15-441: Computer Networks, Fall 2012
Project 2: Distributed HTTP

TAs: Onha Choe (ochoe@andrew.cmu.edu)
Yoshi Abe (yoshiabe@cs.cmu.edu)

Assigned: October 2, 2012
Checkpoint 1 due: October 9, 2012
Checkpoint 2 (final version) due: October 23, 2012

1 Introduction

In this project you will be implementing a Distributed HTTP service on top of the Liso Web server that you built in Project 1. This project has inspirations from Content Centric Networking where as a client you are interested in fetching a particular object (say midterm- soln.pdf). This object may be hosted in multiple servers and the client does not care from which machine the data is served. For the sake of simplicity, throughout this project, we assume that there is some out of the band scheme (which we are not worried about) to specify globally unique names for objects and if two different hosts have objects with the same name, they both contain the same content (eg. if 172.168.1.2:8080, 172.163.1.2:8080 and 172.168.1.4:8080 are serving midterm-soln.pdf, fetching it from any of them would result in downloading the same file). You may in fact want to get it served from the nearest server. In order to perform resource discovery (i.e. which all servers have midterm-soln.pdf), choose the nearest server amongst them and find the next hop to get to it, you will be using a modified version of OSPF to do reliable flooding.

For completing this project, you will need to build a Routing Daemon written in C which will be coupled to the Liso Web server. You will also be implementing a simple Flask Application to run on top of your Liso Server. You will not be required to make any major modifications to the Liso server for this project. But we assume that your Liso Server is feature complete (i.e. with CGI functionality working to run the Flask application). If you are not comfortable with using your implementation of Liso, you may use a replacement web server of your choice (Apache, default web server that comes with Flask etc).

2 Logistics

- This is a group project done in groups of two students. You must find exactly one partner for this assignment. In case there are an odd number of people in the class and you are left out, please contact the TAs.
- All submissions are due no later than 11:59 PM on the due dates. Turn in your submission by creating a directory containing all the applicable code and documentation under /afs/andrew/course/15/441-f12/handin/project2/{checkpoint1, checkpoint2}. The directory name should be"⟨ID1⟩_⟨ID2⟩", where ID1 and ID2 are the andrew ID's of the team members.

- Checkpoint 1: implement the flask web application and the router daemons interface to the flask application.
- Checkpoint 2 (final submission): implement the OSPF routing algorithm and object-name-based nearest server lookup on the routing daemon.
- Resources and announcements, including help session hours, will be made available on the course web site. So please remember to check it constantly.

3 Overview

There are three different components that together form a single node for this project: Liso server, a flask application running over CGI on the Liso Server and a Routing Daemon. There will be multiple such nodes connected by some topology. Each of these nodes host certain objects and if any of these nodes is contacted (via the flask application) for an object, the node should be able to serve the object if available locally or route it to the nearest available node that has the object.

The routing daemon will be a separate program from your Liso server. Its purpose is to maintain the routing state of the network (e.g., build the routing tables or discover the routes to destinations). It also enables resource lookup (i.e., given an object name, it should be able to find the nodes that have the particular object). It should be able to find out the nearest node that has the object. If the object is available on this particular node itself, then it returns the URI to the content, if not it needs to return information regarding the next hop to the closest node hosting the particular object.

The flask application acts as an interface between the client (web browser) and the routing daemon. It queries the routing daemon to find out the next hop information to fetch the object requested by the client. Note that the routing daemon only provides the routing information, the flask application does the forwarding based on the next hop information provided by the routing daemon.

The Liso server is required to host the flask application, but there is no explicit changes required to be made to your server. A node hence will have a Liso daemon and a routing daemon running and the flask application will be executed whenever a client wants to do a lookup for an object using the distributed http service.

In your implementation, the routing daemon will communicate with other routing daemons (on other nodes) over a UDP socket to exchange routing state. It will talk to the flask application running on the Liso server that is on the same node as it via a TCP socket. In order to find out about the network topology, each routing daemon will receive a list of neighboring nodes when it starts. In this project, you can assume that no new nodes or links will ever be added to the topology after starting, but nodes and links can fail (i.e., crash or go down) during operation (and may recover after failing).

