15-441 Computer Networks – Project 4
Distributed Document Routing and Forwarding

Lead TA: Julio López (jclopez+15-441@andrew.cmu.edu)

Assigned: Monday, November 12 / 2002
Due date: Monday, December 2 / 2002

1 Introduction

In this assignment you will be writing a distributed document routing and forwarding system similar in style to Gnutella. The goals of this assignment are that you:

1. Learn the principles of distance vector routing protocols.
2. Learn the principles of packet forwarding and analyze the tradeoffs of address lookup alternatives.
3. Explore the benefits of caching in document networks with a large number of documents.

The purpose of the distributed document system is to be able to share documents (i.e., files) between the participants of the system without the need of a centralized server or repository.

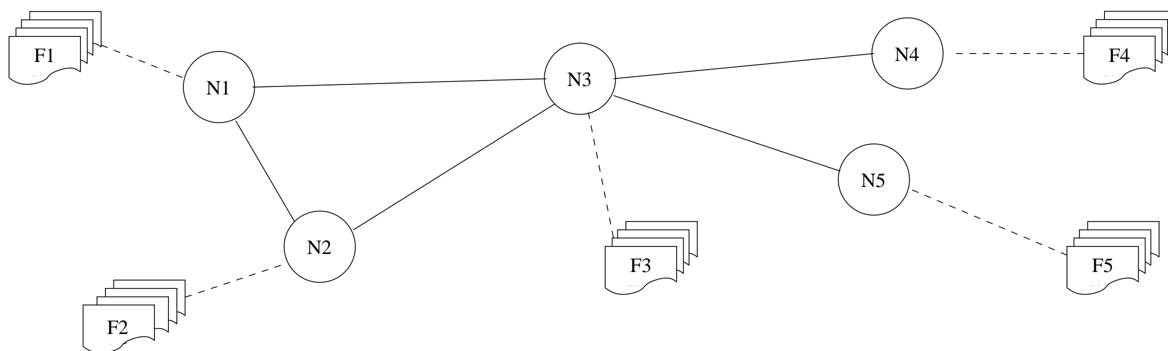

Figure 1: Sample document network.

The system is composed by a set of nodes interconnected by *virtual* links in an arbitrary topology. Each node is a process that we will call the *Document Network Routing Engine Daemon* or `dnred` for short. Each node makes a set of documents available to the system. Figure 1 shows a sample distributed document system composed by 5 nodes $\{N1, ..., N5\}$. The solid lines represent virtual links between the nodes. Node Ni publishes a set of files, denoted by Fi, to the system. The dashed lines connect the nodes to their respective file sets.

The usage model is the following: A user looking for a document uses a client program to submit a query to the system. The client program contacts the local node (located in the same machine) and submits a query packet to the node. The request travels through the system until it reaches the node containing the document. In order for the system to satisfy a user's request, each node in the system performs functions similar to the ones performed in the network layer, namely *forwarding* and *routing*. Forwarding is the action performed by each node to guide a packet toward its destination. When a node receives a packet, it looks up the packet's destination in a *forwarding table* and forwards the packet to the next node in the direction of the destination. Routing refers to the action of building the forwarding tables. The *routable objects* are the request and reply packets that travel through the document network. The packet

types are described in detail in Section 2.3. The *destination objects* are where the routable objects are addressed to. Depending on the type of packet, the destination of a packet can be either a document or a node. Figure 2 shows the details of a node in the network. Each node has a forwarding table, a.k.a. routing table, that it uses to find the next hop when forwarding a packet. The destinations in the forwarding table can be either documents or nodes.

Your task is to write the Document Network Routing Engine Deamon (`dnred`) that runs in each of the participating nodes. This project is divided in two parts: In the first part your program will use a standard distance vector routing protocol and standard forwarding techniques to implement the functionality of the system. In the second part, you will add caching functionality to the nodes to achieve better performance in this particular environment.

Section 2 describes the architecture of the routing engine and its requirements. Section 3 contains the implementation details. The grading criteria is described in Section 4. Section 5 provides suggestions on how to proceed and additional information. Also, at the end of this document there is an annotated reference we hope you find useful.

2 The Document Network Routing Engine `dnred`

When a user submits a request for a document to the network, the goal of the routing engine is to find a node containing the document. The job of your routing engine is to move the various types of packets through the network. In order to achieve this task, the routing engine keeps a forwarding table that allows it to locate the destination, i.e., the node containing a particular document.

