CIS 553 / TCOM 512 Networked Systems

Project 2: PennSearch – A DHT-based Search Engine

1. Introduction

In this project, you will implement a peer-to-peer search engine (PennSearch) that runs over your implementation of the Chord Distributed Hash Table (PennChord). You are expected to read the entire Chord [1] paper carefully, and clarify any doubts with your TAs, instructor, or post questions on the newsgroup.

You will start from your routing protocol implementation in project 1. You can choose either distance vector and link state, or if your project did not work properly, configure to use OLSR as your routing protocol. The use of OLSR can be turned on using a command line flag (*--routing=ns3*) of simulator-main.cc. You are then responsible for first developing Chord as an overlay network layered on top of your routing protocol, followed by building the search engine application that uses Chord.

To help you get started, we have provided the new files:

a. **simulator-main.cc:** This contains an improved simulator-main that has additional SEARCH_LOG and CHORD_LOG functions to log all messages relevant to search engine and Chord respectively. We also add modules for CHORD and PENNSEARCH.
b. **scenarios/pennsearch.sce:** An example scenario file that contains a simple 3 node Chord network, and publishing of document keywords by two nodes, and three example queries.
c. **keys/metadata0.keys, keys/metadata1.keys:** Each file contains the meta-data for a set of documents, where each row "Dn T1 T2 T3…" is a set of keywords (in this case, T1, T2, and T3) that can be used to search for a document with identifier Dn. Each document identifier can be a web URL or a library catalog number, and for the purpose of this project, they are simply a string of bytes that uniquely represent each document. In practice, these keywords are generated by reading/parsing web content or other documents to extract keywords. However, since parsing web pages is not the focus of this project, we have skipped this step, and supply these document keywords to you. You can do the actual extraction as extra credits.
d. **penn-chord-*.[h/cc]:** A barebones Chord skeleton code.
e. **penn-search-*.[h/cc]:** A barebones PennSearch skeleton code

Depending on your preference, you may load the above files into your current repository, or start a new one from scratch, and move your LS and DV implementations from project 1 over. Note that our skeleton code is a starting point, and you are allowed to be creative and structure your code based on your own design. For regular credits, you are however not allowed to make any code changes outside of upenn-cis553 directory, nor are you allowed to change simulator-main.cc. In addition to our sample test script, we expect you to design additional test cases that demonstrate the correctness of your PennSearch implementation.. While the project is due Nov 30, we encourage you to get started early. As a result, we have included a milestone 1 where you should have a basic Chord implementation.

2. PennChord

Chord nodes will execute as an overlay network on top of your existing routing protocol. I.e. all messages between any two Chord nodes 1 and 2 will traverse the shortest path computed by the underlying routing protocol. The PennChord overlay network should be started only after the routing protocol has converged (i.e. finish computing all the routing tables). You can assume that the links used in the underlying routing protocol computation does not changed while PennChord is executed. Also, not all nodes need to participate in the Chord overlay, i.e. the number of nodes in your underlying network topology may be larger than the number of nodes in the Chord overlay. Your PennChord implementation should include finger tables and the use of 160-bit cryptographic hashes.

2.1 Correct Chord Behavior

We will also be using our own test cases to ensure correct Chord behavior. For simplicity, assume each node only maintains one successor (i.e. the closest node in the clockwise direction). You have to implement the **actual stabilization functionality** described in the Chord paper. In particular, we will check for the following features:

- **Correct lookup.** All lookups routed via Chord has to be proven correct (i.e. reach the correct node via the right intermediate nodes. You need to support the basic Chord API, which is **IPAddr ← lookup(k)**, as described in lecture. This API is not exposed to the scenario file explicitly, but used by PennSearch to locate nodes responsible for storing a given keyword.
- **Consistent routing.** All nodes agree on lookup(k).
- **Well-formed ring.** For each node n, it's successor's predecessor is itself. One easy way is to make use your *ring state* command described below to ensure that the Chord ring is formed correctly.
- **Correct storage.** Every item K is stored at the correct node (i.e. lookup(k))
- **Performance:** For a given network size, you need to compute and output the average hop count required by Chord lookups that occur during the duration of your simulation. In other words, you implement the code that will capture the number of hops required by each lookup and output using CHORD_LOG the average hop count across all lookups when the simulation ends. The average hop count must exclude lookups that are generated during periodic finger fixing.
- **Stabilization protocol.** Enough debugging messages (not too many!) to show us that periodic stabilization is happening correctly. Since stabilization generates large numbers of messages, you should provide mechanisms to turn on/off such debugging in your code.

