, domain type or routing algorithm etc. Distance-vector, link-state, path-vector is type of routing algorithm. In this project, we will implement these three types of routing algorithms.

Distance-vector algorithm finds shortest paths using distance between routers themselves and destinations. Bellman-Ford algorithm helps that routers can find shortest paths using distance information. Link-state algorithm finds shortest paths using connectivity graph of routers. Each router broadcast information about its

neighboring routers to know the whole structure of the network graph. We use Dijkstra algorithm to find shortest paths in this network graph. Path-vector algorithm gets paths using list of routers that packets pass through to reach the destination.

B. Configuration of Simulator

We implemented simulator using C++. There are several classes in this simulator. Below is a simple explanation for each class. In Appendix A, we provide functions of each class.

Class *Network* represents the whole simulator. There are 3 inherited classes *Network_DV, Network_LS, Network_PV.* Each inherited class represents the simulator that uses each routing algorithm.

Class *Auto_sys* represents the autonomous system (AS). The autonomous system is a collection of connected IP routing prefixes under the control of network operator. We assume that each autonomous system in this simulator has 1 IP prefix, and that the IP address of the representative router in each autonomous system is at the beginning of the range of the IP prefix. There are 3 inherited classes *AS_Dist_Vect, AS_Link_Stat, AS_Path_Vect.* Each inherited class represents the autonomous system that uses each routing algorithm.

Class *Routing_Table* represents the routing table in router. There is a vector of pointer of class *Routing_Info*. *Routing_Info* represents the routing information in the routing table. There are inherited classes for *Routing_Table, Routing_Table_DV, Routing_Table_LS, Routing_Table_PV,* and inherited classes for *Routing_Info, Routing_Info_DV, Routing_Info_LS, Routing_Info_PV.*

Class *Forward_Table* represents the forwarding table in router. There is a vector of class *Forward_Info,* that represents the forwarding information. This forwarding table only contains the information of neighboring autonomous systems.

Class *Packet* represents the packet that routers communicate. Detail structure of packet for each routing algorithm will be explained.

Class *IP_prefix* represents the IP prefixes. IP prefixes can be expressed such as 147.46.0.0/15.

After compiling codes, you can run the binary file using below command.

./route_sim <filename> D/L/P

Capital letter, D, L or P means mode Distance-Vector, Link-State, Path-Vector. In <filename>, type the graph file that you want to simulate. The graph file should place in *graph_csv* folder and be in *.csv* format. Graph files are formatted as follows. First line will be *n,* the number of autonomous systems. Second line to (n+1)-th will be the IP address and netmask for IP prefix of each autonomous system. (n+2)-th line will be *m,* the number of links in the network. From (n+2)-th to (n+m+2)-th line, there are 3 integers in the line. The two previous integers represent the endpoints of link, last

integer represents metric (cost) of the link. You can make your own network following this file format.

If we type previous command in system window, simulator initially makes network following the csv file and makes routing table. After this, you can type command to use simulator. Detailed explanation of command is following.

- ✓ help: It provides list of commands.
- ✓ print [i]: This command prints routing table of autonomous system with ASN i. [i] should be integer. You should type valid index.
- ✓ change [i] [j] [metric]: This command change metric of specific link which endpoints are i and j. Note that this command does not add link. Link between AS i and j should be existed.
- ✓ exit: Finish the simulator.
- ✓ timestamp: print current timestamp

In this simulator, you only need to consider and implement routing table initialization and routing table update when the link metric changes. You only need to implement functions related with *update_rt_table()* and *send_packet_neighbor()* functions in class *Auto_sys* and its inherited classes. *update_rt_table()* updates routing table using received packet from neighboring AS. *update_rt_table()* returns whether the packet should be sent to the neighbor. *send_packet_neighbor()* sends packet to neighboring AS. Note that you should update using packets, not accessing neighbor's routing table directly.

