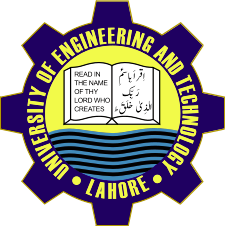
Operating Systems EE431L

Complex Engineering Problem - Report



*Parallelized implementation of Gauss-Seidel method using OpenMP and POSIX Threads*

*Submitted by:*

2016-EE-189 Muhammad Kamil

*Submitted to:*

Mr. M. Usama Zubair

CEP Report  
Parallelized implementation of Gauss-Seidel method using OpenMP and POSIX Threads

Muhammad Kamil 2016-EE-189 kamiljaved98@gmail.com

December 2020  
  
Department of Electrical Engineering  
University of Engineering & Technology, Lahore

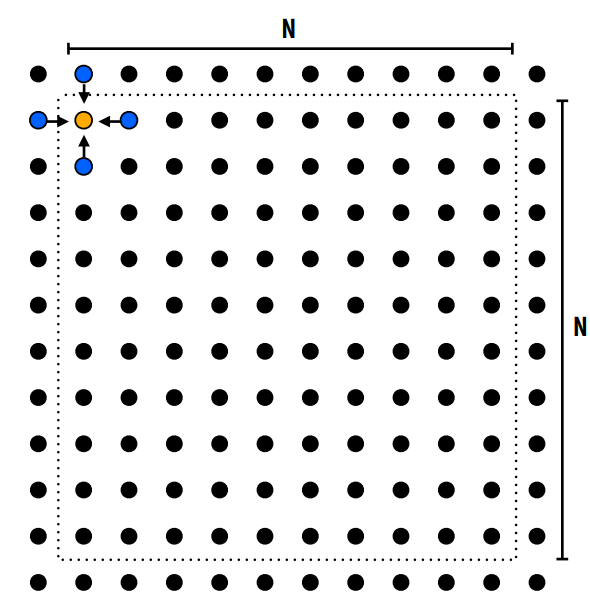
Table of Contents

|  |  |  |  |
| --- | --- | --- | --- |
| Section | Part | Title | Page No. |
|  |  |  |  |
| **I** |  | *PROBLEM STATEMENT* | 13 |
|  |  |  |  |
|  |  |  |  |
| **II** |  | *RED-BLACK CELLS APPROACH* |  |
|  | *1* | Sequential Implementation | 13 |
|  | *2* | Parallel Implementation with OpenMP  (a) using pragma omp critical  (b) using per-thread diff (no padding)  (c) using per-thread diff (with padding) | 16 |
|  | *3* | Parallel Implementation with Pthreads  (a) using mutex lock  (b) using per-thread diff (no padding)  (c) using per-thread diff (with padding) | 16 |
|  |  |  |  |
|  |  |  |  |
| **III** |  | *ANTI-DIAGONALS APPROACH* |  |
|  | *1* | Sequential Implementation | 13 |
|  | *2* | Parallel Implementation with OpenMP  (a) using pragma omp critical  (b) using per-thread diff (no padding)  (c) using per-thread diff (with padding) | 16 |
|  | *3* | Parallel Implementation with Pthreads  (a) using mutex lock  (b) using per-thread diff (no padding)  (c) using per-thread diff (with padding) | 16 |
|  |  |  |  |
|  |  |  |  |
| **IV** |  | *SUMMARY OF RESULTS* | 13 |
|  |  |  |  |

Section I

*Problem Statement*

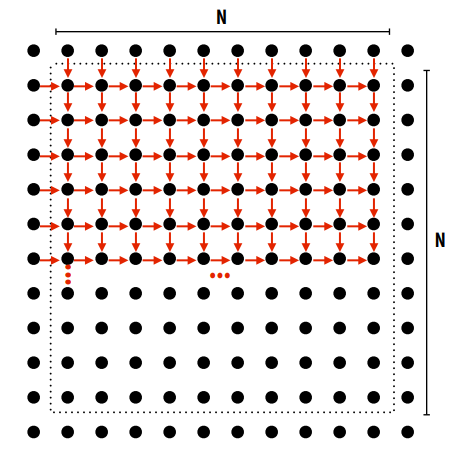
The task is to solve a partial differential equation on (N+2) × (N+2) grid (2D), in a parallel fashion, that is, to perform Gauss-Seidel sweeps over the grid until convergence. For a cell at point [i, j], the new value is calculated using the formula:



The simplest method to do a sweep would be to start at the first cell (indicated in yellow above), and move through the row, and then onto the next row. Repeat the process until convergence occurs.

To implement this in a parallelized method, some dependencies (per iteration over entire gird) must be taken care of, enumerated below.

1. Each row element depends on element to left.
2. Each row depends on previous row.



In this report, two approaches, namely (1) Red-Black Cells Approach, and (2) Anti-Diagonals Approach, are discussed, and the results of their implementation have been shared.

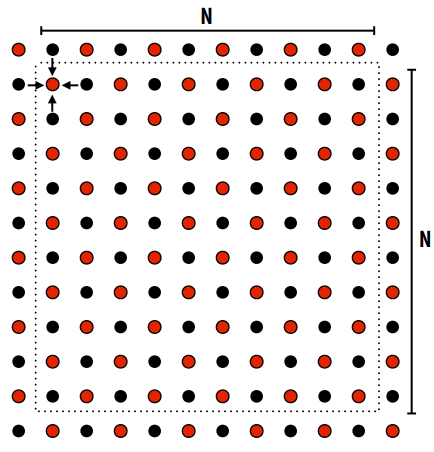
Section II.1

RED-BLACK CELLS APPROACH

*Sequential Implementation*

**Methodology**

In this approach, the alternate grid cells are bunched into red and black groups, as follows:



First, update all red cells in parallel. When done updating red cells, update all black cells in parallel (respect dependency on red cells). Repeat until convergence.

Hashing is the process of mapping a value of arbitrary nature to a function space, usually one that provides a fixed length value (string/number). In most cases, hash functions are aimed at being practically irreversible (one-way functions). This quality has important use in cybersecurity, such as signing and verification of data to detect tampered or corrupted data. SHA-1 is such a hash function, which scrambles the input to a 20-byte string (called the message digest) that is extremely hard to be reverse-engineered.

Mapping to a fixed length means that there would be collisions, and hence an output value cannot be mapped back to a sole input value. A method to map input to fixed length values is to use modulo arithmetic. Hashing has an important use in databases, in the process of Indexing, where a fixed-size index table maps to a set (table) of values or to an index table at a different level. A value in the index table represents a hash-value, that maps to a table of all the values whose keys have that hash-value. In this way, the key-value pairs can be divided and categorized into different sets based on the hashes (shorter values) of the keys. This results in easier and faster searching of key-value pairs based on the hash-values. The table mapping the different keys to a certain hash value is called a Hash Table.

**Partitioning of Key-space**

A method of mapping a set of keys to a singular value, which can then be used to index and fetch the initial keys (and corresponding data) and allow easier key-search in a store-fetch network, is to partition the key-space such that different ranges of data can be assigned to different nodes. This would reduce the load on a particular node in the system, and provide faster search based on the characteristics of the distribution.

Consistent Hashing is a method of doing such a partitioning. It treats the interval between two keys as a “distance”. A node in the system is assigned all the keys that are closest to it, in terms of their distance provided by some chosen function. An advantage of such a partitioning is that upon addition or departure of nodes, only the key-range for the adjacent nodes are affected and can be readily readjusted.

The Chord technique projects these keys onto a circle, of some fixed length. A key takes a certain place on this circle, which is its distance from the start-point. The nodes are termed as identifiers. Hence, the circle is divided into segments based on the distance between the different nodes. The keys within a segment are assigned to the identifier node at the end that segment, which is referred to as the “successor” for these keys. Chord is the network-setup and algorithm for Distributed Hashing used in this project.

**DHT Characteristics**

Through key-space partitioning, the keys can be distributed into a network of identifier nodes, and such a network, which maps the keys to different nodes, is called a Distributed Hash Table. These are especially employed in large-scale peer-to-peer data-sharing networks, for distribution of resources and de-centralization. This setup comes with additional benefits of scalability and recoverability. Nodes can join and leave at will, and their corresponding keys can be assigned and re-assigned on the fly. A greater number of nodes would mean relatively better query-response time and also a greater bandwidth for exchange.

A DHT may use *consistent* hashing or *rendezvous* hashing for the purpose of distribution of keys to identifiers. Different algorithms exist for maintaining reliability of these networks, by ensuring speedy recovery upon changes (such as node join, departure, failure) and consistency of information across nodes. An important aspect of DHT networks is that a node should be able to reach (or search) any other node in that network, within a fixed number of hops. A goal of overlay algorithms is to minimize the number of hops required in reaching a node from another, by providing information to a node about a number of other nodes. An example is the Chord algorithm, which is discussed in the next part. For more text on DHT, refer to the wiki at [R2].

Section I.2

*Chord (P2P DHT)*

The Chord technique, used in this project, projects the keys onto a circle, of some fixed length, referred to as the Chord Ring. A key takes a certain place on this circle, which is determined by its distance from the start-point. The nodes are termed as identifiers. Hence, the circle is divided into segments based on the distance between the different nodes. The keys within a segment (spanning between two adjacent nodes) are assigned to the identifier node at the end that segment, which is referred to as the “successor” for these keys.

The details of key-assignment, lookup and stabilization algorithms provided by Chord DHT are discussed in this part. For a detailed procedural explanation, refer to the Chord paper available at [R1].

**Hash Function and the Identifier Circle**

Say we have keys derived from some characteristic of the data, such as the name of an image, by first employing a hashing technique such as SHA-1 and then reducing the hash string (160 bits) to a byte-key (e.g. by bitwise XORing all the bytes) for simplicity. This provides enough randomness and fairly even distribution for large number of datapoints. Hence, the length of a key will be . In case of a node, the key can come from hashing the node’s IP Address and Port in the same manner.

Chord, which is essentially a distributed network for quick lookups, uses consistent hashing for assigning these keys to identifiers (the Chord nodes). It assumes a circle spanning from 0 to length , in our case 0 to 255, and places the nodes on this circle according to their identifier value, which we can call the node-ID, modulo .

Any given key is then assigned to the node for which the node-ID is the very next node identifier moving clockwise from on the circle. This node (or node-ID ) is called as the successor node for key . A node’s successor node is also determined in a similar manner, such that the node occurring right after , when moving clockwise from , is the successor of – we can represent this relation as . Hence, if is the key (specifically, Chord point) occurring right after and is the key equal to the node identifier value, then the range of keys that could be assigned to is . We can refer to node as the “*authoritative holder*” for keys in this range.