2.2 Summary of Commands

- **Start landmark node**: Designate a node as your landmark (i.e. node 0). E.g. "0 PENNSEARCH CHORD join 0" will designate node 0 as the landmark node, since the source and landmark nodes are the same.
- **Nodes join**: A PennChord node joins the overlay via the initial landmark node. E.g. "1 PENNSEARCH CHORD join 0" will allow node 1 to join via the landmark node 0. Once a node has joined the network, items stored at the successor must be redistributed to the new node according to the Chord protocol. For simplicity, you can assume all joins and leaves are sequential, i.e. space all your join events far apart such that the successors and predecessors are updated before the next join occurs.
- **Voluntary node departure**: A PennChord node leave the Chord network by informing its successor and predecessor of its departure. E.g. "1 PENNSEARCH CHORD leave" that will result in node 1 leaving the PennChord network. All data items stored should be redistributed to neighboring nodes accordingly. For simplicity, you can assume all joins and leaves are sequential.
- **Ring state debug**: At any node X, a "X PENNSEARCH CHORD ringstate" command will initiate a *ring output message* that initiates from node X, and traverse the entire Chord ring in a clockwise direction to output all successors and predecessors. This message will terminate at the initiating node X. Each node *curretNodeAddr* that receives the message will generate the following output in a single line using CHORD_LOG:
    - *RingState<currentNodeKey>: Pred<predNodeNum, predKey>, Succ<succNodeNum, succKey>*

2.3 Debug logs for grading purposes

For grading purposes, in addition to ring state output above, we require the following information to be printed using CHORD_LOG. All Chord identifiers should be printed in hexadecimal.

Lookup issue debug: Every time a node issues a lookup request, the following message is printed:

- *LookupIssue< currentNodeKey,targetKey>*

Lookup forwarding debug: Every time a node forwards a lookup request, the following message is printed:

- *LookupRequest<currentNodeKey>: NextHop<nextAddr, ,nextKey, targetKey>.*

Lookup results debug: Every time a node returns a result in response to a lookup request back to the node that originated the initial lookup request, the following message is printed:

- *LookupResult< currentNodeKey, targetKey, originatorNode>*

Note that there are no specific commands that will generate lookups. They are generated either by Chord's periodic finger fixing, or PennSearch's invocation of PUBLISH and SEARCH (see below). For our testing purposes, you must leave out lookups generated by finger fixing in the CHORD_LOG output, but only print out lookups that are initiated by PennSearch. You can differentiate lookups within Chord, by setting a Boolean flag to true for each lookup generated via finger fixing, and false otherwise.

3. PennSearch

To test your Chord implementation, we will require you to write PennSearch, which is a simple keyword based search engine.

3.1 Basics of Information Retrieval

We first provide some basic knowledge that you would need to understand keyword-based information retrieval. We consider the following three sets of document keywords, one for each of Doc1, Doc2, and Doc3:

Doc1 T1 T2
Doc2 T1 T2 T3 T4
Doc3 T3 T4 T5

Doc1 is searchable by keywords T1 or T2. Doc2 is searchable by T1, T2, T3, and T4, and Doc3 is searchable by T3, T4, and T5. Typically, these searchable keywords are extracted from the actual documents with identifiers Doc1, Doc2, and Doc3.

Based on these keywords, the *inverted lists* are {Doc1, Doc2} for T1, {Doc1, Doc2} for T2, {Doc2, Doc3} for T3, {Doc2, Doc3} for T4, and {Doc3} for T5. Each *inverted list* for a given keyword essentially stores the set of documents that can be searched using the keyword. In a DHT-based search engine, for each inverted list Tn, we store each list at the node whose Chord node is responsible for the key range that includes hash(Tn).

