Electrical & Computer ENGINEERING

18-441/741: Computer Networks

Project 2: Content Distribution

Questions? Email TAs

1. Introduction

Content in the Internet is not exclusive to one computer. For instance, an episode of your favorite sitcom on Netflix will not be stored on one computer, but a network of multiple machines. In this project, you will create an overlay network that connects machines that replicate common content. You will implement a simple link-state routing protocol for the network. We could then use the obtained distance metric to improve our transport efficiency.

In a Nutshell: You are required to write a program called "**content_server**" in **Python**. We will run this program on multiple computers; connections between nodes follow an **undirected**, **connected** graph (there will not be multiple islands of nodes where islands are not connected), in which nodes are these computers and edges depict their distance metrics. Each computer runs the same program (i.e., **content_server**) with a corresponding configuration file that specifies its port number, address, and distance metrics of all its "neighbor" nodes (i.e., other computers). We will simultaneously execute this program on multiple computers. Each program must correctly infer the graph of the entire network in real-time, including the number and list of all nodes and the aggregate distance metrics of all nodes reachable from itself. In addition, we will terminate (or restart) some nodes at a random time, and your program must quickly respond to the corresponding changes in the graph (e.g., updating the number of nodes and the aggregate distance metrics).

2. Project Tasks

Your main goal is to create a program **content_server** in Python (i.e., content_server.py). A configuration file is specified when the program initially runs (different computers use different configuration files). After the program starts, it waits for additional command inputs via the keyboard (stdin), and outputs required information on the screen (stdout). It should output **nothing** unless an output is requested by a command input.

2.1 Configuration File

Your node should be able to read a configuration file located in the same directory in which we execute the server. **You can assume that the configuration file always exists there.** As an example, if we are to run a

program under /home/student/project2, with a configuration file named node21.conf, the command to start the server will be (note the "**-c**" here):

python3 /home/student/project2/content_server.py -c /home/student/project2/node21.conf

An example configuration file looks like this:

```
uuid = f94fc272-5611-4a61-8b27-de7fe233797f
name = node1
backend_port = 18346
peer_count = 2
peer_0 = 24f22a83-16f4-4bd5-af63-9b5c6e979dbb, ece003.ece.local.cmu.edu, 18346, 10
peer_1 = 3d2f4e34-6d21-4dda-aa78-796e3507903c, ece005.ece.local.cmu.edu, 18346, 20
```

1. **uuid** – The node's unique identifier. We will provide different identifiers for different nodes.
2. **name** – A user-friendly name used to call the node. We will provide different names for different nodes.
3. **backend_port** – Back-end port of the node (the default UDP-Transport backend 18346 is used here).
4. **peer_count** – The number of neighbors specified in this configuration, followed by **exactly** peer_count lines providing specific peer information. Each line has the format of:
   peer_x = uuid, hostname, backend_port, distance metric
   (note: x starts from 0 to peer_count-1; uuid, hostname, backend_port, and metric will appear in this order; commas will be used to separate them)

You can assume that all configuration files form a self-contained graph depiction, e.g., there will be no missing edges or nodes. Besides, field 1-3 (uuid, name, and backend_port) will always exist, and will appear **before** field 4. However, field 1-3 might not appear in the exact same order as above. In addition, the "peer_x" lines will always appear in the order from peer_0 to the last one (peer_count-1). Make sure, however, that your program ensures some robustness by properly handling whitespaces (e.g., missing ones) and line endings (e.g., '\r\n' lines and '\n' lines). **You don't have to update the configuration files even if the network changes in the future.**

2.2 Node Identifier

Each peer node should have a unique identification, and we use UUID (Universally Unique Identifier), a 16-byte (128-bit) number. You can find multiple online tools to help generate some example UUIDs when you test your program. During a node termination or restart, this field should not change.

Your program should support a command input "**uuid**" after it is started and respond with the UUID of this node in the format of a Python dictionary. The returned string should be evaluable (e.g., by using ast.literal_eval()). Below is an example; please follow the format, letter cases, and field names.