For further discussion, we’ll represent a node with node-ID 1 as N1, the key 4 as K4 etc. Assume the case for a Chord network with (Chord circle spanning numbers 0 to 63), having number of nodes, which are {N5, N15, N25, N33, N45, N53, N61}. Key assignment to different nodes in this network, for a few example keys, is shown in Figure 1 on the next page. We’ll use this network state as an example for discussing the different Chord procedures.

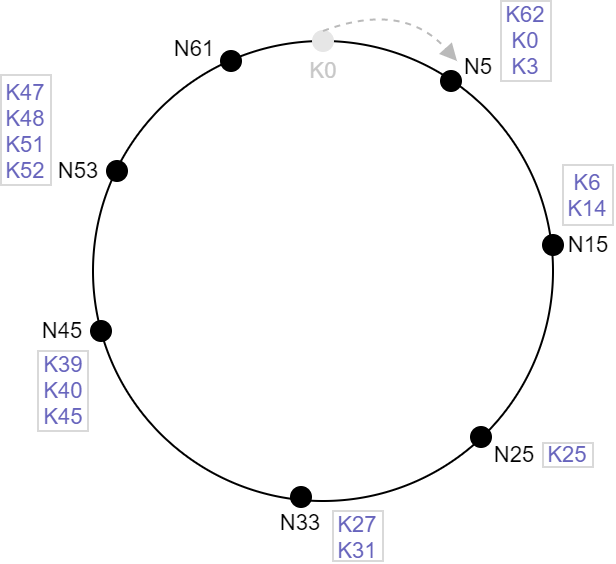
****

Figure 1

**O(*N*) Case**

For the simplistic case, each node is given information only about its successor on the Chord ring. In a practical network, this information could be obtained (shared across nodes) when a node joins in the network, discussed later on. Lookup in such a network can only be done by linear search. Consider that a query for key K51 is generated at node N15. It first checks its own key-space range, but upon not finding K51 there, it passes the query on to its successor, N25, which will pass it onto its successor (N33) and so on until the query reaches the node N53. N53 gets a hit for K51 in its range and storage, and the corresponding value (data) is either returned along the traversed path, or sent back directly to N15 if the information to contact the originator (N15) was encapsulated in the search query.

When a new node wants to join the network, it may send a join request to any known node. The known node then forwards this join request; it travels along the Chord until it reaches a node which contains the id in its covered key-range, such that . Node then splits its range, assigns for itself the range and then contacts to tell that (itself) is ’s successor and provides with its key-space range i.e. , where is ’s old predecessor. Note that no longer knows its actual successor (which is now ) – this could be fixed by sending a redirect message (containing info for node ) from node when it is next contacted by during a search or node-join (on-demand fixing). It can also be fixed using a periodic process wherein each node contacts its *assumed* successor, to update its successor info, if ’s successor has changed.

This method of searching a key (or the place to be given to a new joining node) in the network is *correct*, because any key/identifier value could be searched starting at any node – provided the successor is known. But this approach on its own is very slow. In the worst case all N nodes would have to be traversed to get to a desired key, which corresponds to time-complexity O(N).

In a practical network, where failures of nodes may occur, any fault would essentially break the chain and make it impossible to get to certain keys from certain locations. Hence, to improve reliability, additional information is kept at each node, discussed henceforth.

**Finger Table and the O(log*N*) Case**

The complete Chord algorithm calls for storing info of more than one other node at a given node. Particularly, a node stores info of its successor, its predecessor, and a number of other nodes stored in a table, called as ’s *finger table*. Note that these additional nodes are not required to be known for the correctness of the network, but rather for the improvement of query response-time (faster routing) and reliability.

Finger table for a node has (identifier length) entries. The node stored at index of the finger table of node , is the node that succeeds by a distance of at least on the Chord ring (using ), i.e. , where . A table entry stores the identifier for node as well as relevant information needed to contact . The node stored at would be , provided (which an identifier must be). Hence, ’s first finger table entry corresponds to the successor of , and so the successor need not be stored separately. The first few of rest of the nodes, for a network with large ring-length and many nodes, are generally ones that are in the vicinity of node , and the reach expands as we go to higher indexes in the finger table. This means that we have a variety of distances to choose from, for forwarding a given query, and we can determine and choose the most optimal node, for the hop, among the known ones.

Consider the case for Chord network given in Figure 1, where . The finger table for node N33 would be as follows:

|  |  |  |
| --- | --- | --- |
| Index |  | Node entry at |
| 1 |  | N45 |
| 2 |  | N45 |
| 3 |  | N45 |
| 4 |  | N45 |
| 5 |  | N53 |
| 6 |  | N5 |

Besides these entries, node N33 knows about its successor N45 () and its predecessor N25 (stored when the N33 received join request for predecessor N25). We now look at the routing procedure with the help of an example. Suppose that N33 is provided a query for key K55. It will know that K55 does not lie in its key-space range . It would then test whether the key lies between its ID and its successor’s ID i.e. in the range , which it doesn’t. Then, it looks in its finger table, to find the largest node-ID in the finger list that precedes the key 55. This can be done by starting at the highest index and moving backwards while testing if the finger at lies in the range (where for this search). As soon as we find the first entry that satisfies the range, we forward the request to that entry. In this case, the closest predecessor to K55 in the finger table is N53. Hence, the query is forwarded to N53, which will then forward it to its successor N61 having range – which would be the query end-point as the key doesn’t exist with the authoritative holder. So, using the finger table, we saved one hop (through N45) as compared to the linear search method.

Note that we don’t forward the query to a known successor of K55 (N5 in our table) because we cannot assume that N5 is the actual immediate successor for K55, and it would’ve been *incorrect* to do so, as the actual immediate successor (the authoritative holder) of K55 is N61 – which we aren’t aware of. Hence, this cautionary measure is in fact necessary to maintain correctness of lookups in the DHT.

At the same node N33, now assume that we have generated a query for K15. We’ll have to search the finger table for a suitable node to forward to, in this case also. Starting at the highest index, we get N5 as the closest preceding node (the largest one satisfying the range condition – the ranges wrap around as we’re on a circle). So, in this case, we’re forwarding the query to the farthest node we know of on the Chord (moving clockwise), thereby saving 3 hops. N5 would then forward the query to its successor N15, where the query is satisfied.

Note that the largest entry in a node ’s finger table is at a clockwise distance of at least ( is identifier length) from , and is half of (the Chord ring size). Hence, any node can, if needed, at maximum forward a query to at least half the distance across the Chord ring starting from (it’ll go further based on the successor located in the last finger entry). This means that, after each hop for a query , on average, the distance remaining to reach the authoritative holder node of , is at least halved. So, average number of forwardings needed is O(logN) in a Chord network with nodes maintaining finger tables. From the instance finger table, it can be noted that a node has more information about nodes in closer vicinity to it, but can also reach much farther distances if needed.

**Information Correction and Stabilization**

Knowing the correct successor is crucial to the correct working of the Chord DHT. As discussed earlier, a new joining node is made aware of its successor . The successor can also provide with ’s old predecessor , so that can set as its predecessor. But now, ’s information of its successor is incorrect (it may be but it should be ). In the original Chord algorithm, this inconsistency is fixed by running a periodic stabilization process at each node .

In this stabilization process, a node asks its known successor for the predecessor of (stored at ). If ’s successor information is correct, then should be equal to . But if , this means that one or multiple nodes were added in between and . In which case, we simply check if received has an ID in the range , and if it does, we assign as our (’s) new successor. Node then *notifies* its new successor () of ’s existence – upon which can update its predecessor if lies in the range or ’s known predecessor has failed. A separate process at each node periodically checks whether their predecessor has failed, so that a node is able to welcome any other nodes notifying it of being its possible predecessor.

For the complete Chord experience, finger tables must also be consistent (correct) for each node, to reduce the number of hops required in searching a key or a node. A periodic process is used for this as well. Each time this process runs, a counter determining the index of the entry to be fixed, is incremented. Hence, one entry is adjusted per execution of this process. For fixing of a given entry (, we simply run a query to find which is what should actually exist at . And hence, if the entry were previously incorrect, it is updated after a result for the issued query is received from some node in the DHT network.

Some additional measures that can aid in reconciling of the successor information, and handling changes such as node departure are also provided by the Chord algorithm, discussed briefly in the next part.

**Additional Measures (Successor List) and Node Departure**

If a node fails or ungracefully leaves the Chord network, then the successor information of its predecessor and the predecessor information of its successor becomes inconsistent. As discussed in previous part, the inconsistency caused in the successor’s predecessor value can be fixed by a regular checking of predecessor failure and by updating the predecessor when notified by a candidate node. But the leaving node’s predecessor’s successor information would be compromised and not readily fixable in most cases. This fault can be fixed by keeping a successor list at each node, having immediate successors of that node. This reduces the possibility of running into a situation where no working successor is known, by simply removing the failed successor(s) from the successor list and assuming the first working successor as the actual one. The successor list for a node can be easily stabilized by copying ’s successor’s successor list, adding ’s successor to the start of the list, and removing the last entry from the list. Upon successor failure, after identifying a new successor, the successor list is reconciled with it.

Furthermore, if a node were to depart from the network voluntarily, then before leaving it can do the courtesy of (a) providing its successor with its keys, and (b) notifying its successor and predecessor before leaving, informing them of each other, so that they can reconcile their predecessor value / successor list with each other.

Section II

Methodology

Section II.1

*Distributed Hashing and DHT Structure*

The methodology in these sections is described according to the Chord DHT network implemented based on the assigned CEP (PA2 Assignment) of the course. Further details of the problem statement are provided on the webpage at [R3].

**Node Class and Structure**

The *dhtn* (or *dhtdb*) class in the project defines the set of data and functions for a given node. A node may or may not store information (name, address, port) about a known (provided) node, based on whether it was the first node in the DHT. It contains an instance *self* of type *struct dhtnode\_t*, which holds information necessary to contact a node, such as address and port, as well as the identifier value (node-ID) assigned to that node in the Chord network. The hashing involved in assigning this ID is discussed in the next part.

Besides this, the node class contains a finger table , in which is the node’s successor, and an extra final entry is kept at (where = 8, for our byte-identifier network). is the identifier length for our network. Separately, an array is kept, having value at index with *,* to avoid their recomputing. Other than that, each node class has an instance of class, which handles the loading and caching of images (maintaining a *bloom-filter*), and serving of images to querying clients. The different functions a node has (or is capable of performing) are – function to initialize a new DHT network, function to request join to a known node, compute/recompute node-ID, functions to handle incoming packets, joins and searches, forwarding, generating search queries, and functions to fix inconsistent finger table entries.

**Node and Image Identifier Assignment**

A node is assigned an ID based on the SHA-1 value computed from its IPv4 Address and Port number. The SHA-1 message digest is then reduced to a single byte (values 0 to 255) by XORing together all its bytes. A node is the *authoritative holder* of keys in the range , in the Chord network.