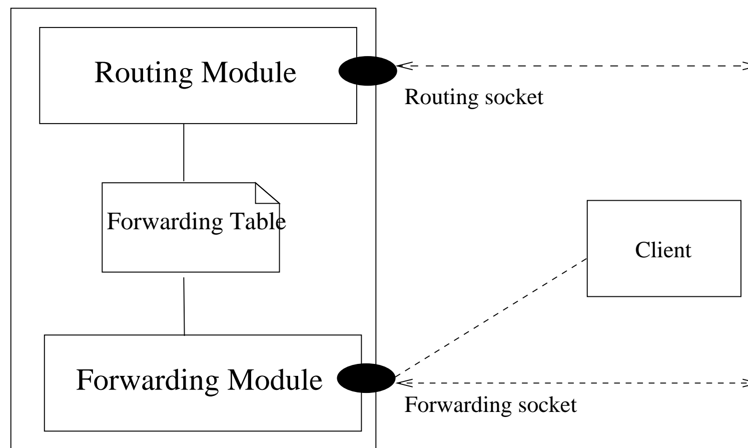

Figure 2: Sample node architecture.

Figure 2 shows the architecture of the routing engine. The functionality of the routing engine at each node is split into a *routing module* and a *forwarding module*. Each module uses a UDP socket to exchange information with neighbor nodes. The routing module takes care of building the forwarding table to route requests. It does so by executing a distributed routing algorithm. The forwarding module is responsible for sending packets to the next hop toward the destination. This module performs its task by looking up the destination in the forwarding table to obtain the address of the next hop. The forwarding table stores information about the routes to get to a destination (See Section 2.1). Section 2.2 describes the requirements of the routing module. Section 2.3 describes the forwarding module and packet types. To request a document, a client process sends packets to the engine's forwarding port. Section 2.4 describes how to process client requests.

Bootstrapping the Routing Engine In order to start populating the forwarding table, the routing module needs some information about the documents to publish and neighboring nodes. The routing engine obtains the names of the files to publish from a directory passed to the program in the command line arguments (See Section 3.2). The information about the neighbors is stored in a configuration file. The name of the configuration file is passed to the routing engine as a parameter in the command line. The configuration file contains an adjacency list of the neighboring nodes. The routing engine must read and parse this configuration file when it starts. Section 3.3 describes the format of the configuration file.

2.1 The Forwarding Table

The forwarding table stores information to locate a particular document or node. Each entry in the table should contain at least the following information:

Destination: Name of the destination node or document.
Next hop: Id of the next node in the path to reach the destination.
Distance: Distance to the destination in number-of-hops.

Sometimes the forwarding table will be used to locate a document and sometimes to find a return path to a node, thus the table contains entries with both types of destinations, i.e., documents and nodes.

Destination	Next hop	Distance
document1-at-node1	N1	2
document2-at-node1	N1	2
document1-at-node2	N2	1
document1-at-node3	N3	2
document1-at-node4	N3	3
document1-at-node5	N3	3
:128.2.201.2:20200	N1	1
:128.2.201.3:20202	N2	0
:128.2.201.4:20204	N3	1
:128.2.201.5:20206	N3	2
:128.2.201.6:20208	N3	2

Figure 3: Sample Forwarding Table

Figure 3 shows an example of a simplified forwarding table. Document destinations are specified by a character string containing the name of the document. Node destinations can be stored in the following way: use *special document names* to represent node addresses. To represent a destination node address create a string of the form:

`:dotted-decimal-node-ip-address:node-port`

Then, store the string in the forwarding table. For example, to store an entry with destination a node with IP address 128.2.201.2 and port number 20200, create the string "`:128.2.201.2:20200`" and store it in the entry's destination field.

Q: How does the routing engine find the route to a destination? If the destination is a document, it uses the name of the document to locate the appropriate entry in the forwarding table. If the destination is a node, then it converts the node's IP address and port to the string "`:node_ip:fwd_port`" and uses this string to look up the destination in the forwarding table.

You have complete freedom to choose the design of your forwarding table, e.g., linked list, binary tree, array, etc. A word of advice, your internal representation of the forwarding table should not be exposed to the rest of your routing engine. You would not want to have to change your entire program should you decide to change the implementation of the forwarding table to improve performance. Here is a suggested API to access the forwarding table:

- `ft_add_route(const char* dest, unsigned short next_node_id, int distance);`
- `ft_delete_route(const char* dest, unsigned short next_node_id);`
- `ft_modify_route(const char* dest, unsigned short next_node_id, int distance);`
- `ft_get_route(const char* dest, unsigned short* next_node_id, int* distance);`

2.2 Part 1: Routing Module – Distance Vector Protocol

The routing module implements a protocol to exchange routing information with neighboring nodes. In the first part of the project your routing engine will use a distance vector routing protocol to update the forwarding tables of the participating nodes. You may follow the RIP v2 specification [8, 10].