A query for keywords "T1 AND T2" will return the document identifiers "Doc1" and "Doc 2", and the results are obtained by intersecting the sets {Doc1, Doc2}, and {Doc2, Doc3}, which are the inverted lists of T1 and T2 respectively. We only deal with AND queries in PennSearch, so you can ignore queries such as "T1 OR T2".

Note that the query result is not the actual content of the documents, but rather the document identifiers that represent documents that include both T1 and T2. In typical search engine, an extra document retrieval phase occurs at this point to fetch the actual documents. We consider the actual document content retrieval step out of scope of this project, although it is included as one of the extra credits.

3.2 Summary of Commands

- **Inverted list publishing.** "2 PENNSEARCH PUBLISH metadata0.keys" means that node 2 reads the document metadata file named *metadata0.keys.* Node 2 then reads each line, which is of the form "Doc0 T1 T2 T3 …", which means that the Doc0 is searchable by T1, T2, or T3.
  After reading the metadata0.keys file, node 2 constructs an inverted list for each keyword it encounters, and then publishes the respective inverted indices for each keyword into the PennChord overlay. For instance, if the inverted list for "T2" is "Doc1, Doc 2", the command publishes the inverted list "Doc1, Doc2" to the node that T2 is hashed to. This node can be determined via a Chord lookup on hash(T2). As a simplification, the inverted lists are append-only, i.e. new DocIDs are added into a set of existing document identifiers for a given keyword, but never deleted from an inverted list.

- **Search query.** "1 PENNSEARCH SEARCH 4 T1 T2" will initiate the search query from node 1, and take the following steps via node 4:
  a) Retrieve the inverted list of T1 from the node that is T1 is hashed to;
  b) Send the list retrieved in step a) to the node that T2 is hashed to;
  c) Send the intersection of the inverted lists of T1 and T2 as the final results back either directly back to node 1, or to node 4 (which forwards the results to node 1). If there are no matching documents, a "no result" is returned to node 1.
  For simplicity, you can assume there is no search optimization, so inverted lists are intersected based on left-to-right ordering of search terms. Note that the search may contain arbitrary number of search terms, e.g. "1 PENNSEARCH SEARCH 4 T1 T2 T3".

NOTES:

- You can assume that each document identifier appears in only one metadataX.keys file. For instance, if node 0 publishes metadata0.keys, and node 1 publishes metadata1.keys, you can assume that both files do not contain overlapping document identifiers. This emulates the fact that in practice, each node will publish inverted indexes for documents that it owns, and one can make the assumption that each node owns a unique set of documents.
- On the other hand, each searchable keyword may return multiple document identifiers. For instance, there are two documents Doc2 and Doc3 that can be searched using the keyword T3.
- You can assume that only nodes that participate in the PennSearch overlay can have permission to read document keywords and publish inverted indexes. For instance, in our penn-search.sce, only nodes 0, 1, and 2 are allowed to publish inverted indexes, although there may be more nodes in the underlying topology.
- Nodes, which are outside the Chord network, may initiate SEARCH by contacting any node, which is part of the PennSearch overlay. For instance, in our example command above, "1 PENNSEARCH SEARCH 4 T1 T2" means that node 1 (which may be a node outside the PennSearch overlay) can issue a search query for "T1 and T2" via node 4, which is already part of the PennSearch overlay.

3.3 Debug logs for grading purposes

For grading purposes, we require the following information to be printed using SEARCH_LOG:

Inverted list publish debug: Whenever a node publishes a new inverted list entry, the following debug message is generated:

- *Publish< keyword, docID>*

Inverted list storage debug: Whenever a node (that the keyword is hashed to) receives a new inverted list entry to be stored, the following debug message is generated:

- *Store< keyword, docID>*

Note that if CHORD_LOG is turned on, this means that between each Publish and Store output message, we should see a series of lookup output messages

Search debug: Whenever a node issues a search query with terms T1, T2,…,Tn, output

- *Search< T1,T2,…Tn>*

Inverted list shipping debug: For each inverted list <docIDList> being shipped in the process of the search, output:

- *InvertedListShip< targetKeyword, docIDList>*

Search results debug: At the end of intersecting all keywords (T1, T2,…,Tn), output the final document list <docIDList> that is being sent back to the initial query node:

- *SearchResults< queryNode, docIDList>*

Empty search results debug: In the process of the search, if any intermediate node results in an empty inverted list to be sent back to the initial query node, output:

- *SearchResults< queryNode, "Empty List">*

4. Extra credits:

We have suggested several avenues for extra credit, to enable students to experiment with the challenges faced in designing the Chord protocol to work 24x7 over the wide-area. Note that doing extra credit is entirely optional. We offer extra-credit problems as a way of providing challenges for those students with both the time and interest to pursue certain issues in more depth.