Similarly, when *imgdb* is loading images into its database, it computes the SHA-1 message digest of the image’s name, which is then reduced to a byte value. 3 bits corresponding to XOR result of different bytes of the message digest, are set in the 64-bit bloom-filter to record the presence of an image. Upon searching, if all those three bits corresponding to the image name are found set, that means the image may exist in our range (and we can search for it), but if even one of those bits is not set, that means that the image doesn’t exist in our range (and we don’t have to search through our assigned key list).

Hence, an image is assigned a byte-size key (or identifier) based on its name, ranging from 0 to 255. And, it is the successor of this key in the Chord network that is assigned that key (the image). Hence, a node essentially loads all the images having an ID in the range into its *imgdb* bloom-filter, and search for an image having key/id in that range should end up at that node , if the Chord is in a correct state (all nodes know their correct successors i.e. ).

**Node Join Procedure**

When we provide a new node-class with the contact information of a node already in the Chord network, it sends a JOIN message (containing ID and contact address of ) to this node . Upon receiving the JOIN message, disconnects with the sender. The node sees if the ’s ID lies in its key-space range. Assume that it doesn’t, then will check in its and get the largest ) value preceding ’s ID. Then, it gets the corresponding node entry in the finger table, and checks if ’s ID lies in its range (or after it). If it lies before it, then we set a bit in the message type showing our expectation that the query should be satisfied by . Node forwards the query to and waits for a response. If ’s ID lies in ’s range it will close the connection with contact , send a WLCM message providing it with a successor () and predecessor (’s ); also updates its own predecessor as well as rest of finger table entries. Node reloads its *imgdb*-database based on the new key-space range it is authoritative for, and would load images in the range based on ’s ID and provided predecessor’s ID.

But if *j*’s ID does not lie in ’s range either and had set the expectation, it means that has a bad finger table entry, for which will suggest a replacement (’s predecessor) to – node will update its finger table entry, fix the rest of its finger table and then redirect the node-join request to the suggested node. If had not set an expectation, that means the ’s ID might’ve lied beyond ’s range (moving clockwise in the Chord), in which case will further forward the join request to a suitable node in its finger table, and the forwarding process would continue until the node having ’s ID in its key-space range is reached, upon which that node will contact directly and send it a WLCM message as described before. At any point in the forwarding process, if the ID provided by collides with the current node’s ID or node’s known predecessor ID, a REID message is sent to the join-requester to tell it to set a different ID for itself and then try re-joining.

**Image Search Procedure**

If an image queried by a *netimg* client is not found in the *imgdb* bloom-filter or the range of images loaded for a *dhtn* node, the *imgdb* instance returns a MISS to the main node process, upon which the *dhtn* node generates a SRCH message/query for propagation though the DHT network. It includes *self* as the originator node of the SRCH query, so that it can be contacted by the authoritative node, and the name and ID of the requested image.

In a similar manner as before (in the node-join process), a suitable node in the finger table is found to forward the request to, expectation is set based on the existence of image ID in the selected node’s key-space range, wait is done for a REDRT message etc. The query receiving node does similar things as before too – searching its key-space range for image ID (and searching its database range for image), checking if expectation is set to decide whether to further forward the SRCH query or to suggest the sender of a different finger[] entry replacement (current node’s predecessor is suggested) to redirect the message to (and to indicate the sender to update its finger table). If the image ID is in the current node’s key-space range, but doesn’t exist in its *imgdb* bloom-filter or database, then the node will contact the SRCH originator provided in the SRCH message to inform it of a database MISS. If the image is found in current node’s database range, it sends the originator node a RPLY message instead, which in this project, is a way of allowing the originator to cache the image ID into its bloom-filter, load the queried image and send it to the querying *netimg* client.

Section II.2

*Fixing of Inconsistent Finger Tables*

Initially, when the first node starts a network, it sets all its finger table entries to *self*, such that the predecessor () and successor () are also *self* (which is correct for a Chord network with only one node).

As discussed in the previous section, the finger table entries of a node are fixed whenever,

**(a)** a node-join occurs (JOIN message is received) and the joining node happens to be in ’s

authoritative range

**(b)** a WLCM message is received upon attempting to join a Chord network

**(c)** a REDRT message is received during forwarding of a JOIN or SRCH message

In each case, a certain node is (or two nodes are) received based on which the fixing is done. We have two functions that perform the updating of the finger table:

* *fixup*(index ) – which fixes all entries after index , based on the node-ID of , till index , unless its range-condition is violated midway, and,
* *fixdn*(index ) – which fixes all entries before index , based on the node-ID of , till index 0, unless its stop condition is met.

For case (a), the joining node is set as ’s predecessor (), and then *fixdn*() is called to set all entries below index , satisfying a certain range, equal to the new node in finger table (); the range to be satisfied is discussed shortly. If the newly joining node happens to be the second node entering the DHT network, then it is also set to be ’s successor () – and *fixup*(0) is called to set all entries above index 0, satisfying a certain range, equal to the new node in the finger table (), unless a stop condition is met.

For case (b), two nodes are received in the WLCM message, the first is the sender of the WLCM message (successor) which we set as our successor () – and call *fixup*(0) – and the second node received is the sender’s old predecessor, which we set as our predecessor () – and correspondingly call *fixdn*().

In the last case, (c), receiving a REDRT message means that we expected the selected finger entry, say , to which we forwarded the JOIN/SRCH query, to have the query ID in its key-space range (we informed it of our expectation by setting the bit in the type of the message we sent), but the receiver () noticed that the ID does not lie in its range – it probably lies before moving anti-clockwise on the Chord, but somewhere after the point of the node forwarding the message moving clockwise. Such a case is termed as *overshooting* on the part of forwarder . This means that there exists a node that the receiver (of forwarded message) is aware of but isn’t and which should be at the place of of – as ’s ID is lesser than ’s current and larger than of . The best candidate for such a node () is the receiver’s known predecessor (receiver’s ) as it is the lowest-ID known node by the receiver (of forwarded message) that is still larger than ’s , and hence a better candidate for ’s . The receiver hence replies with a REDRT message containing its predecessor ( receiver’s ), and node replaces its with this suggested node . It then calls *fixup*() and *fixdn*() to fix the values above and below index in its finger table.

Lets now take a look at what the *fixup*(index ) and *fixdn*(index ) actually do. What they do is work with the information they have, and basically replace as many entries as they can with the entry at provided index , based on satisfaction of the range-constraint of the different finger table entries. These functions, for reference are provided below.

*// fixup: fix finger[] entries after given index*

void **dhtn**::

**fixup**(unsigned char j)

{

  for (int i=j+1; i<DHTN\_FINGERS; i++)

  {

    if (!**ID\_inrange**(fIDs[i], self.dhtn\_ID, finger[j].dhtn\_ID))

      break;

    finger[i] = finger[j];

  }

}

*// fixdn: fix finger[] entries before given index*

void **dhtn**::

**fixdn**(unsigned char j)

{

  for (int i=j; i>=0; i--)

  {

    if (fIDs[i] == finger[i].dhtn\_ID)

        break;

    else if (**ID\_inrange**(finger[j].dhtn\_ID, fIDs[i]-1, finger[i].dhtn\_ID))

    {

      finger[i] = finger[j];

    }

  }

}

In the case of *fixup*(), we consider that entry at index in the finger table of node (having ID ) should be the successor to the point , which for entry can be checked by testing if lies in the range – if it does for the distances corresponding to entries after , that means is a predecessor for those previously stored entries at after index . So, we replace those entries with . As soon as we reach the point where does not lie in the range , that means that lies before that , and is no longer a candidate successor for further distances. Hence, we have accounted for the added node in the finger table for the entries after .

In case of *fixdn*(), we move backwards from index , and for each index , check if entry lies in the range because if it does, that means that is a successor to that Chord point and a predecessor to that present entry, i.e. is closer to than is to – hence, we replace all those with . When, for an index before in the finger table, a condition such that is equal to the present entry , occurs, this means that the range consists of only one point (). And so, the present entry is the correct one as its existence (we assume that nodes don’t leave the DHT) means that it is the only possible node-ID that should be allowed to take the point of . We stop walking further backward before the index where this occurs, because we can be sure that if the node is the accepted successor to some where , then can’t be the immediate successor for any with , as a correct node is known to exist such that it occurs before and after . It may also be the case that they’re all the same point – but we needn’t check for that.

Section IV

Results

Section IV.1

*DHT Setup*

In this part of testing and obtaining results, as guided by the assignment rubrics, 5 random numbers were generated, in the range [0, 255], from rand() function in C using my roll no. (189) as seed. These 5 numbers were taken as node-IDs, and a Chord network was obtained by first starting with one node and then having other nodes join in.

The random numbers were generated using the following program *randroll.cpp*:

#include <stdio.h>

#include <stdlib.h>

int **main**(void)

{

*// Use roll number (189) as seed for random generator*

**srand**(189);

*// [N, M] = [0, 255]*

    int N = 0, M = 255;

*// generate and print the random numbers*

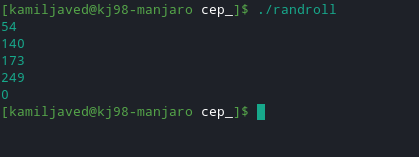
    for(int i = 1; i<=5; i++)

**printf**("%d\n", N + **rand**() / (RAND\_MAX / (M - N + 1) + 1));

    return 0;

}

When run in the terminal on my machine (name *kj98-manjaro*), it gave the following output:



Hence, the node IDs used were {54, 140, 173, 249, 0}. The DHT was started with node 54 as the first node, and then node 140 was added by sending join request to 54, then 173 was added by sending join request to 140, then 249 using join request to 173, and finally 0 using join request to 249.

The terminal outputs, during the join process, for each node, in the order {54, 140, 173, 249, 0}, including their finger tables after completion of all joins, are shown in the Figures henceforth.