Here is a list of requirements, assumptions and simplifications for your routing protocol:

- Your routing decisions will be based on hop counts as opposed to some link cost.
- Your implementation of the routing protocol must ensure routing re-converges if communication with a node in the network is lost (e.g., a node goes down or a network failure occurs) or if communication with a node is reestablished (e.g., a node comes back up).
- The routing protocol should handle the *count-to-infinity* problem by using *Split Horizon with Poisoned Reverse* and *Triggered Updates* as described in the RIP v2 RFC [10] and the class textbook [9].
- You do NOT have to provide authentication or security for your routing protocol messages.
- You only need to store the single best route to a given destination document.
- Document names are unique. A document is stored in only one node and its location does not change over time. Documents are only added to or removed from the system. It is not possible to remove a document from one node and store it in a different node at a later time, it can only be stored back in the same node.
- All documents have equal popularity, that is the request rate for all documents is the same. This is not really important for part 1.

Executing the Routing Protocol The main loop of your routing module could look something like this pseudocode:

```
while (1) {
  /* each iteration of this loop is a ''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();
    set_hop_counts_to_infinity_for_all_routes_through_down_neighbor();
  }
}
```

Let's walk through each step. First, our routing module waits for an event. If the event is an incoming advertisement, it receives the advertisement and updates its forwarding table if any advertised route is new, or better than the current route based on hop counts. In the event of a tie in hop counts for a given destination it always chooses the route where the address of next hop (i.e., ip_addr:port) has the lowest numerical value.

If the event indicates that a predefined period of time has elapsed and it is time to advertise the routes, then the routing module advertises all of its routes (conceptually, its entire forwarding table) to all of its direct neighbors by sending the information in UDP packets to the neighbors' addresses.

Finally, if our routing module does not received any advertisements from a certain neighbor for a number of advertisement cycles [1], it considers communication with the neighbor to be broken and sets all routes through that neighbor to have a hop count of infinity. The routing module propagates this information it next advertises its routes.

2.3 The Forwarding Module

The forwarding module waits for incoming packets on a UDP port, i.e., the forwarding socket. The module handles the incoming packets and processes them according to their type. In order to forward a packet, this module looks up the destination in the forwarding table and obtains the address of the next hop, then it sends the packet to the next hop over UDP using the forwarding socket.

[1] The `-e entry-timeout` command line argument specifies when it is time to consider a neighbor to be down and expire the corresponding entries in the forwarding table

2.3.1 Packet format:

source IP	dest IP	source port	dest port	seq. number	type	TTL	name length	doc. name	data

Figure 4: General packet format

Figure 4 shows the general format of a packet. Below is a description of the fields:

- **Source Node IP Address**: Length: 4 bytes. This field contains the address in network byte order of the source node (or client) that initiated the transaction. This is not the address of the previous node in the forwarding path, which can be obtained through different means, i.e., the sockets API.
- **Destination Node IP Address**: Length: 4 bytes. This field contains the address in network byte order of the destination node (or client) that initiated the transaction. This is not the address of the next node in the forwarding path, which can be obtained through different means, i.e., from the forwarding table. This address can have a value of 0.0.0.0, in which case the name of the document is used to route the packet to its destination.
- **Source Node Port**: Length 2 bytes. This field contains the port number in network byte order of the node that initiated the transaction.
- **Destination Node Port**: Length 2 bytes. This field contains the port number in network byte order of the destination node. If it is 0, then the document name is used to route the packet.
- **Sequence number**: Length 2 bytes. This field contains a sequence number in network byte order. The sequence number is assigned by the packet's source node.
- **Type**: Length: 1 byte. This field specifies the packet type, which can be one of the following ERROR, LOOKUP, QUERY, REPLY, DATA_REQUEST, DATA_REPLY. The forwarding engine performs different actions depending on the packet type. These actions are described below (Section 2.3.2).
- **TTL**: Length: 1 byte. This field specifies the time to live of the packet. If this field has a value of 0 before reaching the destination, the packet is discarded and a packet containing an error code is sent back.
- **Name Length**: Length: 1 byte. This field specifies the length of the next field, i.e., the document name.
- **Document name**: Length: variable (See `Name-Length` field). This field contains the name of the destination document the transaction refers to. The field contains a character array of size `Name-Length`, NOT including a terminating NULL character.
- **Data**: Length: variable, can be 0. This field contains additional data depending on the type of packet.

