ECE/CS 658 - Internet Engineering

Lab 5 - Distributed Hash Table Based Content Searching in a Distributed Application-Layer Network

1. Goals

At the start of the lab, please send an email to the TA notifying members of your group or if you are working individually. Each group/student will receive a dedicated BS Server in reply to this mail.

Develop a structured-overlay-based solution that allows a set of nodes to share contents (e.g., music files) among each other.

Consider a set of nodes each with a set of files that it is willing to share with other nodes. A node in the system (X) that is looking for a particular file issues a query to identify a node (Y) containing that particular file. Once the node is identified, the file can be exchanged between X and Y. Goal is to support information discovery using a structured overlay topology.

After completing this lab, you will have developed an efficient solution to search for contents in a large distributed system. You will be able to design and develop useful structured-overlay-based applications such as simple search engines to find contents in a large Peer-to-Peer network. You will further learn the importance of following a given protocol specification.

2. Assignment

You will develop a Distributed Hash Table (DHT) based content/resource discovery system for a structured P2P network (see uploaded **DHT Intro**). We will focus only on discovering static contents/resources such as files. You will implement only the resource discovery portion of a P2P protocol that will identify a set of nodes having a given file. No need to implement file transfer between peers. We will use Chord as the underlying DHT scheme. It is highly recommended that you read the Chord papers [1, 2] to gain better understanding about the DHT and the paper further provides relevant pseudo-codes. You are encouraged to reuse as much code as from Lab 4.

The first phase of the assignment is to generate the Chord overlay topology and the contents of each node. The network will consist of at least 20 and 40 PlanetLab nodes sharing 200 files among them, with each node contributing *n* files. Some files may be present in multiple nodes and some may not be in any node. A list of file names is provided, and each node is initialized with *n* (e.g.: *n* = 7 or n = 10) randomly selected files from this list. Then form the network and initialize the node contents as follows: A new node that comes up get connected to the Bootstrap Server (BS). For simplicity, the BS will give you a list of IP addresses of all the peers that are currently in the network. Based on that, you need to build and maintain the finger table and key table. Your peer should be able to add file names into the P2P network as well as search for existing files.

3. Steps

Here we explain how to join and maintain the connectivity in the P2P network. See Section 5 for specific message formats used to talk to peers and BS.

1. Generate the key for the peer by hashing its IP address
2. Bind and wait on a TCP port so that other peers can connect
3. Each new node/peer added to the system will register at the given Bootstrap Server (BS) by providing node's IP address and port number. BS will be maintained by the TA.
   - BS will respond only to the messages specified in Sec 5.1. Thus, it should be used only to find nodes currently in the system. It will not respond to any network formation or query messages.
   - BS will give you all the *IP address:port* information of existing peers in the network
   - Use TCP to connect to BS
4. Build the finger table
   - Your peer needs to inform other peers that need to modify their finger tables as a consequence of your entry/departure into/from the network
5. Get the relevant keys from your successor that you are responsible for
   - Build you key table based on the received keys
   - Keep the keys in the successor to enhance stability, if your peer fails
6. Insert entries related to new files into the network
   - Each new node will pick *n* files
   - file names randomly from the given list of **filenames.txt** (uploaded). Display the filenames selected upon command.(See section 4) .
   - Convert the filenames to lower case before hashing
   - If you filename is *Baby_go_home.mp3*, tokenize it and store a separate (*key*, *value*) for each keyword; *baby*, *go*, *home*, *baby_go_home*, and *baby_go_home.mp3*. The objective is that a user should be able to find any file with a given keyword, e.g., *home*.
   - Find peers that are responsible for each key that you generated
   - Store *keys* and associated *values* (IP address, port number, actual filename) on those peers
7. Search for a given file name or part of it
   - Search for each file in **queries.txt** (uploaded) by reading one line each.
   - Convert the query string to lower case and tokenize it before hashing
   - Hash the given string and generate the search key
   - Find the peer responsible for the key(s)
   - Retrieve the actual filename, IP, and port number of the peer that registered the file