What is a practical usage scenario for this? Anyone running large distributed web applications would require accessing objects remotely. They often have to pull in objects from lots of servers at once, as quickly as possible, in serving a single page – for example, Facebook, Google, etc. The point-to-point routing that we use in this project could be optimized out as described, but we are keeping it for now to demonstrate routing.

4 Definitions

- **Node**: The routing daemon and the Liso server with the flask application pair running together that is part of the larger network. In the real world, a node would refer to a single computer, but we can run multiple virtual nodes on the same computer since they can each run on different ports. Each node is identified by its nodeID.
- **NodeID**: A unique identifier that identifies a node. This is an unsigned 32-bit integer that is assigned to each node when its Liso server and routing daemon start-up.
- **Neighbor**: Node 1 is a neighbor of node 2 if there is a virtual link between 1 and 2. Each node obtains a list of its neighbors nodeIDs and their routing, local and Liso ports at start-up.
- **Object**: A file as a null terminated character string. To simplify the project, we assume that the object name will be at most 9 characters long and may not contain spaces or special characters.
- **Nexthop-URI**: The routing information returned by the routing daemon to the flask application will be in the form of a URI which we call Nexthop-URI. In case the object is available locally on the node, doing a GET on the Nexthop-URI will retrieve the content. If not, doing GET on the URI will invoke the flask application running on the next hop neighbor on the shortest path to the object. Note that to the flask application this does not make a difference, the routing daemon has to handle the case of checking whether the object is available locally or not.
- **Liso port**: The TCP port on the Liso server that talks to clients.
- **Routing port**: The UDP port on the routing daemon used to exchange routing information with other routing daemons.
- **Local port**: The TCP port on the routing daemon that is used to exchange information between it and the flask application. For example, when the flask application wants to find out the route to remote user, it queries the routing daemon on this port. The socket open for listening will be on the routing daemon. The flask application will connect to it.
- **OSPF**: The shortest path link state algorithm that you will implement.
- **Routing table**: The data structure used to store the next hops that packet should take used in OSPF.

5 Flask Applicaton

Here is the workflow for the flask application.

1) The server is supposed to handle CGI requests at /rd/* and they should be redirected to the flask web application that you have written. Typically the distributed http service lookup request will be a GET request at /rd/⟨port⟩/o1.html (e.g., /rd/7689/o1.html). Here the rd stands for routing daemon and indicates that we want to use the distributed http to retrieve the object (here o1.html). The ⟨port⟩ here stands for the local port (for communication between the flask application and the routing daemon). The request at the front end node is generated by a web form where the user can enter the object that it needs to retrieve using the distributed http service

and the front end node receives the request.

2) The flask application receives the object name from the HTTP GET request and now needs to connect to the routing daemon using the local port (specified in the URL). It then needs to send the following a RDGET request which is described as follows:
Request: GETRD ⟨length⟩ ⟨object⟩
Response: OK ⟨length⟩ ⟨Nexthop-URI⟩

Example:
Request: GETRD 7 o1.html
Response: OK 39 http://176.8.34.1:8000/rd/6789/o1.html
(Here the node with ip:port 176.8.34.1:8000 is on the next hop to the nearest server that has the object o1.html and hence the request is redirected to the routing framework on that node, because the link has /rd/*.)
Request: GETRD 7 o1.html
Response: OK 37 http://localhost:8080/static/o1.html
(Here the node has the o1.html locally and hence gives the URI corresponding to it.)

Note here that this logic is completely done by the routing daemon and the flask application just has to get the URI and does not have to care whether it is calling another routing lookup or whether the file is available locally.

3) The Python app should do a urllib.urlopen() on the returned ⟨Nexthop-URI⟩.

4) Similarly the web app should also handle adding new files using POST operation using an ADDFILE operation. Addfile works similarly, except it is a POST operation, not a GET and the POST URL will be of the form: /rd/addfile/⟨port⟩/⟨object⟩ When POSTing your web app should save the object into the static folder with the name being the sha256sum (see Python hashlib) of the file contents and once the file is saved to disk, contact the local Routing Daemon at the port specified by the URL using TCP connection and do the following:
Request: ADDFILE ⟨length⟩ ⟨object⟩ ⟨length⟩ ⟨local relative path⟩
Response: OK 0

Example:
Request: ADDFILE 7 o1.html 13 /static/8fb7c
Response: OK 0
(Here the o1.txt is the identifier of the object and the actual file is placed at relative path location /static/8fb7c. With this information, the routing daemon should later be able to reconstruct the URL to get this by doing http://localhost:8080/static/8fb7c.)