2.3.2 Packet Types and Forwarding Actions

In all cases, the forwarding module will decrease the TTL field and if the TTL is 0, then the forwarding module will discard the packet. If the TTL is not 0 then the forwarding module performs an action depending on the packet type as follows:

- **ERROR**: Upon receipt of this type of packet, forward the packet toward the destination node using the destination :ip-address:port tuple. If a route is not known discard the packet. The data portion of the packet contains 2 fields. The first field is the destination address of the original packet that caused the error. This field is 6 byte longs containing the :ip-address:port tuple in network byte order. The second field is a 2 byte error code in network byte order. The following error codes are defined: TTL_EXPIRED, DEST_UNREACHABLE, DEST_UNKNOWN, TIMEOUT.

- **LOOKUP_REQUEST**: This packet is used to query a node's forwarding table directly. The destination address/port is 0/0. The document name field contains the name of the document to be looked up. The forwarding module will lookup the address in the forwarding table and return the corresponding entry in a LOOKUP_REPLY packet.

- **LOOKUP_REPLY**: This packet contains the response to a LOOKUP_REQUEST packet. The data portion of the packet contains a routing entry in the following format: 6 byte address in network byte order of the next hop to reach the destination and a 1 byte hop count that indicates the number of hops needed to reach the destination.

- **QUERY**: This packet contains a request to locate a document. The document is specified in the document name field. The destination address/port should be 0/0. The action for this type of packet depends on the following conditions:

  - If this node does not contain the document and an entry that matches the destination is found in the forwarding table, forward the packet to the next hop.
  - If this node does not contain the document and no entry is found in the forwarding table, discard the packet and send back to the source node an ERROR packet with a DEST_UNKNOWN error code.
  - If this node contains the document, generate a REPLY packet. The source address of the packet is this node's address, the destination address is the source address specified in the QUERY packet. Handle the REPLY packet as described below.

- **REPLY**: This packet is generated in response to a QUERY packet when a matching document is found. The data field of the packet contains the following fields: `Route_length` (1 byte) and an array of size `Route_length` containing the reverse route from the node originating the reply back to the source of the transaction. Notice that the minimum array size is 1, and the 0-th position in the array corresponds to the address where the document is stored. Upon arrival of one of these packet, the forwarding module will increase the `Route_length` field and add its own address to the end of the route array in the REPLY packet. Then it forwards the packet to the next hop in the path toward the destination address/port if known, otherwise the packet is discarded. Notice that this set of actions, if implemented correctly, effectively provide a reserve record path mechanism.

- **DATA_REQUEST**: This packet is similar to the QUERY packet and the actions taken by the forwarding module are the same except for the case when the requested document is found in a node. When the requested document is found in a node, then the routing engine generates a DATA_REPLY packet, including in the packet the address of this node in the source address/port fields. The DATA_REPLY packet also contains the size and the contents of the document.

- **DATA_REPLY**: This type of packet is generated in response to a DATA_REQUEST packet when a matching document is found in a node. The data field in the DATA_REQUEST packet contains the following fields: 4 bytes for the length of the document and the rest of the data field contains the contents of the document. You may assume all documents are less than 1024 bytes long. Upon receiving a DATA_REPLY packet, the forwarding module sends the packet to the next node in the path toward the destination address/port (originator of the transaction) if known, otherwise the packet is discarded and a DESTINATION_UNKNOWN error packet is sent back to the source.

2.4 Handling Client Requests

In order to test your system, you should build a simple client program. The client interacts with the routing engine by sending request packets to the engine's forwarding port. The client can send packets of type LOOKUP_REQUEST, QUERY and DATA_REQUEST to the routing engine. For simplicity, your routing engine is only required to support 1 client at a time, however, the client might send multiple packets simultaneously.

When a client sends a packet it specifies 0.0.0.0:0 as the source IP address and port. The routing engine gets the client's IP address and port from the UDP packet. It then replaces the source address in the packet with its own node address and port number. Also, the routing engine should remember the IP address and port number of the client, and the sequence number of the packet received from the client. Then it forwards the packet as usual. When a packet of type ERROR, REPLY or DATA_REPLY arrives to a node and the destination address matches the node's address

(i.e., this node is the destination of the packet), the routing engine should forward the packet to the client. Also, if a response for a packet has not arrived after 30 seconds, then the routing engine should send an ERROR packet to the client with a TIMEOUT error code.

2.5 Part 2: Caching

In Part 1, we assumed that all documents were equally popular. Now suppose that the document popularity exhibits a heavy-tailed distribution, i.e., a small percentage of the documents are very, very popular, whereas the large majority of the documents are not very popular. Also, the following restriction is relaxed, a document can exist in various nodes in the network simultaneously, more over, the system is allowed to copy and move documents from one node to another. The only restriction is that the origin node, i.e., the node where the document first appeared, must always keep a copy of the document unless explicitly deleted by the user.