    - Store the results in a text file.
8. Each peer should be able to display its current key table and finger table
9. Leave the network
    - You must tell the BS when you leave the network
    - You must also tell other peers to update their finger table that will be affected by departure of your peer
    - Give all your peer's keys to its successor
10. You are expected to demonstrate the operation of the system as described above. In addition, prepare a report with the following results:
    a. Pick 5 nodes randomly and issue queries, one after the other (no parallel queries). You should be able to search for both the entire file name and parts of it.
        - e.g.: If query ask for "Lord", "Lord rings", or "Lord of the rings" consider "lord of the rings" as a match if the node has a file with that name. Consider only complete words, e.g., if you search for "Lord", file with "Lo Game" is not a match similarly if you search "Lo", "Lord of the ring" is not a match.

    b. Find number of application-level (overlay) hops and latency required to resolve each query. After resolving all the queries, find number of query messages received, forwarded, and answered by all the 20 nodes. Also find their finger table and key table sizes.

    c. Repeat Steps 10.a & 10.b, 5 times and collect all the statistics. Every time pick a different node as the 1st node. Also try *n*=5 and *n*=10.

    d. Find min, max, average, and standard deviation of hops, latency, messages per node, and finger/key table size. Also find per query cost and per node cost. Plot distribution (CDF) of hops, latency, and messages per node. Repeat step a-d for P2P network of 20 and 40 Planetlab nodes. Comment on the effects of scaling network size on search performance (messages, hops, and latency).

    e. Prepare a report (must not exceed 8 pages) by including your findings from Step 10.d. Also compare and contrast this solution with the solution that you developed in Lab 4. Comment on how to improve the query resolution while reducing messages, hops, and latency.

4. Things to Note

1. You can work individually or as a group of two. Extra credit if done individually.
2. <u>Each of you will receive a dedicated BS. This information will be sent to you via Canvas email. You will receive a BS server running on CSU network and one running on PlanetLab. You should use the local BS for testing your code before running your code on PlanetLab with the PlanetLab BS.</u>
3. Each peer takes 3 arguments *self_port_no*, *BS_IP*, *BS_port*
    - Where *port_no* is the port number used by peer, *BS_IP* is the IP address of the BS, and *BS_port* is the port number used to connect to BS.
    - For example, *peer 12501 129.82.230.131 5000*
4. Use MD5 as the hash function. This will generate a string with 32 hexadecimal digits (128-bits).
    - For simplicity, use only the last 32-bit (last 8 hexadecimal digits) to represent a key.
    - For communicating with other peers and BS use the last 8 hexadecimal characters.
    - Convert the 8 hexadecimal character key string into a 32-bit integer for internal use. This will make it easy to figure out your successor and predecessor and range of keys.
5. Peer *key* is generated by hashing the IP address. This key will be used by the BS to uniquely identify your peer.
6. Assume no spaces in the filename. This is just to simply the protocol. You may use a unique symbol to indicate space (e.g., '_' or '#").
7. Use TCP to communicate among peers and to the BS. Always keep a port open so other peers can send their messages. Handle message by message. Allow a peer to connect, perform a task, and terminate the connection to that peer.
8. The following commands should be used to start/display specific functions as mentioned:
    a. **details** : Display the node's IP, Port and Key.
    b. **fingertable**: Display the node's finger table
    c. **keytable**: Display the node's key table
    d. **files**: Self files
    e. **search** : Start the search
    f. **exit** : Make the self node leave the network
    g. **exitall** : Make all the nodes in the network leave (optional to be implemented)
9. Use string tokenize to break a command and extract different elements of it.
10. You need to show all the messages being sent recieved and forwarded by the nodes and by who to whom. Use proper formatting.
11. You may use the pseudocodes in Chord paper. However, no need to implement stabilization features.
12. TA will maintain the BS.
13. If your program crashes, you will get a 9998 from BS when you try to register a node (REG) again (unless you use a different IP or port). To simplify and maintain a consistent view of the distributed system at the BS (if this happens), your node needs to unregister before attempting to register again. If you are going to rerun a node with a different IP, you have to issue an unregister request (UNREG) for the previous entry. You may do this through netcat (only for testing and debugging) by manually issuing the command. You may issue 'PRINT' command through netcat to see all the valid entries in the BS.
14. Actively participate to discussion on Canvas and talk to TA whenever you have concerns...
15. Start the lab early...