You can find a detailed description of this protocol in the Labs page on the course web site.

6 Routing Daemon

The routing daemon on start up reads two files – configuration file and file list file. The configuration file describes the neighborhood of a node. The neighborhood of a node 1 is composed by node 1 itself and all the nodes n that are directly connected to 1. For example, in Figure 4, the neighborhood of

node 1 is 1, 2, 3. The file contains a series of entries, one entry per line. Each line has the following format:
nodeID hostname routing-port local-port server-port
where nodeID assigns an identifier to each node, hostname gives the name or IP address of the machine where the neighbor node is running, local-port gives the TCP port on which the routing daemon should listen for the python flask application, routing-port refers to the port where the neighbor node listens for routing messages and server-port refers to the TCP port on which the Liso server at that node listens for clients.

Similarly the file list file contains the list of objects and their corresponding relative file path, that are being served by this particular server on startup. Each line of the file is of the format object relative-file-path.

The routing daemon has to maintain the following information:

- Routing table containing node id and the next hop path. This is constructed from the Link State Announcements using OSPF algorithm (see next section for details). Each node should also know the cost to each of the other nodes in order to find the nearest node that has a particular object.
- Maintain a mapping from the local objects and their relative path on the server. This can be populated by reading the file list file on boot-up. Also when the flask application does ADDFILE call, it should make an additional entry for the new object, relative file path mapping.
- Should be able to find the nearest node that hosts a particular object. The list of nodes that host a particular object can be obtained by parsing the file list to identify the local files as well as by parsing the link state announcements that contain the list of files hosted on remote nodes. The nearest node amongst these must be chosen. In case there is a tie, the node with numerically lower nodeID must be chosen.

When the flask application does a GETRD request with an object name, the routing daemon has to to check if the object is local. If it is local then send the URI built from the relative file path, the peer name of the server and the Liso server port number. If the object is not available locally but available on a remote node then it needs to find the closest remote node on which the file is available using the routing table and then find the next hop to get to the node and redirect to the flask application running on that node. It then needs to construct the URI based on the peer name of the next hop neighbor, its Liso server port, its local port and the object name.

7 Link State Routing Protocol

7.1 Basic Operation

You will implement a link-state routing protocol similar to OSPF, which is described in the textbook in chapter 4, and in more detail in the OSPF RFC1. Note, however, that your protocol is greatly simplified compared to the actual OSPF spec. As described in the references, OSPF works by having each router maintain an identical database describing the networks topology. From this database, a routing table is calculated by constructing a shortest-path tree. Each routing update contains the nodes list of neighbors and list of objects available locally. Upon receiving a routing update, a node updates its routing table with the best routes to each destination. In addition, each routing daemon must remove entries from its routing table when they have not been updated for a long time. The routing daemon will have a loop that looks similar the following:

```
while (1) {
    /* each iteration of this loop is "cycle" */
    wait_for_event(event);

    if (event == INCOMING_ADVERTISEMENT) {
        process_incoming_advertisements_from_neighbor();
    } else if (event == IT_IS_TIME_TO_ADVERTISE_ROUTES) {
        advertise_all_routes_to_all_neighbors();
        check_for_down_neighbors();
        expire_old_routes();
        delete_very_old_routes();
    }
}
```

Lets walk through each step. First, our routing daemon A waits for an event. If the event is an incoming link-state advertisement (LSA), it receives the advertisement and updates its routing table if the LSA is new or has a higher sequence number than the previous entries. If the routing advertisement is from a new router B or has a higher sequence number than the previously observed advertisement from router B, our router A will flood the new announcement to all of its neighbors except the one from which the announcement was received, and will then update its own routing tables.