You will add caching to the routing engine to improve performance by capitalizing on these properties of the system. The caching strategy that you will implement is as follows: Whenever the forwarding module receives a DATA_REPLY packet, it will store in its cache the document contained in the packet (i.e., the document's name and content). That is, the engine caches every document that it forwards. The replacement policy is LRU (Least Recently Used). If your cache is full, you will *evict* from the cache the document that has not been accessed for the longest period of time. The size of the cache is restricted by two parameters: The maximum number of entries that can be cached and the maximum size of the cache for document data. This values are specified in the command line of the routing engine.

3 Implementation Details

Details, details, details! This section provides additional information about the requirements of your routing engine.

3.1 Programming Guidelines

Your programs must be written in the C programming language. You are not allowed to use any custom socket classes or libraries, only the standard libsocket and the provided `librtrace` library. C++ submissions will not be accepted. **You can only use the Andrew Linux machines (i.e. linux.andrew.cmu.edu), for compiling and testing your program in this project**, this is because we are providing a binary library that has been compiled for Linux. So you are responsible for making sure your program compiles and runs correctly on these machines. We recommend using `gcc` to compile your program and `gdb` to debug it. You should also use the `-Wall` flag when compiling to generate full warnings and to help debug.

For this project, you will also be responsible for turning in a GNU Make (`gmake`) compatible Makefile. See the GNU make manual [4] for details.

IMPORTANT: These are the implementation requirements you should meet:

- The routing engine should use two (and only 2) UDP ports to exchange data with other nodes. One port should be used by the routing module to exchange routing messages, the other port should be used by the forwarding module to forward packets.
- You must use the `rt_sendto()` and `rt_recvfrom()` wrapper functions to send and receive data through the UDP sockets. These functions are defined in the `librtrace` library.
- Your routing module must call the procedure `rt_print_route_op()` whenever you change an entry in the forwarding table. The parameters of this function are described in the header file `rtrace.h`.
- Your routing engine **must implement** the `rt_print_table()` procedure. This procedure is called by the library to print the whole forwarding table of a node. The format of the output for the forwarding table information is described in section 3.5.

These trace items are written to a separate file for each routing process (named rt1.trace for node 1, etc.), and will be used for grading purposes; they are also helpful for debugging.

3.2 Invocation and Command Line Arguments

Your document network routing engine daemon, `dnred` must accept the command line arguments in any arbitrary order. Some arguments are optional, if an optional argument is left out when launching the process, the program should use the default value specified below. If a required argument is left out, the program should print a *usage message* and exit with an error code not equal to 0.

- `-f` *filename*: This is a required parameter. It specifies the name of the configuration file that contains the information about the neighbor nodes. Section 3.3 describes the format of this file.
- `-i` *node-id*: Sets the node id for this process. This is a required argument.
- `-p` *directory*: Specifies the name of the directory containing the documents that this node publishes. For example if the process is run with the following argument: `-p ./networks/project4/dir1`, then the routing engine reads the files from this directory and advertises their names in the routing protocol. This is a required argument.
- `-a` *seconds*: Specifies how often the routing process should advertise its forwarding table to its neighbors. This is an optional argument, the default value is 30 seconds.
- `-y` *infinite-value*: Specifies the number to use as infinite value. This is an optional parameter, its default value is 16.
- `-u`: This is an optional argument that indicates the use of triggered updates. If this flag is not present, the routing engine will not use triggered updates.
- `-e` *seconds*: This is an optional parameter and specifies the entry timeout value. It is used to expire old entries in the forwarding table. Default value 180 seconds.
- `-g` *seconds*: This is an optional parameter and specifies the elapsed time before an expired entry is deleted. Default 120 seconds.
- `-c` *cache-entries*: This is an optional parameter that indicates the number of entries to cache. The default value is 0, i.e., no caching.
- `-s` *size*: This is an optional parameter that indicates the cache's data size in KBytes. Default: 1 KB if caching is enabled.
- `-n` *number*: This is an optional argument that specifies the number of routing cycles to execute. By default it should be *infinity*, meaning the routing module should execute forever. Use this option while debugging your program to control how many iterations of the routing algorithm are performed. If this argument is specified, once the routing module performs the given number of routing advertisement cycles, it should print the forwarding table by invoking `rt_call_print_table()`, stop advertising routes and ignore other route advertisements. The process should not exit, it should continue forwarding packets with the current forwarding table.
- `-d`: When this option is present the process may print debugging messages to the standard error device (`stderr`). When this option is not present, the process should not print any output to the console.