5. Protocol

We will use a character-based protocol to make it easy to debug. You may issues commands through netcat to the registry and other peers to check whether commands are correctly responded. Each message starts with a *command* (in uppercase characters) that can be up to *n* characters long. Rest of the message will depend on the command. Each element in the command is separated by a *white space*. Following figure indicates the commands and how they are related to peers.

Commands and their relatioship to peers.

Figure 1 - Commands and how they are related to peers (P2P Registry is another name for BS).

5.1 Register/Unregister With Bootstrap Server

***Register Request message* - used to register with the BS**

```
length REG IP_address port_no key
```

- *e.g., 0036 REG 129.82.123.45 5001 1234abcd*
- *length* - Length of the entire message including 4 characters used to indicate the length. Always give length in xxxx format to make it easy to determine the length of the message
- *REG* - Registration request
- *IP_address* - IP address in xxx.xxx.xxx.xxx format. This is the IP address other peers will use to reach you. Indicated with up to 15 characters
- *port_no* - Port number. This is the port number that other peers will connect to. Up to 5 characters
- *key* - Last 32-bits (last 8 characters) of the hash of the IP address

***Response message* - BS will send the following message**

```
length REGOK no_peers IP_1 port_1 key_1 IP_2 port_2 key_2 ... ...
```

- *e.g., 0069 REGOK 2 129.82.123.45 5001 1234abcd 64.12.123.190 34001 4567ef12*
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- *REGOK* - Registration response
- *no_ peers* - Number of peer entries that are going to be returned by the registry
  - If = 0, request is successful no peers in the system
  - If ≥ 1, request is successful, list of peers contacts will be returned
  - If = 9999, failed, there is some error in the command
  - If = 9998, failed, already in the register, unregister first
  - If = 9997, failed, can't register. BS full.
- *IP_1* - IP address of the 1st peer. Peers are in random order as they got registered
- *port_1* - Port number of the 1st peer
- *key_1* - Key of the 1st peer
- *IP_2* - IP address of the 2nd peer. Peers are in random order that they got registered
- *port_2* - Port number of the 2nd peer
- *key_2* - Key of the 2nd peer
- This list will continue depending on number of peers in the registry

***Unregister Request message* - used to unregister from the BS**

```
length UNREG key
```

- *e.g., 0019 UNREG 1234abcd*
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- *UNREG* - Unregister request
- *key* - Last 32-bits (last 8 characters) of the hash of the IP address. Your peer will be uniquely identified by the key

***Unregister Response message* - BS will send the following message**

```
length UNROK value
```

- *e.g., 0014 UNREGOK 0*

- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- *UNREGOK* - Unregister response
- *value* - Indicate success or failure
    - If 0 - successful, if 9999 - error while unregistering. Key may not be in the BS or command is incorrect

5.2 Update Finger Table

***Request message* - Used to request other relevant peers to update their finger table to reflect the entry/departure for your peer**

```
length UPFIN type IP port key
```

- *e.g., 0041 UPFIN 0 129.248.182.11 5001 1234abcd*
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- *UPFIN* - Request peer to update the finger table. A separate message has to be send to each peer that needs to update its finger table as a consequence of the entrance of the new peer
- *type* - Indicate whether the message is for a peer entry/departure into/from the network.
    - If 0 - this is an entry into the network
    - If 1 - this is a departure from the network
- *IP* - IP address of the peer that enter/leave the network
- *port* - Port number of the peer that enter/leave the network
- *key* - Key (hast of IP address) of the peer that enter/leave the network

***Response message* - Receiving peer will send the following message**

```
length UPFINOK value
```

- *e.g., 0014 UPFINOK 0*
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- *UPFINOK* - Finger table update response. Each peer that received the finger table update request has to respond to this
- *value* - 0 if successful, 9999 if command is incorrect, 9998 other erros

5.3 Get Keys from Successor

***Request message* - Used to request keys that the new peer is responsible for, from the successor**

```
length GETKY key
```

- *e.g., 0019 GETKY 1234abcd*
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- *GETKY* - Request keys that I am responsible for, from my successor
- *key* - my key so that successor can determine which subset keys to send to me

***Response message* - Successor will send the following message**

```
length GETKYOK no_keys IP1 port1 key1 filename1 IP2 port2 key2 filename2 ... ...
```