If the event indicates a predefined period of time has elapsed and it is time to advertise the routes, then the router advertises all of its objects, and links to its direct neighbors. If the routing daemon has not received any such advertisements from a particular neighbor for a number of advertisements, the routing daemon should consider that neighbor down. The daemon should mark the neighbor down and re-flood LSA announcements from that neighbor with a TTL of zero When your router receives an announcement with a TTL of zero, it should delete the corresponding LSAs from its table.

If the event indicates that a new object was added using ADDFILE, the router should send a triggered update to its neighbors. This is simply a new link state advertisement with a higher sequence number that announces the routers new state. If a node has not sent any announcements for a very long time, we expire it by removing it from our table.

If B receives an LSA announcement from A with a lower sequence number than it has previously seen (which can happen, for example, if A reboots), B should echo the prior LSA back to A. When A receives its own announcement back with a higher sequence number, it will increment its transmitted serial number to exceed that of the older LSAs.

Each routing announcement should contain a full state announcement from the router all of its neighbors and all of its objects. This is an inefficient way to manage the announcements, but it greatly simplifies the design and implementation of the routing protocol to make it more tractable for a 3 week assignment. Each time your router originates a new LSA, it should increment the serial number it uses. When a router receives an updated LSA, it recomputes its local routing table. The LSAs received from each of the peer nodes tell the router a link in the complete router graph. When a router has received all of the LSAs for the network, it knows the complete graph. Generating the object routing table is simply a matter of running a shortest-paths algorithm over this graph.

7.2 Reliable Flooding

OSPF is based upon reliable flooding of link-state advertisements to ensure that every node has an identical copy of the routing state database. After the flooding process, every node should know the exact network topology. When a new LSA arrives at a router, it checks to see if the sequence number on the LSA is higher than it has seen before. If so, the router reliably transmits the message to each of its peers except the one from which the message arrived. The flooding is made reliable by the use of acknowledgement packets from the neighbors. When router A floods an LSA to router B, router B responds with an LSA Ack. If router A does not receive such an ack from its neighbor within a certain amount of time, router A will retransmit the LSA to B. Using the LSA, each node should also be able to find the nodes on which a particular object is available. Once it knows the nodes, it can find the closest node (which could even be itself).

7.3 Protocol Specification

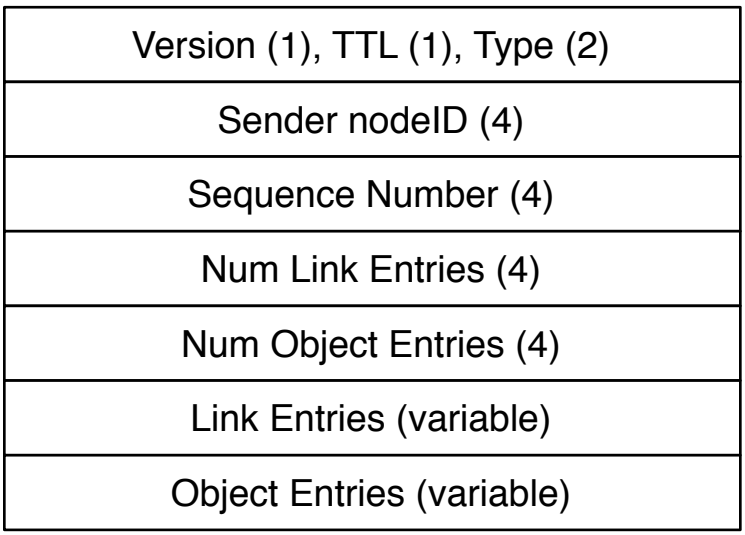

Figure 1: OSPF Packet Format - the figure shows the routing message format, with the size of each field in bytes in parentheses.

- **Version**: the protocol version, always set to 1.
- **TTL**: the time to live of the LSA. It is decremented each hop during flooding, and is initially set to 32.
- **Type**: Advertisement packets should be of type 0 and Acknowledgement packets should be of type 1.
- **Sender nodeID**: The nodeID of the sender of the message, NOT the immediate sender.
- **Sequence Number**: The sequence number given to each message.
- **Num link entries**: The number of link table entries in this message.
- **Num object entries**: The number of objects announced in this message.