3.3 Configuration File Format

This file describes the *neighborhood* of a node. The neighborhood of a node A is composed by node A itself and all the nodes n that are directly connected to A, i.e., there is a link between the node n and A. For example, in Figure 1 the neighborhood of node $N2$ is $\{N2, N1, N3\}$; the neighborhood of node $N5$ is $\{N5, N3\}$. The format of the configuration file very simple, so it is easy to parse. The file contains a series of entries, one entry per line. Each line has the following format:

```
node-id hostname route-port forward-port
```

The first field (`node-id`) assigns an identifier to each entry (node). The `hostname` field specifies the name of the machine where the neighbor node is running. The `route-port` specifies the port where the neighbor node listens for routing messages. Similarly, the `forward-port` specifies the port the neighbor node uses to receive and forward regular packets.

```
2 localhost 15202 15203
1 host1.domain 15200 15201
3 128.2.205.35 15204 15205
```

Figure 5: Sample configuration file for node $N2$.

Q: How does a node find out which ports it should use for routing and forwarding? When reading the configuration file if an entry's `node-id` matches the local `id` of the node, then the node uses the specified port numbers to route and forward packets. Figure 5 contains a sample configuration file corresponding to node $N2$ for the network in Figure 1. Notice that the file contains information about node $N2$ itself. Node $N2$ uses this information to configure itself.

3.4 The `librtrace` Library

This library contains support functions that you will use in your routing engine. Here is an overview of the routines and variables available in the library. For detailed information about the parameters refer to the `rtrace.h` file. When building your programs, make sure you link your executable against the most recent version of the library.

- `rt_init()`: This routine initializes the library. Call it when the program starts.
- `rt_sendto(...)`: Wrapper function for the `sendto()` system call. The parameters and semantics are the same as in the system call. This routine performs various logging functions before calling `sendto()`.
- `rt_recvfrom(...)`: Wrapper function for the `recvfrom()` system call. The parameters and semantics are the same as in the system call. This routine performs various logging functions before calling `recvfrom()`.
- `rt_print_route_op(...)`: Function to print route change events. The first parameter specifies the type of event, e.g., add, delete, modify, query, etc. Call this function when a modification is made to the forwarding table, i.e., an entry is added, deleted or modified.
- `rt_print_route_entry(...)`: Function to print an entry in the forwarding table in the `rt_print_table()` function.
- `rt_call_print_table()`: Function to initiate the printing of the forwarding table. This function sets up the necessary environment before calling `rt_print_table()`.
- `rt_print_table()`: This is a *call-back* function to print the forwarding table. You **must define and implement** this call-back function in your code.
- `rt_should_print_table`: This flag indicates whether it is time to print the forwarding table or not. Check this variable often, e.g., before calling `select()`. If this variable has a non-zero value, then invoke the `rt_call_print_table()` function.

DISCLAIMER: We reserve the right to change the support code as the project progresses to fix bugs and to introduce new features that will help you debug your code. You are responsible for reading the b-boards to stay up-to-date on these changes. We will assume that all students in the class will read and be aware of any information posted to b-boards. Please also refer to the FAQ document available at the course web page.

```
1: node id: 4
2: sequence: 7
3: 3 2 n1doc1
4: 3 2 n1doc2
5: 3 2 :128.2.1.56:15200
6: 3 1 :128.2.1.56:15202
7: 3 2 n3doc1
8: 4 0 :128.2.1.58:15204
9: 4 1 n4doc1
10: ...
```

Figure 6: Sample forwarding table output.

3.5 Forwarding Table Output Format

You are responsible for implementing the `rt_print_table()` function. Whenever this function is called, your routing process should print the whole forwarding table to the specified file descriptor.

Figure 6 shows how the output should look like. The first line contains the node id of the process printing the table. Line 2 contains a sequence number used to identify the forwarding table. You do not have to worry about printing these first two lines, they are automatically printed by the function that calls `rt_print_table()`. Lines 3 to 9 display entries in the routing table. Each line has the following format:

```
next_node_id   metric   destination
```

The first field is the node id of the next node to reach a destination. The second field is the number of hops required to reach that destination. The last field is the name of the destination. Notice the special *node destinations* in lines 5, 6 and 8.

In order to produce the output shown in Figure 6 your routing process simply has to iterate through the forwarding table and for each entry call the `rt_print_route_entry()` function with the appropriate parameters.

4 Grading criteria

When we grade your submission, we will run `gmake` to build your programs and then run a combination of tests and grading scripts to evaluate your submission. If your project generates compiler warnings during compilation, you will lose credit; if your server dumps core during our testing, you will lose substantial credit. Remember to compile with the `-Wall` flag, otherwise you will lose points if this is not in your `Makefile`.