Simulator initializes the routing table of all AS in the network using *init_rt_table().* Then *send_packet_neighbor()* sends packets. Simulator repeats *update_rt_table()* - *send_packet_neighbor()* to update routing tables. Simulator will stop this action if all AS do not need to send packets. The process of simulator when we change metric of link is similar to process of initialization of routing tables. We change the routing table of two endpoints using *change_rt_table().* Then *send_packet_neighbor()* sends packets. The simulator repeats *update_rt_table()* - *send_packet_neighbor()* to update the routing tables. Figure 1 is the logic flow of the simulator that you need to implement.

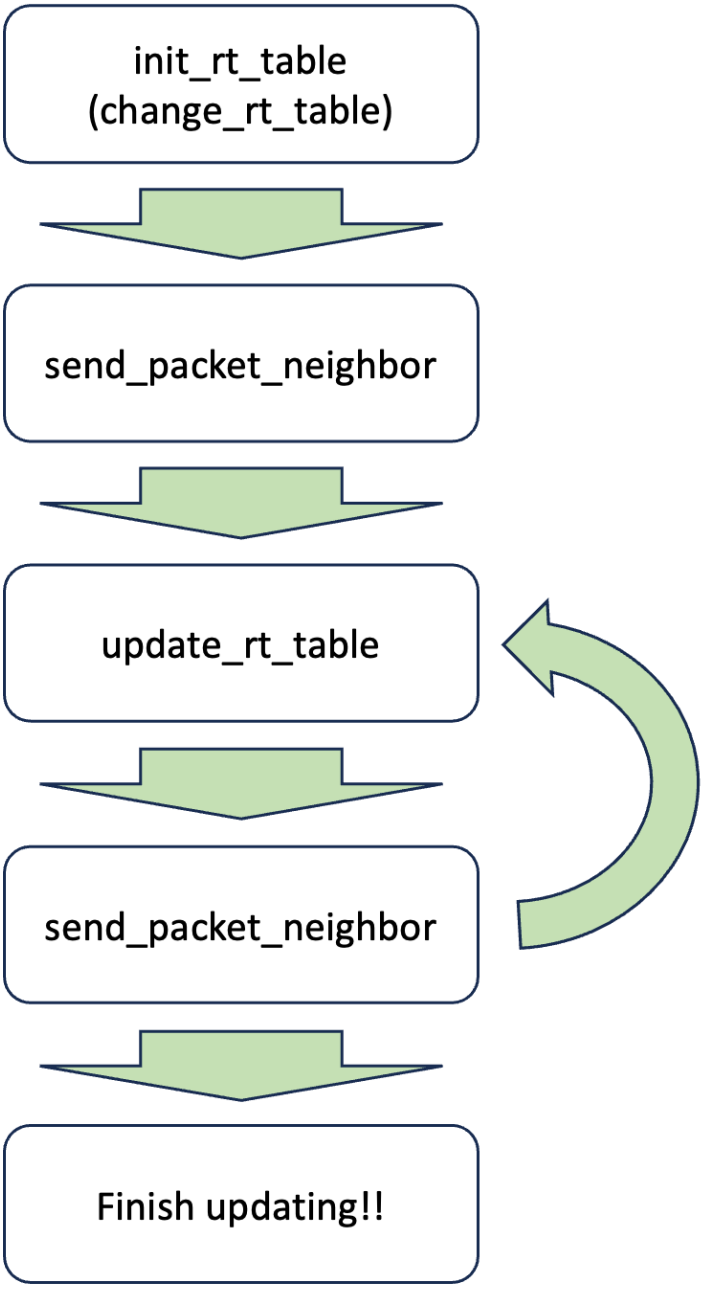

Figure 1 Logic flow of simulator

C. Detail in Routing Protocol

We first introduce the common ground in the protocols to be implemented. Three routing protocols use **synchronous update**, that is, each router updates its own routing table simultaneously. They also send their packets simultaneously to neighboring AS. Routing information must carry about IP prefix of destination, IP address of the next hop, total metric to destination. The format of the packet header used by the routing protocol does not depend on the protocol type. The format of the packet header is shown in Table 1.