- **Link entries**: Each link entry contains the nodeID of a node that is directly connected to the sender. Each entry in this field is 4 bytes in size.
- **Object entries**: Each object entry contains the name of the object that is hosted at the sender. These should be null-terminated strings.

All multi-byte integer fields should be in network byte order. An acknowledgement packet looks similar to an announcement packet, but it does not contain any entries. It contains the sender nodeID and sequence number of the original announcement, so that the peer knows the LSA has been reliably received.

7.4 Requirements

Your OSPF implementation should have the following features:

- Given a particular network configuration, the routing tables at all nodes should converge so that the forwarding will take place on the path with the shortest length.
- In the event of a tie for shortest path, the next hop in the routing table should always point to the nodeID with the lowest numerical value. Note that this implies there should be a unique solution to the routing tables in any given network.
- Remove the LSAs for a neighbor if it has not given any updates for some period of time.
- You should implement Triggered Updates (when a link goes down or when a new object is added at a node).
- If a node or link goes down (e.g., routing daemon crashes, or link between them no longer works and drops all messages), your routing tables in the network should re- converge to reflect the new network graph. You should not have to do anything more to make sure this happens, as the above protocol already ensures it.

Note that your implementation should follow the OSPF specification as precisely as possible, supporting its features to deal with corner cases. On the other hand, in this lab you do not need to implement the following:

- You do not have to provide authentication or security for your routing protocol messages.
- You do not have to jitter your timer with randomized times.

8 Implementation Details

Your routing daemon must be written in the C programming language. You are not allowed to use any custom socket classes or libraries, only the standard socket library and the provided library functions. You may not use the csapp wrapper library from 15-213, or libpthread for threading. We disallow csapp.c for two reasons: first, to ensure that you understand the raw standard BSD sockets API, and second, because csapp.cs wrapper functions are not suitable for robust servers. Temporary system call failures (e.g., EINTR) in functions such as select could cause the server to abort, and utility functions like rio readlineb are not designed for nonblocking code.

For implementing the flask web application, you will have to use python. If need be, you can write a C based web application but it might be more complex. You can take a look at the Python tutorial and the Flask tutorial online to create the simple webapp needed for this project.

We encourage the use of anything for testing. Use Wireshark, web browsers, Python scripts etc.

8.1 Compiling

You are responsible for making sure your code compiles and runs correctly on the Andrew x86 machines running Linux (i.e., linux.andrew.cmu.edu / unix.andrew.cmu.edu). We recommend using gcc to compile your program and gdb to debug it. You should use the -Wall and -Werror flags when compiling to generate full warnings and to help debug. Other tools available on the Andrew unix machines that are suggested are ElectricFence (link with - lefence) and Valgrind – use this with full leak checking to ensure you have no memory leaks. For this project, you will also be responsible for turning in a GNU Make compatible Makefile. See the GNU make manual for details. When we run make we should end up with the Liso web server compiled lisod and the router daemon compiled routed.

8.2 Command Line Arguments

The routing daemon will always have arguments – functional or not:

The Router Daemon needs to be run as follows:
usage: ./routed ⟨nodeid⟩ ⟨config file⟩ ⟨file list⟩ ⟨adv cycle time⟩ ⟨neighbor timeout⟩ ⟨retran timeout⟩ ⟨LSA timeout⟩

- **node id**: A unique identifier for a node (32 bit integer).
- **config file**: Name of the configuration file that contains information about neighboring nodes (including itself).
- **file list**: A file containing the object to relative path mapping of the file on the local web server.
- **adv cycle time**: In seconds, it determines the amount of time between each advertisement cycle. You may set it to 30.
- **neighbor timeout**: The elapsed time, in seconds, after which we declare a neighbor to be down if we have not received any updates from it. You may assume that this is a multiple of adv cycle time. You may set it to 120.
- **retran timeout**: The elapsed time, in seconds, after which a peer will attempt to retransmit an LSA to a neighbor if it has not yet received an LSA acknowledgement from it. You may set it to 3 seconds.
- **LSA timeout**: The elapsed time, in seconds, after which we expire an LSA if we have not received any updates for it. You may again assume this to be a multiple of adv cycle time. You may set it to 120.

The Liso server will be run as before (See Project 1 documentation).