Poor design, documentation, or code structure will probably reduce your grade by making it hard for you to produce a working program, and making it hard for us to understand it. Egregious failures in these areas will cause your grade to be lowered even if your implementation performs adequately. Putting all of your code in one module counts as an egregious design failure. It is better to have partial functionality working solidly than lots of code that doesn't actually do anything correctly.

4.1 Grade breakdown

The following is a checklist of all of the features your routing engine must possess. As you write your programs, you should cross off each feature from the list once you have implemented and tested it. All of these features will be tested during grading.

√	Description	(%)
	Initialize forwarding table (no route exchange)	5
	Exchange forwarding tables (no failures)	20
	Robustness: Handle communication failures	20
	Forwarding: Handle different packet types	20
	Forwarding: Error handling	15
	Part 2: Caching	15
	tests.txt file	5

4.2 What to turn in

Below is a listing of all of the files you should submit. The hand-in directory is `/afs/cs.cmu.edu/academic/class/15-441-f02-users/group-NN`. Use the lower group number between you and your teammate. Please note that you will not be able to write to the hand-in directory after the submission deadline. Check the late policy on the course web site and notify the TAs and professors if you will be using late days.

- GNU compatible `Makefile`. Make sure all the variables and paths are set correctly such that your program compiles in the hand-in directory. The `Makefile` should build an executable named `dnred`.
- All of your source code (files ending with `.c` or `.h` only, no `.o` files)
- `tests.txt`: File containing documentation of your test cases and any known issues you have. Below is an explanation of what you should include in this file.

4.3 The tests.txt file

Remember to submit the `tests.txt` file along with the rest of your assignment. The file should contain the following information:

1. **Documentation for Part 1**: Include a short paragraph describing the general design and implementation of your protocol and daemon for part 1.
2. **Documentation for Part 2**: Include a short paragraph describing the general design and implementation for the caching part.
3. **Testing strategies**: As part of this assignment, you are responsible for testing your submission and making sure everything works. You should document all of your testing strategies in this section.
4. **Outstanding issues**: Also, if you have any outstanding issues you know about, but are not able to fix, you should let us know about them.

5 From start to finish

5.1 Suggested checkpoints

Below is a table of *suggested* checkpoints. While it is not required that you meet all of the milestones by the specified dates, you should keep these deadlines in mind to gauge your progress. This assignment is due on **Monday, December 2nd, by 11:59 PM**. The late policy is explained on the course web site [1].

Date	Description
November 12	Assignment handed out. **PLEASE START EARLY!**
November 18	Distance Vector Routing protocol working.
November 21	Forwarding Module working.
November 30	Part 2: Caching working. Start rigorous testing.
December 2	Assignment due date. See section 4.2 for details on turning in your assignment. Remember that the assignment is due before 11:59 P.M.

5.2 Where to get help

Make sure you check the annotated reference at the end of this document, it contains sources of valuable information. If you have a question, please do not hesitate to ask us for help, that's why we're here! General questions should be posted to the class bulletin board [3], *academic.cs.15-441*. If you have more specific questions, especially ones that require us looking at your code, please drop by our office hours. The instructors' office hours are listed below:

Instructor	Office	Hours
Prof. Mor Harchol-Balter	Wean 8119	Tuesday 2:00 – 3:00 p.m.
Prof. Srini Seshan	Wean 8212	Thursday 3:00 – 4:00 p.m.
Julio López	Wean 8205	Monday 2:30 – 3:30 p.m.
Justin Weisz	Wean 3604	Tuesday 5:00 – 6:00 p.m.
Umair Shah	Wean 3710	Wednesday 2:00 – 3:00 p.m.
Xavier Appe	Wean 3108	Friday 2:00 – 3:00 p.m.

5.3 How to succeed in this assignment

You guessed, **start early**, the benefits of starting early should be clear by now. Read over this handout several times. If anything is unclear, please send email to the b-board `academic.cs.15-441`, or come to office hours. We are here to help! Once you understand the assignment, here is a *suggested* plan of attack. This outline is solely for guiding you through this assignment.

1. Parse the command line arguments and the configuration file.
2. Implement a simple forwarding table structure.
3. Populate the forwarding table with the routes to the immediate neighbors and documents in the node's directory.
4. Implement the routing protocol.
   - Use simple topologies with a few nodes.
   - Create a socket and contact neighbors and perform one iteration of the routing protocol.
   - Perform multiple loops of the routing protocol until routes converge.
   - Implement the full routing algorithm loop.
   - Ensure that your implementation handles complicated network configurations.
   - Add robustness. Handle node and link failures and count to infinity, make sure your implementation re-converges when links and nodes go down and come up.
5. Implement the forwarding engine. Create the socket for the forwarding module and add functionality for each packet type, one packet type at a time.
6. Implement part 2: Add caching to your routing engine.
7. If time permits optimize your forwarding table data structures.

