CIS 5530: Project 2
A DHT-based Search Engine

Spring 2025

Milestone 1 due Mar. 31
Full solution due Apr. 16

Directions

You must work in groups of two for this project. Please regularly check Ed throughout this course for clarifications on project specifications. You are required to version control your code, but please only use the GitHub repository created for you by the 5530 staff. You should have a different repo for project 2 vs the one used in project 1 even if your team members stayed the same. Do not work in public GitHub repositories! Please avoid publishing this project at any time, even post-submission, to observe course integrity policies. If you are caught using code from other groups or any sources from public code repositories, your entire group will receive ZERO for this assignment, and will be sent to the Office of Student Conduct where there will be additional sanctions imposed by the university.

1 Overview

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 paper carefully, clarify doubts with your TAs or instructor, or post questions on Ed.

You will start using standard OLSR as your routing protocol. The use of OLSR can be turned on using a command line flag `--routing=NS3` to `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. As Extra Credit, you can use your routing protocols from Project 1 and run Project 2 over the Project 1 network layer (using `--routing=LS/DV`).

To help you get started, files related to Project 2 include:

- `simulator-main.cc:` In addition to what you learned in project 1, it has `SEARCH_LOG()` and `CHORD_LOG()` functions to log all messages relevant to the search engine and Chord overlay, respectively. It also includes modules for CHORD and PENNSEARCH.
- `scenarios/pennsearch.sce:` An example scenario file that contains a simple 3-node Chord network, the publishing of document keywords by two nodes, and three example queries.
- `keys/metadata0.keys, keys/metadata1.keys:` These keys are located inside the `contrib/-upenn-cis553/` directory and not in the scenario directory. Each file contains the meta-data for a set of documents, where each row "*DOC 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 *DOC*. 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 supplied these document keywords to you.

- `penn-chord-*.[h/cc]`: Skeleton code for your Chord implementation.
- `penn-search-*.[h/cc]`: Skeleton code for your PennSearch implementation.
- `grader-logs.[h/cc]`: These files contain the functions you need to invoke for the autograder to correctly parse your submission. Please read the documentation in the header file.

The command to compile and run Project 2 is the same as Section 2.3 and 2.4 in Project 1 code documentation. Please do not use `--result-check` flag in the command, and please comment out calls to `checkNeigbhorTableEntry()` and `checkRoutingTableEntry()` if you use your LS and DV implementation. You can also opt to use ns-3's OLSR implementation instead of your own LS and DV.

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 credit, 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. 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 do not change 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 32-bit cryptographic hashes.

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 must reach the correct node via the right intermediate nodes. You need to support the basic Chord API, IPAddr ← lookup(k), as described in lecture. This API is not exposed to the scenario file explicitly but is 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*, its successor's predecessor is itself. See Ring State described in the autograder section.
- **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 should 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.

After finger fixing is correctly implemented, the average hop count should be $log(N)$ instead of N, where N is the number of nodes in the Chord overlay network.

- **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.

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 leaves 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.

Cryptographic Hashes

We provide a helper class to generate 32-bit cryptographic hashes from IP addresses and strings. It depends on the OpenSSL libcrypto library. To generate a digest for a message, use:

```
createShaKey(ipAddress)
// or
createShaKey(string)
```

3 PennSearch

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

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 T2 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, Doc3} 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 {Doc1, 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 a 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 the scope of this project.

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 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 to 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) Node 1 contacts node 4 with query "T1 AND T2";

  (b) Node 4 issues a Chord lookup to find the node that stores the inverted list of T1, i.e., the node that T1 is hashed to (e.g., Node_T1), and sends the query "T1 AND T2" to Node_T1;

  (c) Node_T1 retrieves an inverted list for T1 from its local store, issues a Chord lookup to find the node that T2 is hashed to (e.g., Node_T2), and sends the retrieved inverted list for T1 together with the query "T2" to Node_T2;

  (d) Node_T2 sends 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 an 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. 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.

4 Milestones

- Milestone 1: (15%) (Autograded)

  We expect a basic Chord implementation where the ring stabilization protocol works correctly. At this stage, finger table implementation is optional for this milestone. We require only the ringstate output to make sure your ring is formed correctly and maintained as nodes enter/leave the Chord overlay.