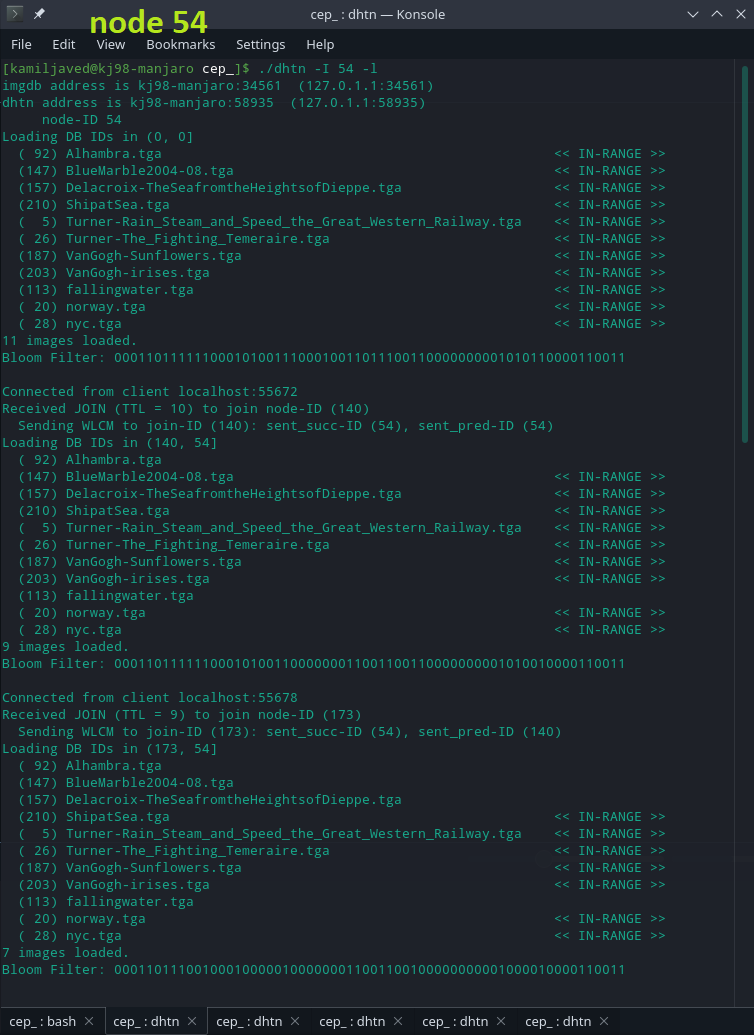
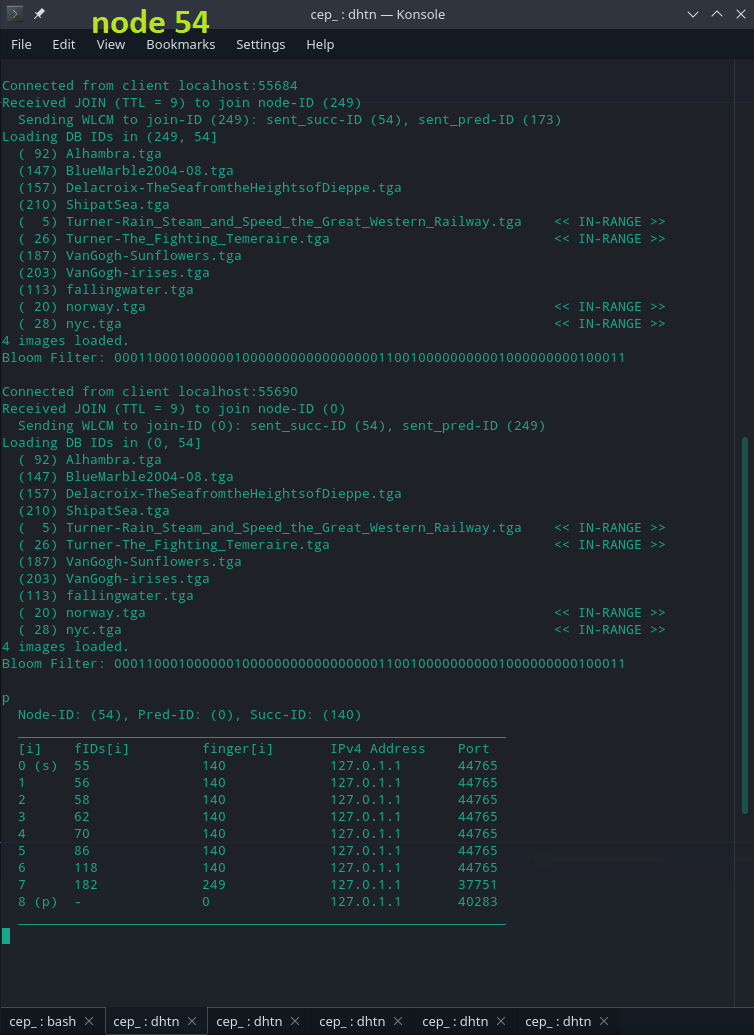