Table 1 Packet header format

Bit offset	0-7	8-15	16-23	24-31
0	Protocol	Type	Length	
32	IP address of Source			
64	IP address of Destination			
96	Timestamp			
128 ~	Body			

- ✓ *Protocol* segment shows us the type of routing protocol. Distance-vector routing protocol, link-state routing protocol, path-vector routing protocol each uses the value *0x80, 0x81, 0x82*.
- ✓ *Type* segment shows us the type of packet in routing protocol. In this simulator, we just use update that value is *0xFF*.
- ✓ *Length* shows us the total length of packet including header and body. Unit of the length is byte. For example, when we send a packet which does not have body segment, length should be 16.
- ✓ *IP address of source and IP address of destination* is IPv4 address of each router. This IPv4 address can be expressed in 32 bits.
- ✓ *Timestamp* is 32-bit integer to support synchronized update.

This packet header is implemented in class *Packet.*

Routing information in routing table consists of the destination IP prefix, next hop IP address, total metric to destination in common. For each routing algorithm, additional data is included in the routing information. Additional data will be explained in detail later.

Distance-Vector Routing Protocol

In distance-vector routing protocol, we update next hop using the total metrics when packets pass through each neighboring AS.

Additional data for distance-vector routing algorithm is *metric_via_neighbor,* total metric to destination when we pass through each neighboring AS. The size of the additional data is equal to the number of neighboring AS. Index of additional data follows index of forwarding table. Initial value of *metric_via_neighbor* is INF.

Segment of packet body for distance-vector routing protocol is shown in Table 2.

Table 2 Packet body segment for distance-vector routing protocol

Bit offset	0-31
0	IPv4 address of target IP prefix
32	Netmask of target IP prefix
64	Total metric to target

Segment of packet body for one routing information consists of 12 bytes. We will send these segments for all data in routing table. If there is routing information about n IP prefixes in routing table, we will send $12n$ bytes for packet body.

Link-State Routing Protocol

In link-state routing protocol, we update next hop using the information of link. We first make map of network similar as adjacency matrix. Then we update the link using Dijkstra's algorithm.

Additional data for link-state routing algorithm is *ASN* (autonomous system number) and *metric_to_AS*. *metric_to_AS* shows us that this target AS is linked to ASN i with this metric. If link does not exist, value will be INF. Note that self-looping link does not exist. Index of metric to AS is equal to ASN.

Segment of packet body for link-state routing protocol is shown in Table 3.

Table 3 Packet body segment for link-state routing protocol

Bit offset	0-31
0	IPv4 address of target IP prefix
32	Netmask of target IP prefix
64	ASN of target
96	metric_to_AS with ASN 0
…	…
32 m+ 64	metric_to_AS with ASN m-1

Packet body segment for one routing information consists of $4m + 12$ bytes where m is the number of autonomous systems. Routers will send these segments for all data in their routing table and **themselves**. If there are routing information about n IP prefixes in routing table, we will send $4(n + 1)(m + 3)$ bytes for packet body.

init_rt_table() is an empty function. *update_rt_table() consists of update_map()* and *update_table_dijkstra(). update_map()* updates the connection map using the received packets. *update_table_dijkstra()* updates the next hop and total metric using Dijkstra's algorithm.

Dijkstra's algorithm finds the shortest path. It picks unvisited node with lowest distance. Then calculate distance through it and update when this distance is smaller than existing one. Repeat this process until it finds the shortest paths for all nodes.

Path-Vector Routing Protocol

In path-vector routing protocol, we update routing table using path. Note that this path-vector routing protocol finds the path that the number of hops is smallest. Metric of link does not affect to result of path-vector routing protocol. So, we do not change metric of the link when we simulate path-vector routing protocol.

Additional data for link-state routing algorithm is *path_to_AS*. In the *path_to_AS*, there are IP addresses of AS that packets pass through. Note that there is no information of themselves in path, but IP address of destination is included in path.

Segment of packet body for path-vector routing protocol is shown in Table 4.