- Milestone 2a (39%), Milestone 2b (46%) (Both autograded)

  Complete working implementation of PennChord and PennSearch. Milestone is split into two parts for easier submission.

  Criteria for Milestone 2A:

  - Node to join the ring (3 points)
    - event : node 0 - node 19 join the ring
    - result : should see lookupissue, lookuprequest, lookupresult log
  - Well formed ring (3 points)
    - event : `3 PENNSEARCH CHORD RINGSTATE`
    - result : check well-formed ring, can also check above request path
  - P keyword metadata (8 points, 2 points each)
    - events :
      ```
      2 PENNSEARCH PUBLISH metadata2.keys
      3 PENNSEARCH PUBLISH metadata3.keys
      4 PENNSEARCH PUBLISH metadata4.keys
      5 PENNSEARCH PUBLISH metadata5.keys
      ```
    - result : should see publish and store log
  - Search correctness (9 points, 3 points each)
    - event : `1 PENNSEARCH SEARCH 4 Johnny-Depp`
    - result : Pirates-of-the-Caribbean Alice-in-Wonderland
    - event : `3 PENNSEARCH SEARCH 10 Johnny-Depp Keira-Knightley`
    - result : Pirates-of-the-Caribbean
    - event : `8 PENNSEARCH SEARCH 17 George-Clooney Brad-Pitt Matt-Damon`
    - result : Ocean's-Eleven Ocean's-Twelve
  - Search consistency (4 points)
    - events :
      ```
      2 PENNSEARCH SEARCH 12 Brad-Pitt
      3 PENNSEARCH SEARCH 13 Brad-Pitt
      4 PENNSEARCH SEARCH 14 Brad-Pitt
      ```
    - result : should all return Mr-Mrs-Smith Ocean's-Eleven Ocean's-Twelve Fight-Club

- Multiple searches from one node at same time (9 points, 3 points each)
  - event : `15 PENNSEARCH SEARCH 15 Tom-Hardy`
  - result : The-Dark-Knight-Rises Mad-Max
  - event : `15 PENNSEARCH SEARCH 3 Emilia-Clarke`
  - result : Game-of-Thrones
  - event : `15 PENNSEARCH SEARCH 12 Chadwick-Boseemann`
  - result : No such file
- Non-chord node issue search (3 points)
  - event : `25 PENNSEARCH SEARCH 9 Tom-Hanks`
  - result : Forrest-Gump Toy-Story
  - event : `21 PENNSEARCH SEARCH 16 Jeremy-Renner`
  - result : Arrival

Criteria for Milestone 2B: Milestone 2B will test correctness for more advanced cases. It is currently designed as an unseen test — but the autograder will give information for any wrong implementations and hints on how to fix them.

Important: in this milestone (2b), you will need to change your Ring State implementation to indicate that all ring state messages for the current ring have been printed to stdout, by printing a *End of Ring State* message after the last node in the ring printed its ring state message.

Autograder

The autograder parses specific logs from your submission. Please make sure you have all of your own printouts commented out before making your submission. Having your own printouts interleaved with the logs autograder expects may cause it to crash. You will use your GitHub repo and the Gradescope infrastructure for your submissions.

In the following logs, " `(...)` " means to use the corresponding arguments for the function. Please take a moment to read the comments in `grader-logs.h`.

- **Ring state (MS1):** At any node X, a `X PENNSEARCH CHORD RINGSTATE` command will initiate a ring output message. The node receiving this command should call the following function to log its ringstate, after that it should pass a message to its successor, which should also call the following function, and so on until X is reached again.

```
GraderLogs::RingState(...)
```

You should see something like this for every node in the ring. Note that if your implementation is correct, IDs, IPs, and Hashes should all form a doubly linked list.

```
Ring State
   Curr<Node currNode#, currIPAddress, currHash>
   Pred<Node predNode#, predIPAddress, predHash>
   Succ<Node succNode#, succIPAddress, succHash>
```

The last node in ringstate (X's predecessor) should call the following function after invoking RingState:

```
GraderLogs::EndOfRingState()
```

```
CHORD_LOG(GraderLogs::GetLookupIssueLogStr(...))
```

- **Lookup issue:** Every time a node issues a lookup request, the following should be called:
- **Lookup forwarding:** Every time a node forwards a lookup request, the following should be called:

```
CHORD_LOG(GraderLogs::GetLookupForwardingLogStr(...))
```