Figure 2 (a): Node 54 terminal output part-1



***COTINUED***

Figure 2 (b): Node 54 terminal output part-2

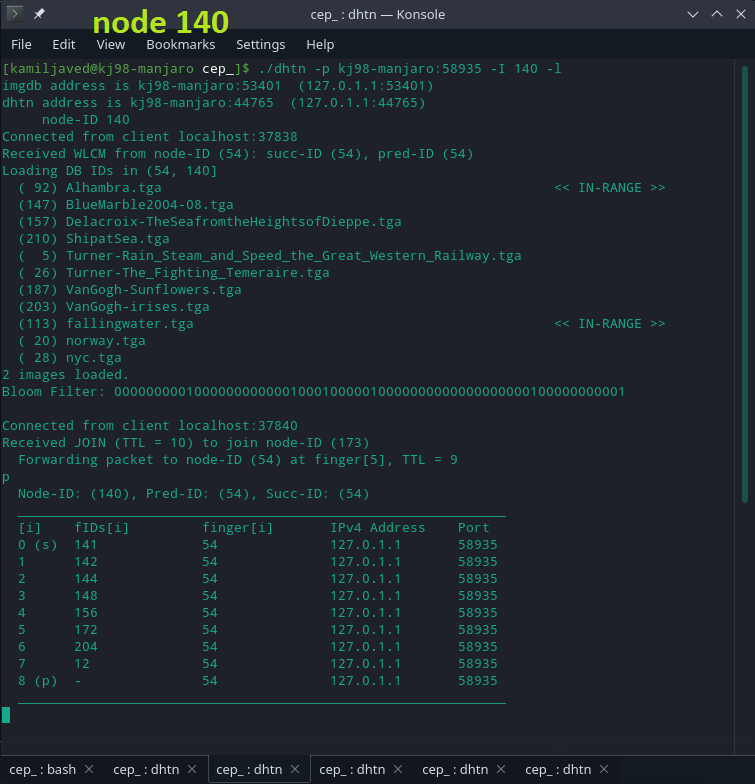


Figure 3: Node 140 terminal output

*Next image is provided on the next page.*

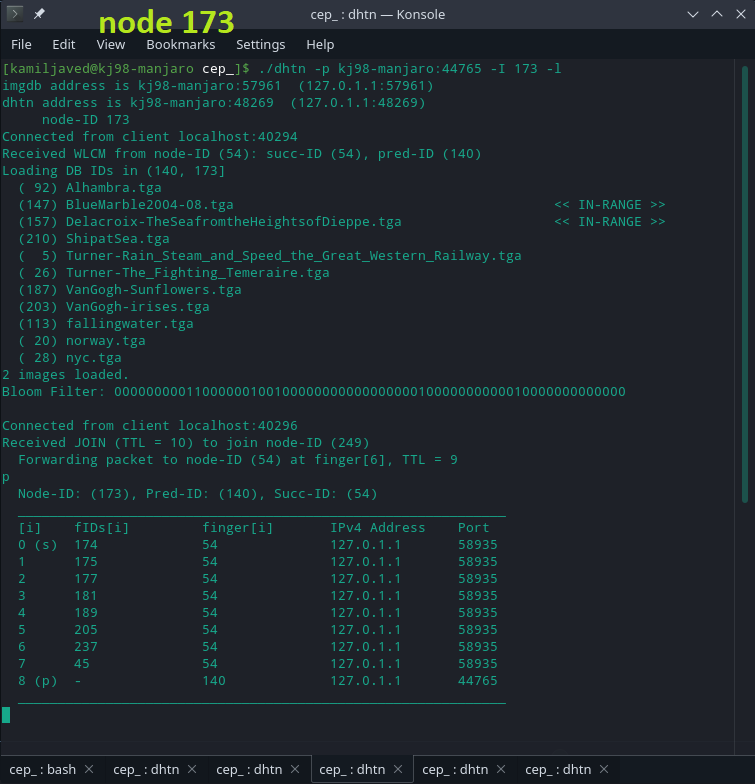


Figure 4: Node 173 terminal output

*Next image is provided on the next page.*

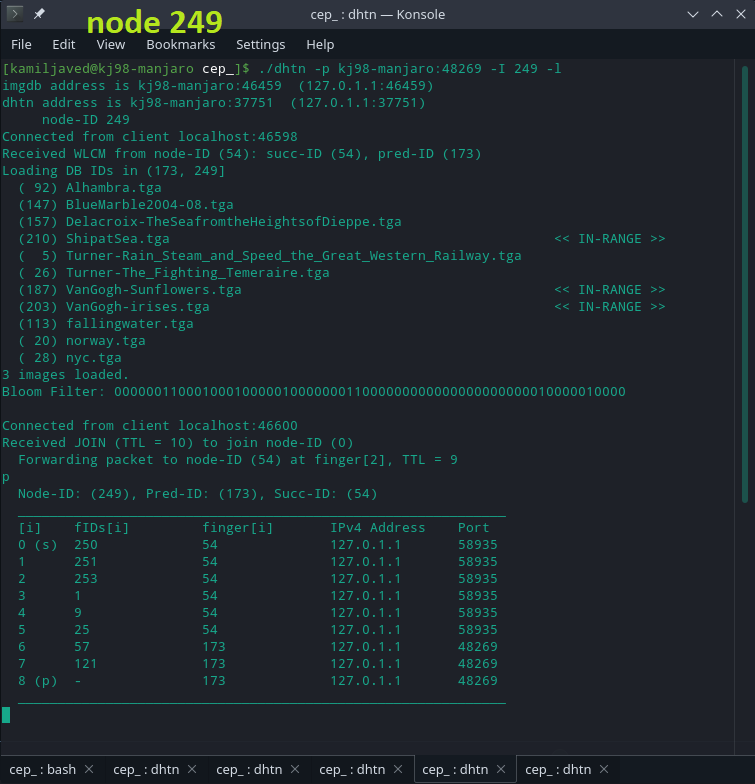


Figure 5: Node 249 terminal output

*Next image is provided on the next page.*

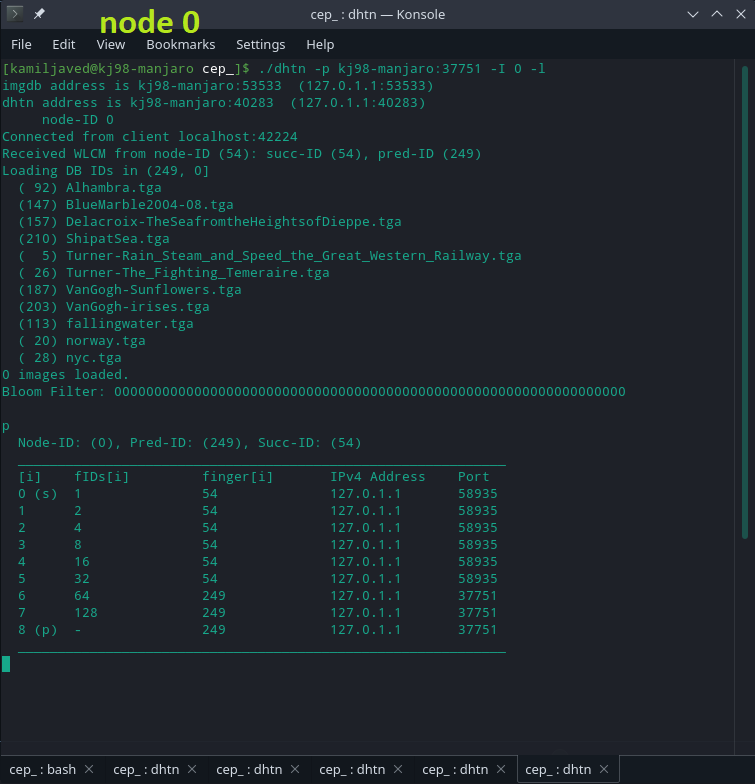


Figure 6: Node 0 terminal output

*Brief discussion of the results is provided on the next page.*

As can be seen in terminal outputs of the joining nodes, each join request gets forwarded (eventually) to node 54. This is because, for each of the join requests made after node 140 has joined, the join-request-receivers have node 54 in their finger tables, and 54 happens to be the suitable node for forwarding, based on the method using to determine the suitable finger entry for forwarding discussed in Sections I.2 and II.1, for each of the join-cases. As a result, node 54 has to split its key-space range and reload the image database multiple times, as shown by Figures 2(a) and 2(b), after serving each join request. Successor & predecessor IDs provided by the node 54 to each of the joining-nodes are shown along with the WLCM message; the nodes use these to initially fix their finger tables.

The node 54 knows its true successor and predecessor, as it is the one serving join-requests of all the nodes. The rest of the nodes, except for node 0, only correctly know their predecessors, based on the fact that they joined after their final predecessors had already joined; but don’t know their actual successors because their final successors arrived in the DHT after their joining. For the case of node 0, it joined after node 249, and is the final predecessor to node 54, hence node 54 was able to inform node 0 of its correct predecessor (node 249) and successor (node 54). Whereas for the rest of the joining nodes, they still assume node 54 as their successor, which was true at the time of each of them joining, but was rendered obsolete upon joining of new nodes occurring at a distance after them in the ring (moving clockwise) but before their known successor.

The rest of the finger entries, may not be correct either, calculated based on *successor after a minimum-distance* for each index, and are based on only the known successors and predecessors, as no redirections (which allow the nodes to fix their finger table entries) have yet happened in the network. The finger table entries for the different nodes would be updated (fixed) based on further node additions in the network, and the redirections received during the forwarding process of join requests and image queries. The final loaded images and the present state of the finger tables, for each of the nodes, is provided in the above figures.

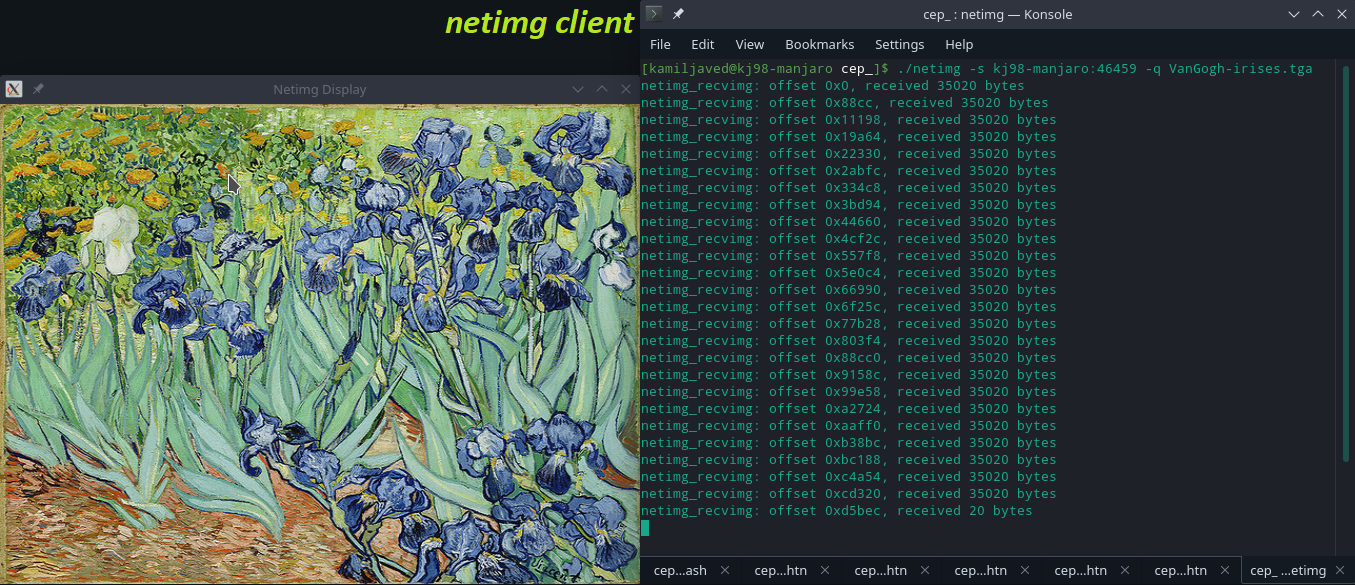
Note that the extra parameter - is used in the execution command for all nodes; it is used to make the nodes use the local IPv4 address for putting in their structs (which are forwarded during joins and searches), so that the nodes are able to contact other nodes on the local network. If - is not used, then in the current implementation, the nodes would put the router global-IP (which has been defined in *socks.ccp* file, or can be provided as argument) as the IPv4 address in the *self* struct (the shown address on which they are supposedly listening). This would be a problem if arbitrary ports are assigned to the nodes, as they’d be unreachable if the ports assigned are not properly forwarded by the network (gateway) router (which is using NAT). Hence, the intent of using the program in a local environment in shown by providing the parameter - in the terminal command. More information about the different additional parameters, and the use of program for across-the-internet Chord DHT setup, is provided in Section V.

Section IV.2

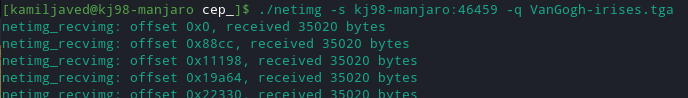
*Image Queries*

**Image Query 1**

In this test, a query for an image was made to a node that is the authoritative for the byte-key of that image i.e. the node to which the query is made has the image in its range and has it loaded in its *imgdb* database. The output shown at the client terminal (and spawned OpenGL window) is shown below.

****

A closer look at the issued query at the client is given below:



The image is queried from the *imgdb* server of node-ID 249, which has the queried image-ID (203) loaded in its database.

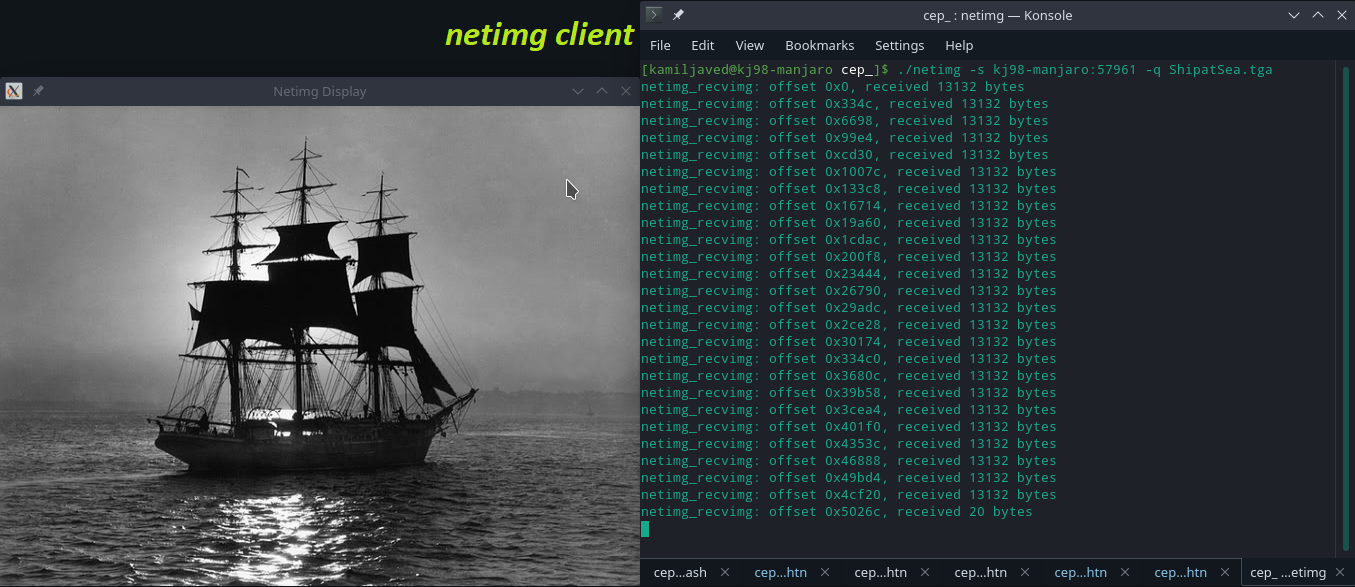
The *imgdb* server running at node 249 successfully finds the image in its range, and serves the *netimg* client.

The terminal output, after issuing the query at the client side, of the serving node (249) is shown in the figure on the next page.