- e.g., *0122 GETKYOK 3 129.82.142.25 5001 5678bcde my_song3.mp3 64.21.135.147 10501 124578ac baby 98.25.145.111 2501 124578ac baby*
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- *GETKYOK* - key response
- *no_keys* - number of keys returned by successor.
    - If same key has multiple sources repeat *key*, *filename*, *IP*, *port* for each copy. Based on this indicate the number of responses.
    - Remember successor still keeps a copy of these keys in its key table
    - If ≥ 1 - successful
    - If = 0 - No keys to return
    - If = 9999 - Error, command may be incorrect
- *IP1* - IP address of the peer that registered the 1st key related to filename (or keyword)
- *port1* - Port number of the peer that registered the 1st key related to filename (or keyword)
- *key1* - key of the 1st filename
- *filename1* - Name of the file (can be a keyword extracted from filename)
- This list will continue depending on number of keys given by the in the successor

5.4 Give Keys to Successor

***Request message* - Used to send keys to the successor when a peer leave the network**

```
length GIVEKY no_keys IP1 port1 key1 filename1 IP2 port2 key2 filename2... ...
```

- *e.g., 0121 GIVEKY 3 129.82.142.25 5001 5678bcde my_song3.mp3 64.21.135.147 10501 124578ac baby 98.25.145.111 2501 124578ac baby*
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format

- *GIVEKY* - Give keys to successor
- *no_keys* - number of keys send to successor. If same key has multiple sources repeat *key*, *filename*, *IP*, *port* for each copy. Based on this indicate the number of responses
- *IP1* - IP address of the peer that registered the 1st key related to filename (or keyword)
- *port1* - Port number of the peer that registered the 1st key related to filename (or keyword)
- *key1* - key of the 1st filename
- *filename1* - Name of the file (can be a keyword extracted from filename)
- This list will continue depending on number of keys given by the in the successor

***Response message* - Successor will send the following message**

```
length GIVEKYOK value
```

- e.g., *0015 GIVEKYOK 0*
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- *GIVEKYOK* - Response message
- *value* - Indicate success or failure
    - If = 0, indicate success
    - If = 9999, Error in command. Ideally sending peer needs to retry with correct command
    - if = 9998, Some other error

5.5 Add a Key to the Network

***Request message* - Used to store a key in the network**

```
length ADD IP port key filename
```

- e.g., *0044 ADD 129.82.62.142 5070 ab3456cd my_song*
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- *ADD* - Add a key to the network
    - First, determine the closets peer to the key from the entries in the finger table, such that it is stored in its successor. Directly send command to the peer found from the finger table
    - If receiving peers is not the successor, it has to find an even closer peer (or successor) from the finger table and forward to it. Just forward the same command.
    - Repeat this process. Forwarding will stop when message is in the successor
    - It is possible to get the response to the command from a different node (successor) so you may terminate the connection to the node selected from the finger table
- *IP* - IP address of the peer that is adding the file to the network. Also useful in directly sending the response back
- *port* - port number of the peer that is adding the file to the network. Also useful in directly sending the response back
- *key* - key related to filename (or keyword)
- *filename* - Actual filename. Not the keyword that is bring stored

***Response message* - The node that stored the key in its key table will send the following message**

```
length ADDOK value
```

- e.g., *0012 ADDOK 0*
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- *ADDOK* - Confirms the addition as successful. The node that sends this message is the one that actually stored the (*key*, *value*) pair
- *value* - Indicate success or failure. 0 if successful, 9999 if failure due to peer unreachable, 9998 if some other error

5.6 Search for a Key

***Request message* - Used to locate a key in the network**

```
length SER IP port key
```

- e.g., Suppose we are storing keyword *baby*, *0036 SER 129.82.62.142 5070 ab3456cd*
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- *SER* - Locate the key in the network
    - First, determine the closets peer to the key from the entries in the finger table, such that it is located from its successor. Directly send command to the peer found from the finger table
    - If receiving peers does not have an entry related to the key, it has to find an even closer peer (or successor) from the finger table and forward to it. Just forward the same command.
    - Repeat this process. Forwarding will stop when key is found from a peer or reach the correct node that is responsible for the key
    - It is possible to get the response to the command from a different node (a node that have the key) so you may terminate the connection to the node selected from the finger table
- *IP* - IP address of the peer that is searching for the file. Useful in directly sending the response back
- *port* - port number of the peer that is searching for the file. Useful in directly sending the response back
- *key* - key related to filename (or keyword) being searched