- **Lookup results:** 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 should be called:

```
CHORD_LOG(GraderLogs::GetLookupResultLogStr(...))
```

- **Inverted list publish:** Whenever a node publishes a new inverted list entry (an entry is a key, value pair), the following should be called:

```
SEARCH_LOG(GraderLogs::GetPublishLogStr(...))
```

- **Inverted list storage:** Whenever a node (that the keyword is hashed to) receives a new inverted list entry to be stored, the following should be called:

```
SEARCH_LOG(GraderLogs::GetStoreLogStr(...))
```

  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. Also, note that when a node leaves the ring, this should trigger its keys to be transferred to another node. This transfer should result in new store messages.

- **Search:** Whenever a node issues a search query with terms T1, T2,...,Tn, the following should be called:

```
SEARCH_LOG(GraderLogs::GetSearchLogStr(...))
```

- **Inverted list shipping:** For each inverted list <docIDList> being shipped in the process of the search, the following should be called:

```
SEARCH_LOG(GraderLogs::GetInvertedListShipLogStr(...))
```

- **Search results:** 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:

```
SEARCH_LOG(GraderLogs::GetSearchResultsLogStr(...))
```

- **Average Hop Count:** The following function should be called on the destructor of each node chord layer. Note that only hops related to searches are counted here (i.e., search layer, not chord layer). For MS1, you should see $O(N)$, and for MS2, you should see $O(log(N))$ hops. Example: If node 8 gets a lookup related to search, then it forwards to node 10 (hop 1), node 10 forwards to node 15 (hop 2), and node 15 sends the response to node 8 (no hop here, it is a response, not a lookup anymore!).

```
GraderLogs::AverageHopCount(...)
```

Hints

- **Own Address:** Use the following API to obtain current node IP in the application layer:

```
Ipv4Address GetLocalAddress();
```

- **Node Hash key:** Use the following API to obtain node hash:

```
uint32_t CreateShaKey(const Ipv4Address &ip)
```

- **Callbacks:** For MS2, make sure you understand callbacks implementation and check how pings are implemented in the Chord layer. Note that these are different pings from project 1, which used network-layer pings. These are overlay network pings.

  This is the callback setter in chord:

```
void PennChord::SetPingSuccessCallback(
      Callback<void, Ipv4Address, std::string> pingSuccessFn) {
   m_pingSuccessFn = pingSuccessFn;
}
```

  This is Penn-search using the setter to register a callback (i.e. ”Hey Chord when you have a PingSuccess please execute my function HandleChordPingSuccess”)

```
// Configure Callbacks with Chord
m_chord->SetPingSuccessCallback(
      MakeCallback(&PennSearch::HandleChordPingSuccess, this));
```

  This is the actual function in Penn-Search:

```
void PennSearch::HandleChordPingSuccess(
       Ipv4Address destAddress, std::string message) {
   SEARCH_LOG("Chord Ping Success! ....");
   // Send ping via search layer
   SendPennSearchPing(destAddress, message);
}
```

  This is Chord executing the callback:

```
m_pingSuccessFn(sourceAddress, message.GetPingRsp().pingMessage);
```

Extra Credit

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.

Similar to Project 1, extra credit will be submitted separately from regular credit on Gradescope. You can demo your extra credit to your assigned TA after the deadline. Make sure to document your work, design choices, and test cases.

- **Demonstrate using your LS/DV from project 1 (5%):** If you use your own LS or DV instead of the default ns-3 routing for your demo, you will get 5% extra credit. This may require some modifications to your project 1 code to get this to work properly, in particular, implement routeInput and routeOutput functionality, which is not tested by our autograder.
- **Chord file system (20%).** The Chord File System (CFS) 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. You can search online for the MIT CFS paper.
- **Chord performance enhancements (5%):** 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 can one 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? Do some research online to find techniques explored by others to solve this problem and implement one such solution.
- **Churn handling and failure detection (10%):** 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.
- **PennSearch on mobile devices (20%).** Using your Android phones, demonstrate a small (2-3) PennSearch network running on mobile devices. This requires taking the ns-3 code and compiling it on Android. If you do not have an Android, you can use an Android emulator.