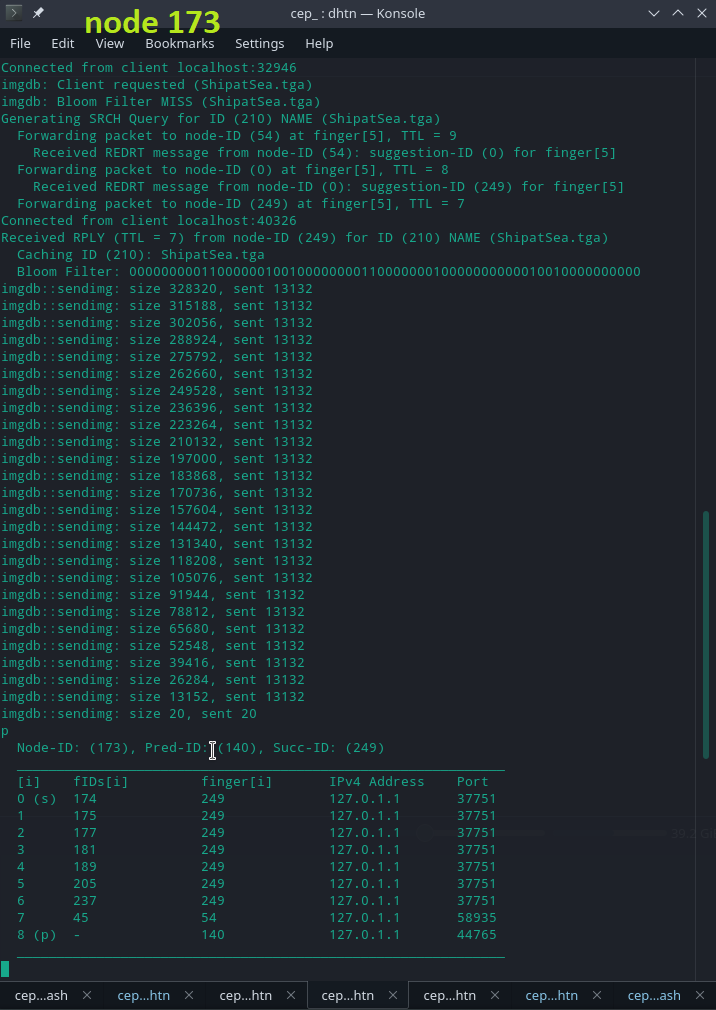
****

**Image Query 2**

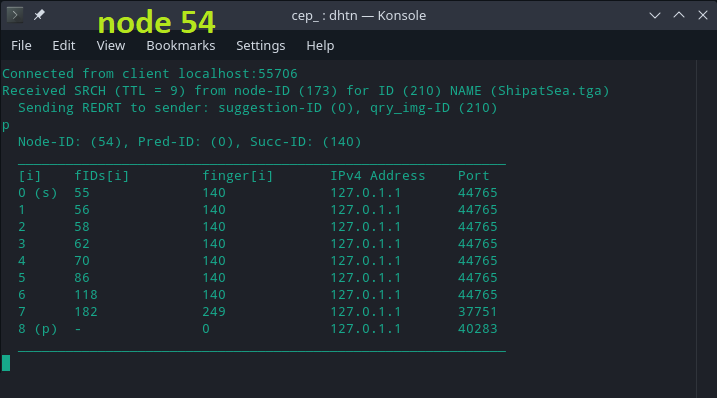
Then, an image was queried from a node that existed in some other node’s database. The queried image (ShipatSea.tga) had the ID 210, which is in the range of node 249. But the query was made to the *imgdb* client of node 173. As can be seen in the provided screenshots (starting at the next page), the node 173 generates a SRCH query for the ID 210, and first forwards it to node 54, from its finger table (provided in Figure 4) at index 5. But node 54 doesn’t have 210 in its range and so it replies with a REDRT message suggesting its predecessor (node 0) to the node 173. Now node 173 replaces its finger[5] (previously 54) with node 0, fixes its other finger table entries, and then re-transmits the query to the new finger[5] i.e. node 0. Node 0 also replies with a REDRT message, suggesting node 249. Node 173 now replaces its finger[5] with 249, again adjusts its other finger table entries, and then re-transmits the SRCH query now to node 249. Node 249 does have 210 in its range and also in its database, and hence it replies to the node 173 with a RPLY message, thereby allowing node 173 to cache and load the image to its database and serve the *netimg* request.

****

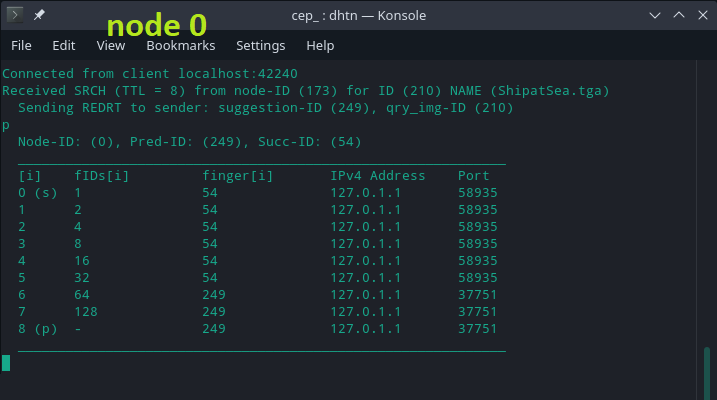
Terminal Output at queried node (173)

****

Node 54 terminal output (redirects node 173 to go to node 0), also showing finger table (unchanged)

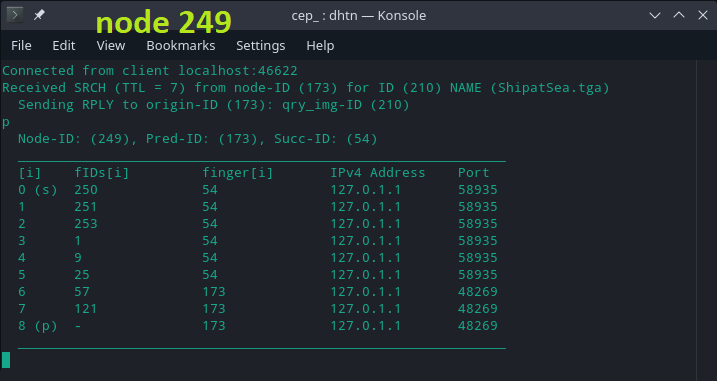
****

Node 0 terminal output (redirects 173 to go to node 249), also showing its finger table (unchanged)

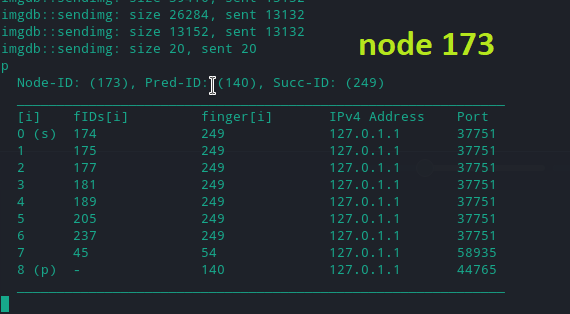
****

*Further images are shown on the next page.*

Node 249 (authoritative for queried ID 210) terminal output (replies to originator 173 with permission to cache image and serve client), also showing finger table (unchanged)

****

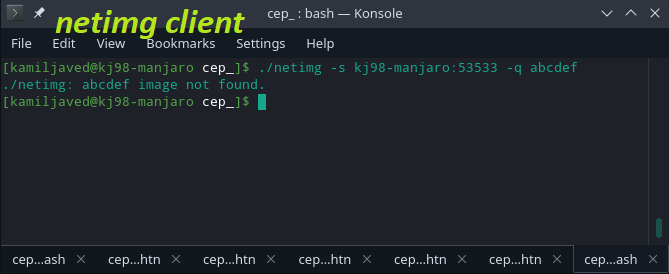
The finger table for node 173, the query originator node, after completion of the query, is as follows:



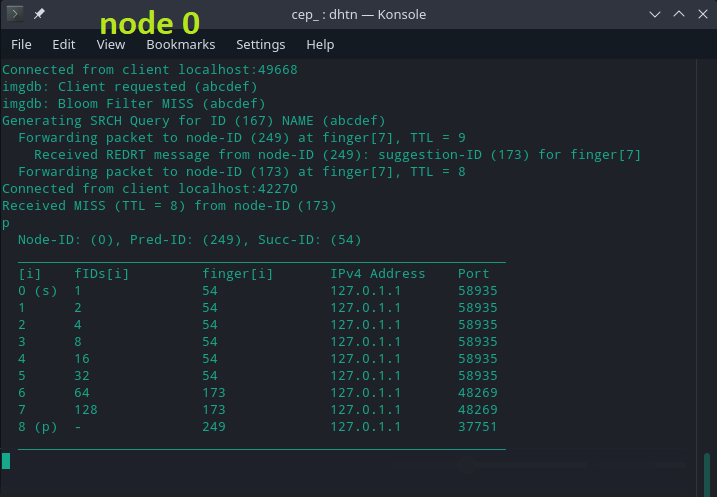
As can be seen by comparing this table to the old finger table for node 173 (Figure 4), changes have happened in 173’s finger table due to REDRT messages received during the lookup. After the completion of the query, based on the redirections received, node 173 has learned that entries at indexes 0 through 6 in its finger table should be 249 (instead of old value of 54). Note that node 173 has successfully learnt its actual successor (249) and predecessor (140) on the Chord ring, and has also fixed up its finger table to a large extent (only index 7 is an incorrect node – should be node 0). Hence, based on the algorithm for updating finger table values, with only a few search queries, the finger tables entries can be corrected significantly.

**Image Query 3**

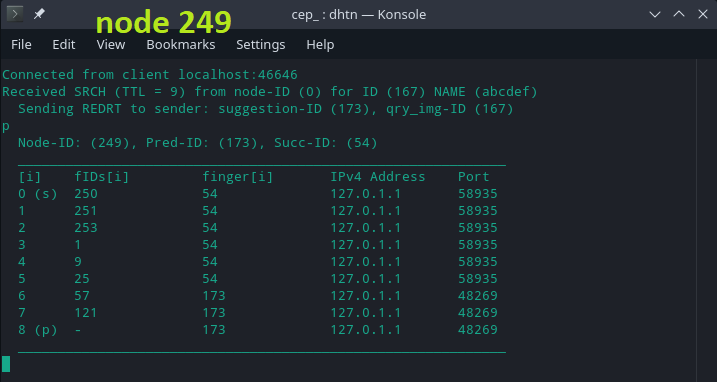
In this test, an image was queried (from some node in the DHT network) that didn’t exist in the database (not in any node’s key-list). The queried image name (abcdef) had the ID 167, which is in the key-space range of node 173. But the query for 167 was made to the *imgdb* client of node 0. As can be seen in the provided screenshots, the node 0 generates a SRCH query for the ID 167, and first forwards it to node 249, from its finger table (provided in Figure 6) at index 7. But node 249 doesn’t have 167 in its range and so it replies with a REDRT message suggesting its predecessor (node 173) to the node 0. Now node 0 replaces its finger[7] (previously 249) with node 173, fixes its other finger table entries, and then re-transmits the query to the new finger[7] i.e. node 173. Node 173 does have 167 in its key-space range but not in its database, which means that the image with that ID does not exist in the entire database of the Chord network, and hence node 173 replies to the node 0 (query originator) with a MISS message, thereby allowing node 0 to inform the requesting *netimg* client with an ‘*image not found*’ message. The terminal output at the *netimg* client is as follows:

****

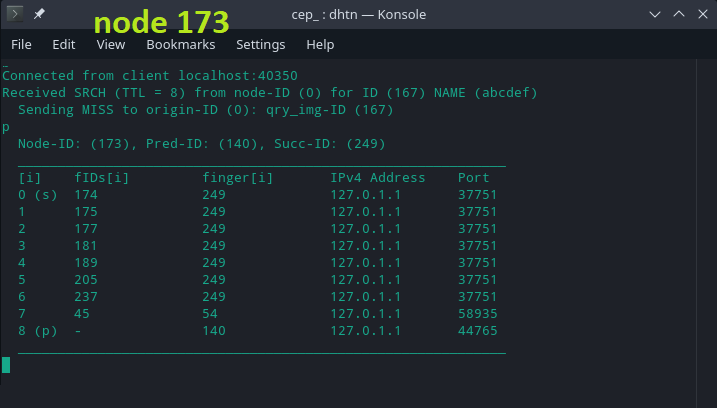
Terminal Output at queried node (0), also showing updated finger table (after query completion):



Terminal Output at node (249) (redirects node 0 to forward to node 173), also showing its finger table (unchanged):

****

Terminal Output at node (173) (authoritative for nonexistent-image ID 167, replies with a MISS), also showing its finger table (unchanged, last changed from previous test):



Section V

Additional Work

Section V.1

*Chord DHT Network across the Internet*

**Configuration Steps**

A number of things were needed to be done to have the nodes be visible across the internet. Specifically, the *dhtn* node server, and the corresponding *imgdb* server, both had to have public IPv4 addresses (for internet referral) and both had to have ports which were *open*, so that the listening servers could be reached across the internet and through the server’s gateway router.