Table 4 Packet body segment for path-vector routing protocol

Bit offset	0-31
0	IPv4 address of target IP prefix
32	Netmask of target IP prefix
64	Length of path_to_AS
96	IPv4 address of first AS in path_to_AS
…	…

Segment of packet body for one routing information is consists of at least 12 bytes. Size of packet body segment is depending on the length of path. Routers will send these segments for all data in their routing table.

In *update_rt_table(),* we just update the paths by adding address of neighboring AS in front of the path.

Misc.

Document

Here are some documents that may help you in this project.

C++ virtual function: https://www.geeksforgeeks.org/virtual-function-cpp/

C++ *new* and *delete*: https://www.geeksforgeeks.org/new-and-delete-operators-in-cpp-for-dynamic-memory/

C++ encapsulation: https://www.geeksforgeeks.org/encapsulation-in-cpp/

C++ inheritance: https://www.geeksforgeeks.org/inheritance-in-c/

C++ STL: https://www.geeksforgeeks.org/the-c-standard-template-library-stl/

C++ enumeration: https://www.geeksforgeeks.org/enumeration-in-cpp/

C++ STL documentation: https://cplusplus.com/reference/stl/

Dijkstra's algorithm: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Update log

We could update and re-upload this document. Note that we will be updating this document to improve or add explanations based on the FAQs, but not to change the essential content such as Grading, Submission, Functions to be implemented. Unless there is a fatal error, the provided code will not be changed.

Caution

Following lists could cause errors or undefined behaviors.

- Opening graph file that does not follow the format.
- Wrong IP prefix in the graph file such as 199.199.199.0/16 or 256.999.0.0/16
- Duplicated range of IP prefix such as 147.46.0.0/15 and 147.47.0.0/16.
- Graph of the network is not a connected graph.
- More than 100 nodes in network
- Multiple link establishment between same pair of AS in graph file.
- Changing global variable *timestamp* freely.
- Fixing *.hpp* files.
- Overflow of values

There are other errors that we haven't found. If we find some new errors for the simulator, we will notice that using ETL board about it.

Grading

We will deduct **20%** per each day for late submission.

Routing Protocol Simulator

The routing protocol simulator will be graded using test cases. These are the total points allocated for each function. A total of 120 points are allocated for the routing protocol simulator.

(20 points) AS_Dist_Vect::update_rt_table()

(10 points) AS_Dist_Vect::send_packet_neighbor()

(20 points) AS_Link_Stat::update_map()

(20 points) AS_Link_Stat::update_table_dijkstra()

(10 points) AS_Link_Stat::send_packet_neighbor()

(20 points) AS_Path_Vect::update_rt_table()

(20 points) AS_Path_Vect::send_packet_neighbor()

Report

Please answer questions in *report.docx*. Answer should be in **English**. We will not give you any points if you write answer in Korean. We want to see **your own thought** using this report. We will be grading the report based on length of answer, and logical flow. A total of 80 points are allocated for the report.

Submission

Take your “auto_sys.cpp” and “report.docx” and compress them into a single zip file, with the name “<YOUR_NAME>_<YOUR_STUDENT_ID>_Proj2.zip”. **DO NOT include any other files than “auto_sys.cpp” and “report.docx” in the .zip file.** Upload the .zip file to the ETL.

We hope you have fun!

2023. 11. 8.

Dongsu Kwak.

Appendix A. Functions in Simulator

Functions for utility

These functions are in *util.hpp* and *util.cpp*.

- ✓ printIP(uint32_t IPv4): print IP address using *IPv4*.
- ✓ printIP_no_new_line(uint32_t IPv4): print IP address using *IPv4*, there is no newline charact at the end.
- ✓ read_file(void** output, char *file_path): function that opens file.
- ✓ make_netmask(int power): it returns format of IPv4 address. range of *power* should be 1~32.
- ✓ in_ip_prefix_range(uint32_t IPv4, IP_prefix ip_range): Return true if that *IPv4* is in range of *ip_range*.

Functions in class *IP_prefix*

These variables and functions are in *util.hpp* and *util.cpp*.