Keyboard Input: uuid<newline>

Response: {"uuid": "f94fc272-5611-4a61-8b27-de7fe233797f"}<newline>

2.3 Reachability

A node is expected to let its neighbors know that it is online (reachable) by periodically sending keepalive messages to them; meanwhile, a node learns that its neighbors are down if it misses their keepalive messages. Specific implementations of such keepalive messages are up to you. For example, you can make nodes send keepalive messages to all its neighbors every 3 seconds and mark a certain neighbor as inactive (unreachable) if three consecutive keepalive messages from this neighbor are missed. Make sure you mention your design in the design document submitted along with your code.

Your program should support a command input "**neighbors**" after it is started and responds with all the **active** neighbors of this node, in the format of a Python dictionary that is evaluable. Below is an example; please follow the format, letter cases, and field names. You should make sure that fields "uuid" and "host" have string values, while fields "backend_port" and "metric" have number values.

Keyboard input: neighbors<newline>

Response: {"neighbors": {"node2": {"uuid": "24f22a83-16f4-4bd5-af63-9b5c6e979dbb", "host": "ece003.ece.local.cmu.edu", "backend_port": 18346, "metric": 10}, "node3": {"uuid": "3d2f4e34-6d21-4dda-aa78-796e3507903c", "host": "ece005.ece.local.cmu.edu", "backend_port": 18346, "metric": 20}}}<newline>

Note that here the information of neighbors is organized by their **name**s; however, you might have noticed that in the configuration files, the names of neighbors are not specified. Thus, to retrieve the names of neighbors, your program is expected to implement link-state advertisement messages in Sec 2.5 and use these messages to get the correct names.

2.4 Adding Neighbors

Your program should support adding neighbors while running, and this should be implemented in a single-lateral manner. Specifically, suppose node1 (f94fc272-5611-4a61-8b27-de7fe233797f) and node4 (3f686f60-1939-4d62-860c-4c703d7a67a6) are not specified as neighbors in the configuration file. After they start, you should be able to use command "**addneighbor**" at node1 to add node4 as its neighbor; meanwhile, node4 is expected to **automatically detect** the fact that node1 is added as its new neighbor. Both of them should start sending keepalive messages to each other afterwards.

The command "**addneighbor**" is used with several arguments, and outputs **nothing**. Below is an example; please follow the format, letter cases, field names, and whitespaces. No quotes will be used.

Keyboard input: addneighbor uuid=686f60-1939-4d62-860c-4c703d7a67a6 host=ece006.ece.local.cmu.edu backend_port=18346 metric=30<newline>

Response:

After command "**addneighbor**" is carried out, if the added node is active, then it should appear in the output of command "**neighbors**", as well as command "**rank**" and "**map**" in the following sections.

2.5 Graph Discovery and Link State Advertisement

In this part of the project, we will use our neighbors to discover the whole graph. In an analogy to the actual network infrastructure, we can view each neighbor as a router. Initially, our node only knows information about our neighbors. With the link-state routing protocol, it gradually discovers the rest of the network.

In a link-state routing protocol, only information about the network links (i.e., edges in the graph) is exchanged between routers. A complete map of the network is then reconstructed on **every** router, and each router computes its own routing table based on the map. This is in contrast to the distance-vector routing protocol where each node tries to share its routing table with its neighbors.

For our project, your program is expected to implement a **link-state advertisement** protocol. **Periodically**, or **when there is a change in a node's neighbor**, each node creates a link-state advertisement which contains:

- The identity of the node which produce the advertisement
- The identity of all its neighbors and their connectivity metric
- A sequence number which increments when the source node creates a new advertisement

The advertisement is then sent to all neighbors. Note that each node should remember, for every other node, the sequence number of the last link-state advertisement it received. Upon receiving a new advertisement message, the receiver should compare the advertisement's sequence number with what it had.

- If the sequence number is lower, the message should be discarded.
- If the sequence number is higher, the receiver should update its network map and forward a copy of the advertisement to its neighbors.

In short, each node not only creates link-state advertisement of its own, but also helps forward link-state advertisement of other nodes in the network. Although it only sends/forwards messages to its neighbors, it learns the existence and connectivity metric of other nodes in the network, which are not its direct neighbors.

Similar to the keepalive messages, specific implementations of such link-state advertisement messages are up to you. For example, you can make nodes send advertisement messages to all its neighbors every 3 seconds and when a new neighbor is added. Make sure you mention your design in the design document.

Note that in practice this mechanism may not be feasible on a large-scale P2P network where there are millions of nodes, but it is simple and sufficient on a small-scale deployment like our project. In general, you can assume that there would be at most 32 nodes when we evaluate your program.