The machines on which servers were running were connected to the internet through a Wi-Fi router, which assigns local IP addresses to each of the connected devices, and uses Network Address Translation (NAT) to remap the network addresses of the devices to into a different IP address space (a public one). The router basically exhibits its own public IP address as the destination for packets incoming to the connected devices – such that all the connected devices effectively have the same public IPv4 address which is equal to the public IP of the router. This can be confirmed by using services like *whatsmyip* and checking that the internet-visible IP addresses of each of the connected devices is the same as that of the router public IP (shown on the router configuration page, for most Wi-Fi routers). Hence, the public IPv4 address of the router (to which the device running a *dhtn*/*imgb* server is connected) can be used to reach the node server. The node server can simply provide INADDR\_ANY as IPv4 address to the bind function to be able to accept incoming connection from all available addresses on the machine, but the node will have to put the router global IP (at the time of starting the server) in the *dhtnode\_t* struct *self* so that other nodes across the internet can reach it (send it messages like WLCM, REID, REDRT, RPLY, MISS etc. in our application).

Furthermore, a Wi-Fi router wouldn’t just forward an incoming packet to the machine running the server – it would have to be told that the packets coming at a given Port # (on the public IP address) should be forwarded to the machine running the listening server. This is done by port-forwarding (or Virtual Servers setup) using the router configuration page. A static local IP can be set on the machine running the server, and the router can be told to forward requests coming at address {Public Router IP, Port-External} to the static IP device with address {Static Local IP, Port-Internal}, where Port-External would be the port which the incoming request provided, and the Port-Internal is the port the server is running on in the local machine. For convenience, the Internal and External Port values can be kept the same (same range), so that we can just use the port # we set on the server to get to the server from some other node. Hence, a summary of the steps involved is:

* Use the router public IP address when filling up the *self* structs of the nodes (address of the router they’re running on), if in a NAT configuration.
* Forward the ports which the servers are listening on, by configuring the router to forward incoming request at those ports to the ports of the provided Static-IP device

**Code Alterations**

Some extra command line parameters were used to setup (start) the node according to the use-case (local running or public running):

* : (used when in local mode)

Tells the node to use local IP address (127.0.0.1/127.0.1.1) when filling the struct (for both *dhtn* and *imgdb* servers), so that port-forwarding

wouldn’t be needed when using the program locally.

* : *<+argument\_\_IPv4\_string>* (used when in public mode)

Provides the node with the public router IP address (in argument), to use when filling the struct (for both *dhtn* and *imgdb* servers), so that the node can be contacted by other nodes across the internet (to receive WLCM,

REID, REDRT, RPLY, MISS etc. messages and *netimg* client requests).

* : *<+argument\_\_IPv4\_string:Port>* (used when in public mode)

This is used in replacement for the argument (which uses sname:port as argument). It provides the node to be run with the public IP and Port of the node (already in DHT) to connect to.

* : *<+argument\_\_dhtn\_Port>* (used when in public mode)

This is used to provide a custom port # (one that we have forwarded in the router) for the *dhtn* listening server to bind to.

* : *<+argument\_\_\_imgdb\_Port>* (used when in public mode)

This is used to provide a custom port # (one that we have forwarded in the router) for the *imgdb* listening server (of the node) to bind to.

Besides these, a command line parameter - <serv\_public\_ip:serv\_opened\_port> was added to *netimg* client so that it can reach an *imgdb* server running on the internet.

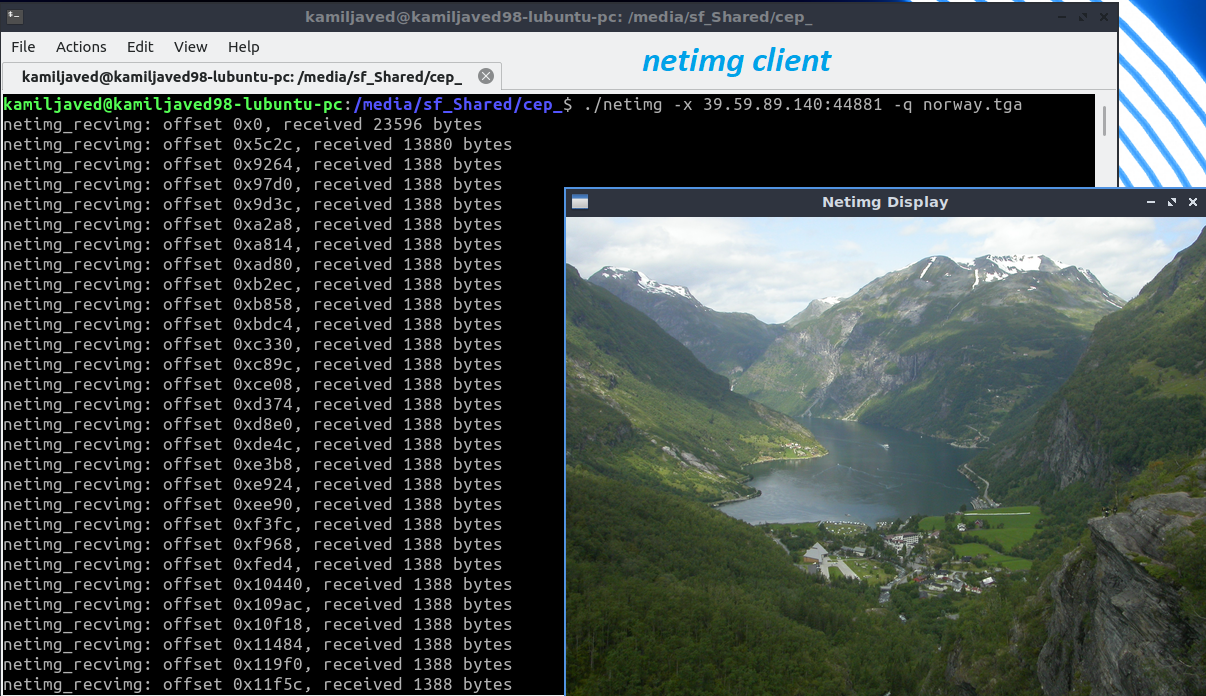
**Results**

First, a test for over-the-internet transfer of image was done by requesting an image through a *netimg* client, running on a cellular hotspot network, from a node’s *imgdb* server (having all images available) running on a Wi-Fi network (after port forwarding).

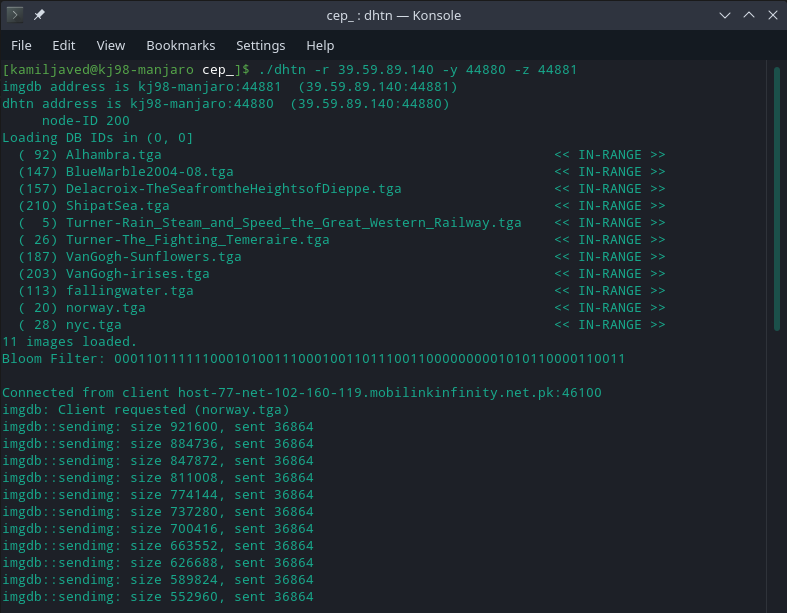
The IP address provided to the *netimg* client is the global IP address of the router to which the server-running device was connected.

*The terminal outputs of the client and the server are provided on the next page.*

Terminal output (and spawned OpenGL window) at *netimg* client:



Terminal output at the queried server node:



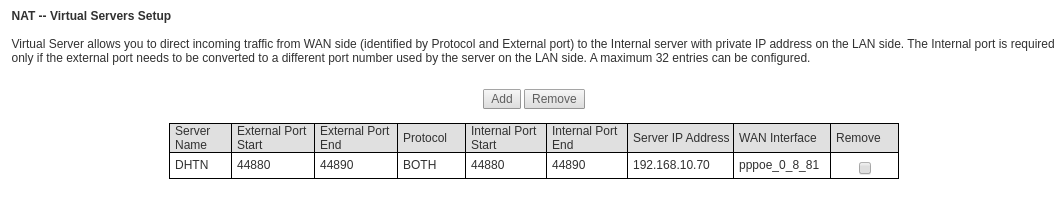
Note that the IP address in the line “Connected from client *host-77-net-102-……………”* indicates that the request has been received from a device that is not on the local network of the server, but it has come from a device across the internet (from my laptop running on Mobile Hotspot i.e. Cellular Data).

For the next (over-the-internet) test, a Chord DHT network was set up between two devices on different networks.