- ✓ uint32_t IPv4: IPv4 address for IP prefix. set_IPv4() to change this value, get_IPv4() to get the value.
- ✓ uint32_t netmask: Netmask for IP prefix. set_netmask() to change this value, get_netmask() to get the value.

Functions in class *Forward_Info*

These variables and functions are in *forward.hpp* and *forward.cpp*.

- ✓ uint32_t IP_gateway: IPv4 address for next hop. set_IP_gateway() to change this value, get_IP_gateway() to get the value.
- ✓ IP_prefix IP_AS: IP prefix that next hop covers. set_IP_AS() to change the value, get_IP_AS() to get the value.
- ✓ uint32_t metric: Metric of the link. set_metric() to change the value, get_metric() to get the value.

Functions in class *Forward_Table*

These variables and functions are in *forward.hpp* and *forward.cpp*.

- ✓ std::vector<Forward_Info> fw_table: we stores forwarding table using STL vector. You can access to *fw_table* using following functions.
- ✓ void add_fw_info(Forward_Info fw_info): Push *fw_info* back. Size of *fw_table* increased by 1.
- ✓ void change_fw_info_metric(int idx, uint32_t metric): Change metric of *idx*-th index in *fw_table* into *metric.*
- ✓ int get_size(): Get the size of *fw_table*.
- ✓ Forward_Info get_fw_info(int idx): get the *idx*-th index of *fw_table*.

Functions in class *Packet*

These variables and functions are in *forward.hpp* and *forward.cpp*.

- ✓ uint8_t protocol: protocol segment in packet header, set_protocol() to change the value, and get_protocol() to get the value. You can easily set the value using enum Protocol.
- ✓ uint8_t type: type segment in packet header, set_type() to change the value, and get_type() to get the value. You can easily set the value using enum Packet_type.
- ✓ uint32_t ip_source: IP address of source segment in packet header. set_source() to change the value, and get_source() to get the value.
- ✓ uint32_t ip_dest: IP address of destination segment in packet header. set_dest() to change the value, and get_dest() to get the value.
- ✓ uint32_t timestamp: Timestamp segment in packet header. set_timestamp() to change the value, and get_timestamp() to get the value.
- ✓ uint16_t length: Length segment in packet header. You can set the value in set_body() and get the value using get_len().
- ✓ uint32_t* body: Packet body.
- ✓ set_body(uint16_t len, uint32_t* body): This function first sets the length value as *len*. It allocates memory for body depending on the length value. And it copies the input to *this->body*.
- ✓ get_body(uint32_t* output): This function copies body to the output. Note that you should allocate memory for *output* before using this function.
- ✓ print_packet(): You can use this function to check the packet.
- ✓ delete_body(): You can free the memory of the body.

Functions in class *Routing_Info* and its inherited classes

These variables and functions are in *routing.hpp* and *routing.cpp*.

- ✓ IP_prefix ip_prefix: IP prefix for the target. You can set the value using set_IP_prefix() and get the value using get_IP_prefix().
- ✓ uint32_t gateway: IPv4 address for the next hop. You can set the value using set_gateway() and get the value using get_gateway().
- ✓ uint32_t total_metric: The sum of the metrics a packet takes to get there. You can set the value using set_total_metric() and get the value using get_total_metric().
- ✓ uint32_t* additional_data: Type of additional data in Routing_Info is depending on the inherited class. *Routing_Info_DV* uses *metric_via_neighbor*, *Routing_Info_LS* uses *metric_to_AS* and *ASN*, *Routing_Info_PV* uses *path_to_AS*. enum additional (= add_t) could help you for additional_data.
- ✓ set_additional_data(add_t ad, int num_data, uint32_t* additional_data): Allocate memory of size *num_data* for ad. Then copy input value additional_data to the variable in class.
- ✓ set_additional_data_idx(add_t ad, int idx, uint32_t additional_data): Change idx-th data of the additional data. Note that you must use this function after set_additional_data.
- ✓ delete_additional_data(add_t ad): Free the memory of the additional data.
- ✓ get_additional_data_size(add_t ad): Return the size of additional data
- ✓ get_additional_data_idx(add_t ad): Return the index-data of the additional data.