Your program should support a command input "**map**" after it is started and responds with the network map, in the format of a Python dictionary that is evaluable. Below is an example (the corresponding graph is Figure 1); please follow the format, letter cases, and field names. Make sure that the metrics have number values.

Keyboard input: map<newline>

Response: {"map": {"node1": {"node2": 10, "node3": 20}, "node2": {"node1": 10, "node3": 20}, "node3": {"node1": 20, "node2": 10, "node4": 30}, "node4": {"node3": 30}}}<newline>

2.6 Priority Selection

Now that each node has the network map, they can construct their own routing table. Specifically, they calculate the distance metrics between themselves and all the other nodes in the network. To achieve this, your program is expected to implement algorithms (e.g., Dijkstra) to find the shortest distance.

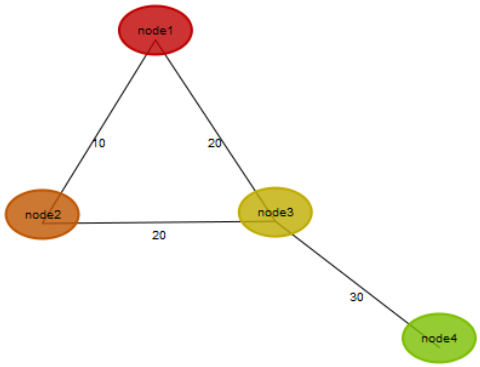

Figure 1: a simple network topology

Your program should support a command input "**rank**" after it is started and responds with the distance metrics from the current node to all the other nodes in the network in the format of a Python dictionary that is evaluable. Below is an example run on node1; please follow the format, letter cases, and field names. Make sure that the metrics have number values.

Keyboard Input: rank<newline>

Response: {"rank": {"node2": 10, "node3": 20, "node4": 50}}

2.7 kill

At any time, the command input "**kill**" to the program terminates it. It has no output.

This command is designed such that your program can safely release resources and exit – we will only terminate your program via "**kill**" during evaluation. Therefore, make sure you implement this properly; otherwise, if your program cannot be terminated as requested, you might fail the evaluation of "**neighbors**", "**map**", and "**rank**".

On the other hand, however, this command should not trigger messages to neighbors that indicate the termination; instead, neighbor nodes should discover the termination via keepalive messages. Imagine in a real-world scenario where some servers experience a power outage – they will never have a chance to predict this and inform their neighbor nodes. If your "**kill**" command triggers unwanted messages, it is considered as a failure and points will be deducted accordingly.

Keyboard Input: kill<newline>

Response:

3. Submitting Your Project

You will submit your project to Gradescope and we will test your code there. Please follow the exact instructions listed below. The place where you upload your files is referred to as the submission root directory.

1. Your **design document (PDF)**. Please include this under the submission root directory (uploading the file directly) and name it as **design.pdf**. A mismatched name will **not** be considered as part of your submission. Tell us about the following in your design doc:
    a. Your keepalive message implementation.

b. Your link-state advertisement implementation.
c. Libraries used (optional; name, version, homepage-URL; if available).
d. Extra capabilities you implemented (optional, anything which was not specified in this doc).

2. Your **code submission**. We accept Python submissions for this project. The main file that starts your server should be named as **content_server.py**, and as mentioned above, an example call will look like this:

   python3 /home/student/project2/content_server.py -c /home/student/project2/node21.conf

   In this project, you are expected to implement the keepalive and link-state messages from scratch without using existing libraries that achieve that in one shot. Meanwhile, you are also expected to implement the distance calculation on your own (e.g., write your own Dijkstra algorithm implementation if you choose that).

   For any python module codes that your server needs to call, please submit them along with your content_server.py under the submission root directory. Note: do not submit the executable files. This will not be considered as a valid submission, and we will not evaluate.

3. You do **not** need to submit any configuration files. During evaluation, we will use our configuration files.

4. Grading

Partial credits will be available based on the following grading scheme:

Items	Points (100 – **441/741**)
Logistics	**Total: 10**
- Code submission & compilation	5
- Design document submission	5
Functions	**Total: 90**
- Identity (uuid)	5
- Reachability (neighbors)	10
- Network change 1 (addneighbor)	10
- Link state advertisement (map)	30
- Priority rank (rank)	30
- Network change 2 (kill)	5