9 Testing

Code quality is of particular importance for server robustness in the presence of client errors and malicious attacks. Thus, a large part of this assignment (and programming in general) is knowing how to test and debug your work. There are many ways to do this; be creative. We would like to know how you tested your code and how you convinced yourself it actually works. To this end, you should submit your test code along with brief documentation describing what you did to test that your server works. The test cases should include both generic ones that check the server functionality and those that test particular corner cases. If your server fails on some tests and you do not have time to fix it, this should also be documented (we would rather appreciate that you know and acknowledge the pitfalls of your server, than miss them). Several paragraphs (or even a bulleted list of things done and why) should suffice for the test case documentation.

10 Submission Format

Your submissions, for both Checkpoints 1 and 2, should contain the following files:

- **Makefile**: Make sure all the variables and paths are set correctly such that your program compiles in the hand-in directory – not just a local machine or account. The Makefile should, by default, always build an executable named lisod.
- **All of your source code**: Files ending with .c, .h, etc. only. No .o files or executables.
- **readme.txt**: File containing a brief description of your design of your current version of lisod.
- **tests.txt**: File containing documentation of your test cases and any known issues you have.
- **vulnerabilities.txt**: File containing documentation of any vulnerabilities you identify at each stage.

Make sure to ensure your code compiles on andrew machines, and describe how to run your code in readme.txt.

11 Grading

This point assignments may change slightly; this is just a rough estimate of the actual grade scheme.

- **Flask Web Application**: 20 points.
  The grade in this section is intended to reflect your ability to write a simple flask web application.
- **Routing Daemon and Flask Interface**: 20 points.
  The grade in this section is intended to test the miniature protocol between the routing daemon and the flask web application and how well it does lookups.
- **OSPF Implementation**: 40 Points.
- **Robustness**: 10 points.
  Server robustness: 5 points, test cases: 5 points.
  Since code quality is of a high priority in server programming, we will test your pro- gram in

a variety of ways using a series of test cases. We will also look at your own documented test cases to evaluate how well you tested your work.

- **Style**: 10 points.
  Poor design, documentation, or code structure will probably reduce your grade by making it hard for you to produce a working program and hard for the grader to understand it; egregious failures in these areas will cause your grade to be lowered even if your implementation performs adequately.

  Document code using Doxygen-style comments. In some of our structured code examples, we showcase an underlying logging facility that logs to a configured file. Use something similar to this to keep traces of your server and debug.

12 Extra Credits

You can gain extra credits for this project. You can choose *at most two out of the three options* below and gain extra credits. Notice that not all options have the same points.

- **Caching**: 5 points.
  Extend your implementation of the flask application to also perform caching of the objects that it fetches from remote nodes (because it is not available locally). The node should store the object to a file in the static folder under the name of the sha256sum of the contents of the object and then do an ADDFILE request to the routing daemon with the object name and the relative path. You also have to do triggered LSA update with the latest object entry announcement.

- **Longest Prefix Match**: 10 points.
  Now assume that the object names have a directory structure: e.g., /cmu/csd/yabe etc. Nodes can advertise that they contain all objects under a particular object directory. That is, if a node advertises that it contatins /cmu/ or /cmu/csd/, it implies that it contains all directories and file objects in that directory path. i.e., it is hosting both the above file objects. So now the object announcements may contain both directory path announcements as well as more specific individual object announcements.

  You need to extend your current implementation to do longest prefix match. i.e., If node 1 announces that it is hosting /cmu/, node 2 announces that it is hosting /cmu/csd/ and node 3 announces that it is hosting /cmu/csd/yabe, lookup for /cmu/csd/yabe should now select node 3 irrespective of whether node 1 or 2 is closer than node 3. The lookup workflow should be modified as follows:

  1) Check if the object is available locally and serve it if so.
  2) If not, do longest prefix match, and among the nodes that have the longest match, route to the closest node.
  3) If there are any ties, route to the node with numerically lower nodeID.

- **Persistent Connections and Pipelined Requests**: 10 points.
  Extend the flask application to maintain a maximum of 10 worker threads to handle 10 persistent TCP connections with the routing daemon. These connections may be formed as and when requests come into the system. Perform pipelined GETRD and ADDFILE requests on each of these connections.