Functions in class *Routing_Table* and its inherited classes

These variables and functions are in *routing.hpp* and *routing.cpp*.

- ✓ std::vector<Routing_Info *> rt_table: Autonomous system stores routing table using STL vector of pointer of Routing_Info.
- ✓ Routing_Info* init_and_add_rt_info(): This is a perfect virtual function. Each inherited class will push back pointer of corresponding Routing_Info. Then it returns that pointer.
- ✓ int get_size(): This function returns size of rt_table.
- ✓ Routing_Info* get_rt_info(): This function returns *idx*-th index of rt_table.
- ✓ int find_idx(uint32_t ipv4): This function returns index that index includes ipv4. If there is not range that covers ipv4, it will return -1.

Functions in class *Auto_sys* and its inherited classes

These variables and functions are in *auto_sys.hpp* and *auto_sys.cpp*.

- ✓ uint32_t ASN: Identifier of autonomous system. set_ASN() set the value, and get_ASN() get the value.
- ✓ IP_prefix IP: Range of IP address that autonomous system possesses. set_AS_IP() set the value, and get_AS_IP() get the value.
- ✓ uint32_t num_AS: The number of autonomous systems in the whole network. set_num_AS() set the value, and get_num_AS() get the value. Note that these functions only work in AS_Dist_Vect and AS_Link_Stat.
- ✓ Forward_Table fw_table: Forwarding table of autonomous system.
- ✓ std::vector<Auto_sys*> neighbor_AS: Array of pointer of neighboring AS. fw_table and neighbor_AS is same index ordering. Just use this pointer when you send packet to the neighbor_pkt.
- ✓ int num_nb: The number of neighboring AS. It is size of neighbor_AS and fw_table.
- ✓ std::queue<Packet> neighbor_pkt: Buffer of packet sending from neighboring AS.
- ✓ void add_fw_table(Forward_Info fw_info): Pushing back fw_info into fw_table.
- ✓ void change_fw_info_metric(int idx, uint32_t metric): Call the function Forward_Table::change_fw_info_metric(int idx, uint32_t metric).
- ✓ int get_fw_table_len(): Return the size of fw_table.
- ✓ int find_fw_info_idx(uint32_t gateway): Return the index that gateway is included.
- ✓ void add_neighbor(Auto_sys* neighbor): Pushing back neighbor into neighbor_AS.
- ✓ int get_num_neighbor(): Return num_nb.
- ✓ Routing_Table_(DV/LS/PV) rt_table: Routing table of AS. Different routing algorithms require different types of tables. rt_table is in inherited classes.
- ✓ int get_rt_table_len(): Return the size of rt_table.
- ✓ Routing_Info* get_rt_info(int idx): Return the idx-th data in rt_table.
- ✓ void print_rt_table(): Print routing table in terminal.
- ✓ void init_rt_table(), void change_rt_info(IP_prefix ip, uint32_t metric): These functions are pre-implemented. Referring these functions, you should implement update_rt_table() and send_packet_neighbor().

Functions in class *Network* and its inherited classes

These variables and functions are in *network.hpp* and *network.cpp*.

- ✓ uint32_t num_AS_net: The number of autonomous systems in network.
- ✓ std::vector<Auto_sys *> AS: Array of pointer of autonomous system.
- ✓ void simulation(char* filename): First, make network using make_network() function. Second, make routing table of every AS using init_simulation. Third, scan the command in terminal and get results based on the command.
- ✓ void make_network(char* filename): Make the configuration of simulation following the file ./graph_csv/filename. make_AS(), make_link() help this function.
- ✓ void malloc_AS(): When we push back pointer of autonomous system, we should push pointer different class. We can do that using this virtual function.
- ✓ void init_simulation(): Following logic flow shown in Figure 1, make routing table.
- ✓ void change_simulation(int as_a, int as_b ,uint32_t metric_link): If there is link between AS as_a and AS as_b, It will change the metric of the link to metric_link. Then following logic flow shown in Figure 1, make new version of routing table.