5.4 Testing

Q: How should you run your programs? To facilitate the development, you can run all your routing engines in the same machine using different UDP ports and different document directories. In order to avoid "*port collisions*" among routing engines from different groups, use port numbers in the range: $[20000+\texttt{group number}\times 100, 20000+\texttt{group_number}\times 100+99]$. E.g., if your group number is 23, then you should choose the port numbers from the following range $[22300, 22399]$. If you are testing a configuration with 3 nodes, then the node addresses could be the following:

Node Id	IP address	Routing Port	Forwarding port
1	127.0.0.1	22300	22301
2	127.0.0.1	22302	22303
3	127.0.0.1	22304	22305

Q: How should you test your programs? The goal of this section is to give you a better idea of how your project will be evaluated, and what you should be on the lookout for while writing your code. The following list is not guaranteed to be exhaustive:

- Make sure your routing engine can handle various network topologies, check the correctness of the forwarding tables. Try simple topologies with few nodes, e.g., 3 - 8 nodes, no loops. Then try more complicated topologies with several nodes and connection patterns including loops.
- Test for node failures. What happens if a neighbor from the configuration file never comes up. What happens if a node goes down after being up for a while. What happens if a node comes back up after having crashed.
- Test for link failures. What happens when a link fails but the node is still alive?
- Test the forwarding engine: Make sure all packet types are handled correctly.
- Stress table lookup: Send multiple LOOKUP request packets to a node, measure how long it takes to process many of these lookups. Also, check how many packets are dropped, i.e., no reply is returned.
- Stress the forwarding system: send at "*full blast*" several QUERY and DATA_REQUEST packets to the system.
- Test error handling: Handle requests for non-existent documents.
- Handle link and node failure when forwarding packets. Are the correct error packets generated?
- For part 2: perform similar tests as for part 1 but with caching turned on. Is the appropriate document returned? Is it read from the cache or requested from the original node?

References

[1] 15-441: Computer networks web site. `http://www.cs.cmu.edu/~srini15-441/F02`.

This is the class's website, it contains information about the courses, including syllabus, lecture notes, project handouts, homework handouts, etc.

[2] BSD sockets programming. `http://world.std.com/~jimf/papers/sockets/sockets.html`.

Gives some code examples for creating sockets and writing to them.

[3] The class bulletin board. `news:academic.cs.15-441`.

We fully encourage you to read and post questions to the class bboard. The TAs will be reading the bboard daily, and this is the best place to get quick answers to short questions. If your question involves a lot of code, or if it will take more one minute to answer, please come to our office hours instead.

[4] GNU make manual. `http://www.gnu.org/manual/make/html_mono/make.html`.

Everything you need to know (and much much more) about `gmake` can be found here. The Makefile for this assignment should be very simple.

[5] Google groups. `http://groups.google.com`.

This is Google's archive of USENET postings. If you have a programming question (i.e. how do I do ____ in C?), or if you can't figure out what a certain function does, this is a great place to find answers.

[6] Lectures 3, 12 – 15. In *15-441 Computer Networks Lecture Notes*. `http://www.cs.cmu.edu/~srini/15-441/F02/lectures/`.

These are the notes from various lectures that contain information relevant to the project.
Lecture 3 provides a very easy introduction to socket programming. If you are new to socket programming, read these notes over before you write any actual code.
Lecture 12 explains IP addressing and forwarding.
Lecture 13 explains the basics of routing and RIP.
Lecture 14 explains the principles of BGP.
Lecture 15 explains Subnetting and CIDR.

[7] Sockets FAQ. `http://www.developerweb.net/sock-faq/`.

Huge FAQ providing many questions and answers to socket programming. Search here if you have specific socket programming problems.

[8] C. L. Hendrick. Routing information protocol. RFC 1058, `http://www.rfc-editor.org/rfc/rfc1058.txt`, June 1988.

This RFC describes version 1 of the RIP protocol.

[9] James F. Kurose and Keith W. Ross. *Computer Networking, A Top-Down Approach Featuring the Internet*. Addison Wesley, 2nd edition, 2002.

This is the class textbook. Chapter 3 explains the principles of a distance vector routing protocol. Also Chapter 2 offers an overview of peer-to-peer systems.

[10] G. Malkin. RIP version 2. RFC 2453, `http://www.rfc-editor.org/rfc/rfc2453.txt`, Nov 1998.

This RFC describes version 2 of the RIP protocol, which is backwards compatible with version 1.