***Response message* - The node that has or responsible for the key will send the following message**

```
length SEROK no_keys IP1 port1 filename1 IP2 port2 filename2 ... ...
```

- e.g., Suppose we are searching for string *baby*. So it will return, *0114 SEROK 3 129.82.128.1 2301 baby_go_home.mp3 171.182.128.21 6301 baby_come_back.mp3 129.82.128.1 2301 baby.mpeg*
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- *ADDOK* - Sends the result for search. The node that sends this message is the one that actually stored the (*key*, *value*) pair
- *no_keys* - Number of results returned
  - If ≥ 1, Successful
  - If = 0 no matching results. Searched key is not in key table
  - If = 9999, failure due to peer unreachable
  - If = 9998, some other error
- *IP1* - IP address of the node having (stored) the file
- *port1* - Port number of the node having (stored) the file
- *filename1* - Actual name of the file

5.7 Error Message

```
length ERROR
```

- *0010 ERROR*
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- *ERROR* - Generic error message, to indicate that a given command is not understood. For storing and searching keys this should be send to the initialter of the message

5.8 Print Peer List

```
length PRINT
```

- *0010 PRINT*
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- *PRINT* - Request the BS to send the list of peer entries in the registry. This is only for debug purposes so issue command through netcat. Your peers are not expected to issue this command

5.9 Clear BS NodeList

```
length CLEAR
```

- *0010* CLEAR
- *length* - Length of the entire message including 4 characters used to indicate the length. In xxxx format
- CLEAR - Request the BS to clear the list of peer entries in the registry. This is only for debug purposes so issue command through netcat. If your program crashes you may use this to clear the BS. Do not include this command in your program.

6. Testing with Bootstrap

- Use the command "nc *Bootstrap_IP Bootstrap_port*" to netcat to the BS Server.
- You will receive a local and PlanetLab BS. Please use the local BS to debug your code and use the PlanetLab BS for final testing.
- Use the commands mentioned in Section 5 to communicate with the BS.
- Any issues with the BS must be notified to the TA with the subject "BootStrap Issues".

7. Demo

- You need to run at least 20 peers in PlanetLab
- Need to show that your peers get registered with the BS
- Need to show that your peers can store (*key*, *value*) pairs in the network
- Display details as per Sec 4.8
- **You should download your submitted file on Canvas and transfer the files to the PlanetLab nodes at the time of demo. Make use of pscp or bash Scripting.**
- You need to make few nodes leave and show the exchange of keys and values accordingly.

8. What to Submit

- Report analyzing performance (must not exceed 8 pages).
  Name your report **Lastname_Firstname_Lab5.pdf**
- You need to submit your code and the makefile (if any). However, do not forget to submit readme.txt file with instructions for compilation and how to run your peer.
- Please do NOT submit executables. Submit all files as a single compressed file (.zip or .tar.gz is preferred).
  Your uncompressed and compressed folder should be named **Lastname_Firstname_Lab5**
- Upload your submission on Canvas by the due date and time.
- Please register your group if working in a group on Canvas.
- **Your most recent submission on Canvas will be considered as your final submission.** Whatever is the time and date of

your most recent submission, will be considered. No exceptions. You will have to download the code of your final submission from Canvas for demo.

9. Grading

- Demo - 60% . You will also face a couple of questions during the demo that should be answered for full score.
  - 45% - Correct working of lab. All files are search correctly. Correct implementation of Chord Structure. Few Questions.
  - 10% - Display of all required details asked to be displayed on the terminal
  - 5% - Clarity and neatness in display of results. Give appropriate spaces, tabs and new lines.
- Report - 30%
  - 25% - Content
  - 5% - Presentation
- Coding style - 10%
- Extra credits - 10%.

10. Late Penalty

- There will a 5% late penalty per day.

11. Deadline

- Code Deadline : 5th April
- Demo Deadline : 10th April

Prepared by Dilum Bandara, Vidarshana Bandara and Anura P. Jayasumana
Last updated 03/17/2015 by Divyanka Bose