For each extra credit X (0,1,2,…), rename your upenn-cis553 folder as proj2ecX_login, where login is your eniac login name. Next, use the command **"turnin –c cis553 –p proj2ecX <proj2ecX_login>"** on eniac to submit your code. Note that you have to submit the entire upenn-cis553 directory, including documentation and scenario files.

For easy grading and submission, feel free to combine your extra credit source code, test cases, and design writeup with the regular credit. **You should only do this if you are very confident that your extra credit implementation does not break the regular credit implementations.**

Some suggested extra credits:

a) **Bandwidth efficient search engine (10%):** The basic PennSearch can be enhanced to save bandwidth as follows: perform the intersection of keywords starting from the one with smallest inverted list, and use of bloom filters. Implement these features and show that your implementation results in lower bandwidth utilization compared to the regular implementation.

b) **Enhanced search features (10%, 5%, 5% respectively)**: Enhanced your search engine with at least the following features (each of which is worth 5%):
   - Generating the document keywords (e.g. metadata0.keys, metadata1.keys) with actual web pages (>20) that you have downloaded, and replacement docID with actual URLs. After returning the search results, fetch the actual documents themselves. **Do the last step sparingly so as not to overload existing web servers with your http requests.**
   - Rank search results based on a reasonable search metric, such as TF-IDF.

  - Support inverted lists that are larger than MTU size of 1500 bytes. For instance, you can use a fragmentation/defragmentation scheme that breaks up an inverted list into smaller size lists in executing the search query.

c) **Chord file system (20%).** The Chord File System (CFS) [3] is a fairly sophisticated application that uses Chord. Hence, this has more extra credit than other applications. You need to demonstrate to us that you can take a few fairly large files, store it in CFS, and retrieve them correctly. To earn the entire 20%, you need to implement all the functionalities described in the CFS paper.

d) **Internet Indirection Infrastructure (15%).** Support three i3-style communication primitives (multicast, anycast, and service composition) as described in lecture notes. To get full credit, you need to implement all the features as described in the i3 paper.

e) **Chord performance enhancements (10%):** Chord has O(log N) hop performance in the virtual ring. However, the adjacent nodes in the ring may actually be far away in terms of network distance. How to take advantage of network distance information (gathered via the link/path costs from ns-3 or your project 1 routing implementation) to select better Chord neighbors? Read this paper [2] for ideas.

f) **Churn handling and failure detection (20%):** Add support to emulate failures of PennChord nodes, mechanisms to detect failures of PennChord nodes, and repair the routing state in PennChord accordingly when failures occur. To ensure robustness, each node needs to maintain multiple successors. All churn events (node joins, leaves, or failures) can happen concurrently, and a node that fails can rejoins the overlay later. You can however not concern yourself with failures at the network layer, i.e .all failures occur at the application-layer node.

g) **PennSearch on speclabs (10%):** In addition to running in simulation mode, turn on the implementation mode of ns-3 (using the TA's custom transport layer implementation), and demonstrate a working search engine on 6 Speclabs nodes. You need to map from our node identifiers to actual IP addresses of the Speclabs machines.

h) **Application-layer multicast (15%):** Using Chord as a building block, build and maintain a single source multicast tree as described in the lecture, where JOIN requests are routed to the root note (for each multicast group), and forwarding is set up on the reverse path. To get full credit, you need to construct the tree, show actual content being disseminate via the tree, and make sure you handle the case where nodes enter/leave the ring. You can assume graceful exits and not sudden failures of nodes.

[1] http://pdos.csail.mit.edu/chord/papers/paper-ton.pdf. Chord Journal paper.
[2] http://pdos.csail.mit.edu/papers/dhash:nsdi/paper.pdf
[3] http://pdos.csail.mit.edu/papers/cfs:sosp01/cfs_sosp.pdf