This across-the-internet DHT test was first tried by collaborating with my classmate Usman Ahmad (2016-EE-169) and the Chord network was successfully set up with multiple nodes running in each of our computers. But we did not record the results at that time, due to being in a hurry and also electricity running out at my side. We scheduled the test for Monday 15-June-2020, but on that day, Usman’s Wi-Fi Broadband network connection was down. As discussed earlier, being on a network that allows port-opening is crucial for being visible on the internet for our application, and Cellular Networks do not provide the ability to open/forward ports on connected devices for listening to incoming connections (acting as a server), so Mobile Hotspot couldn’t be used here.

However, I managed to get hold of a separate Wi-Fi network, thanks to my generous neighbors, to test the DHT application across different networks. Hence, the results provided henceforth have been recorded on my machines, a ‘computer’ and a ‘laptop’, but they have been connected to different Wi-Fi networks, one mine and the other our neighbors’ respectively. This may mean that the packets don’t essentially travel farther into the internet *cloud* but may have been routed within my regional zone. Nevertheless, it does show that the listening nodes are visible and reachable through the internet.

First, Port-Forwarding in the routers of both the networks was done, by selecting a range of ports to be forwarded for the set static IP of the machine to be used for serving. The image below is for the forwarding done for the network the ‘computer’ is on (*network1*).



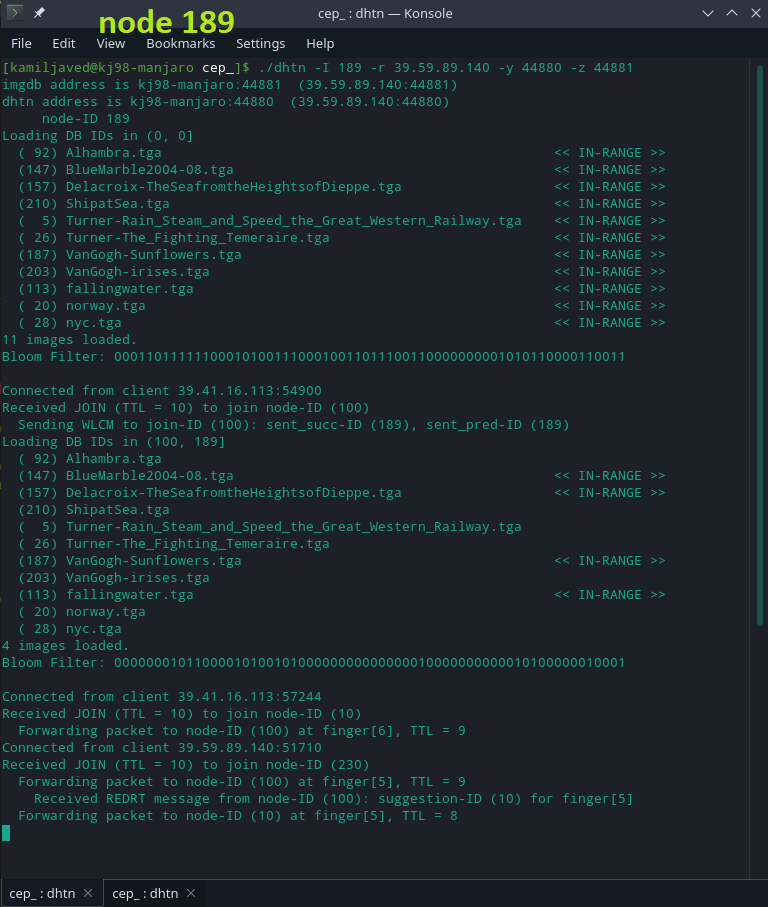
Similarly, range of ports 44550 to 44560 were forwarded on the *network2* router for the static IP set for the ‘laptop’ device. Then, the DHT was set up in the next step.

So, first a DHT network having the nodes with IDs 189, 100, 10, and 230 was set up. At the time of running, *network1* router had the public IP address 39.59.89.140, and *network2* router had the public IP address 39.41.16.113. The DHT was started with the node 189 running on *network1* (39.59.89.140), then node 100 running on *network2* (39.41.16.113) was added to the DHT by sending a JOIN request to node 189. After that, node 100 running on *network2* (39.41.16.113) was added to the DHT by sending a JOIN request to node 189. Finally, node 230 running on *network1* (39.59.89.140) was added to the DHT by sending a JOIN request to node 189.

The terminal outputs of the different nodes in the order {189, 100, 10, 230}, during the complete join-process, are shown henceforth.

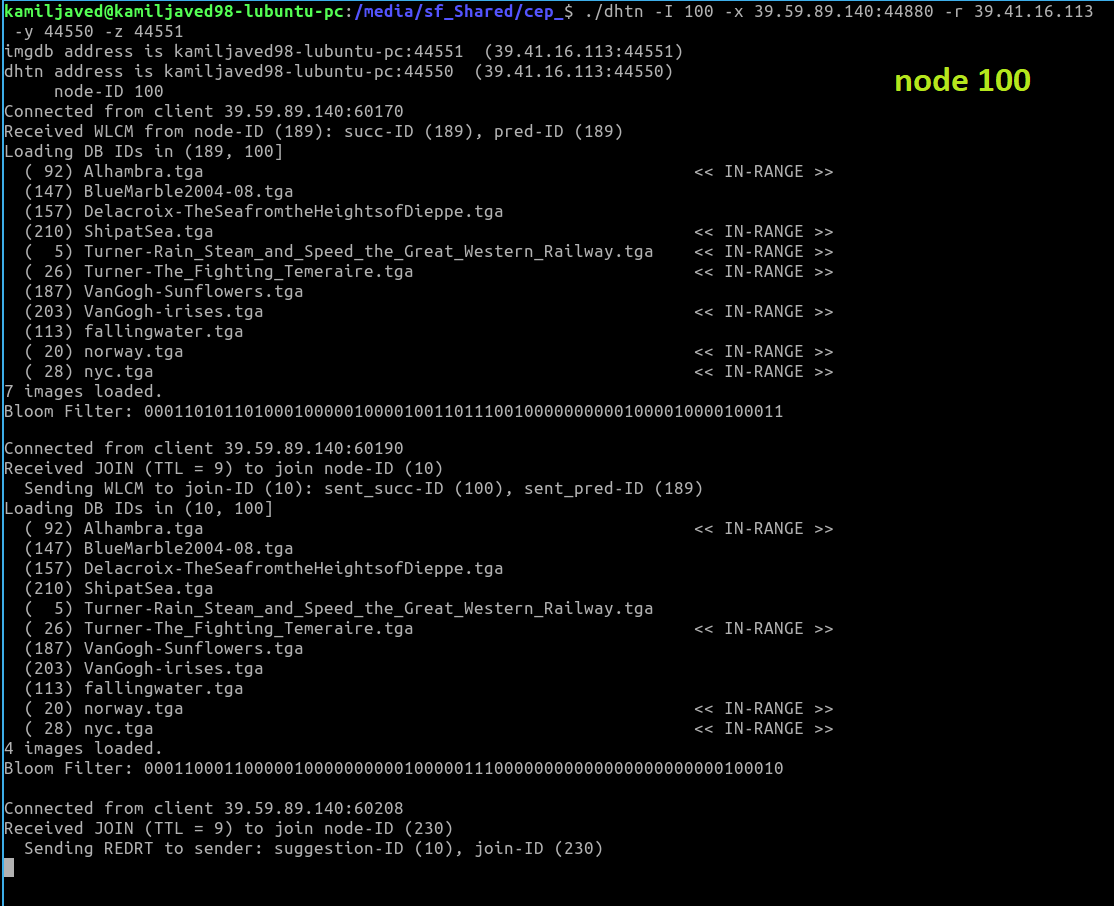
*The figures (terminal outputs) start at the next page.*

Terminal output of node 189 (*network1*) during all the nodes’ join process :



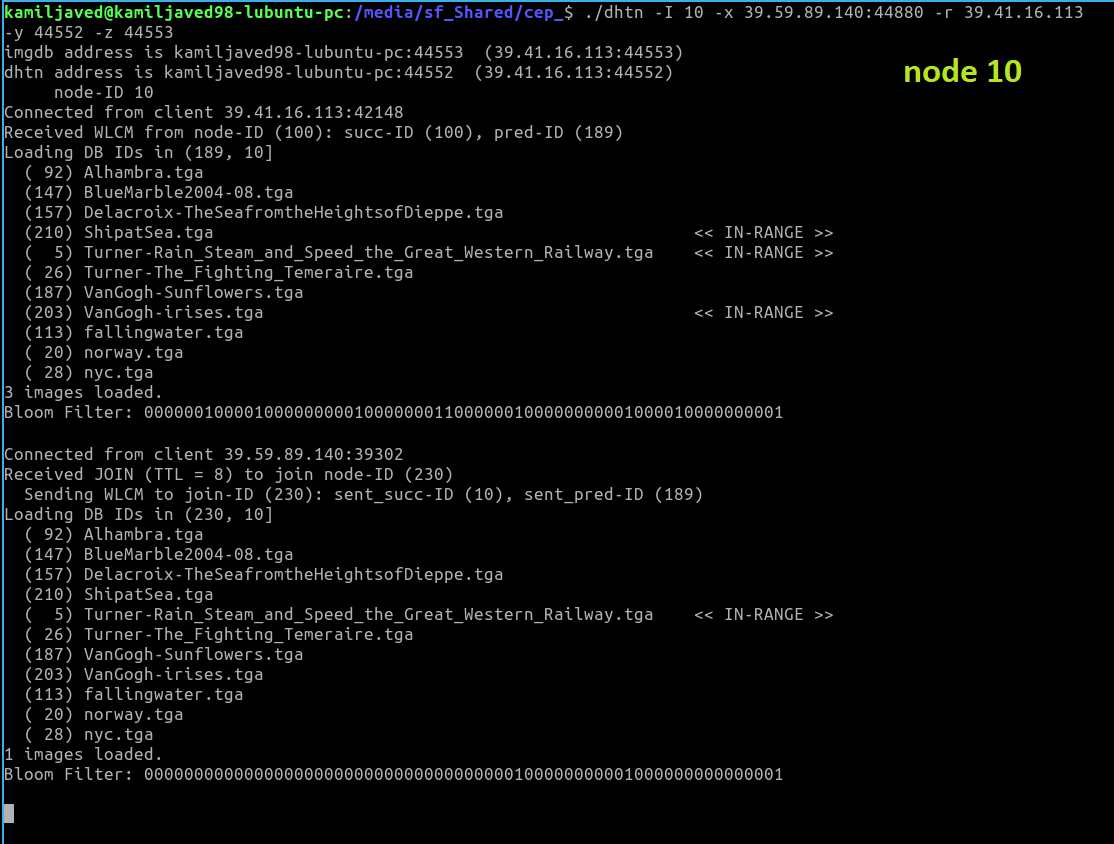
*Terminal output of next node is given on the next page.*

Terminal output of node 100 (*network2*) during all the nodes’ join process :



*Terminal output of next node is given on the next page.*

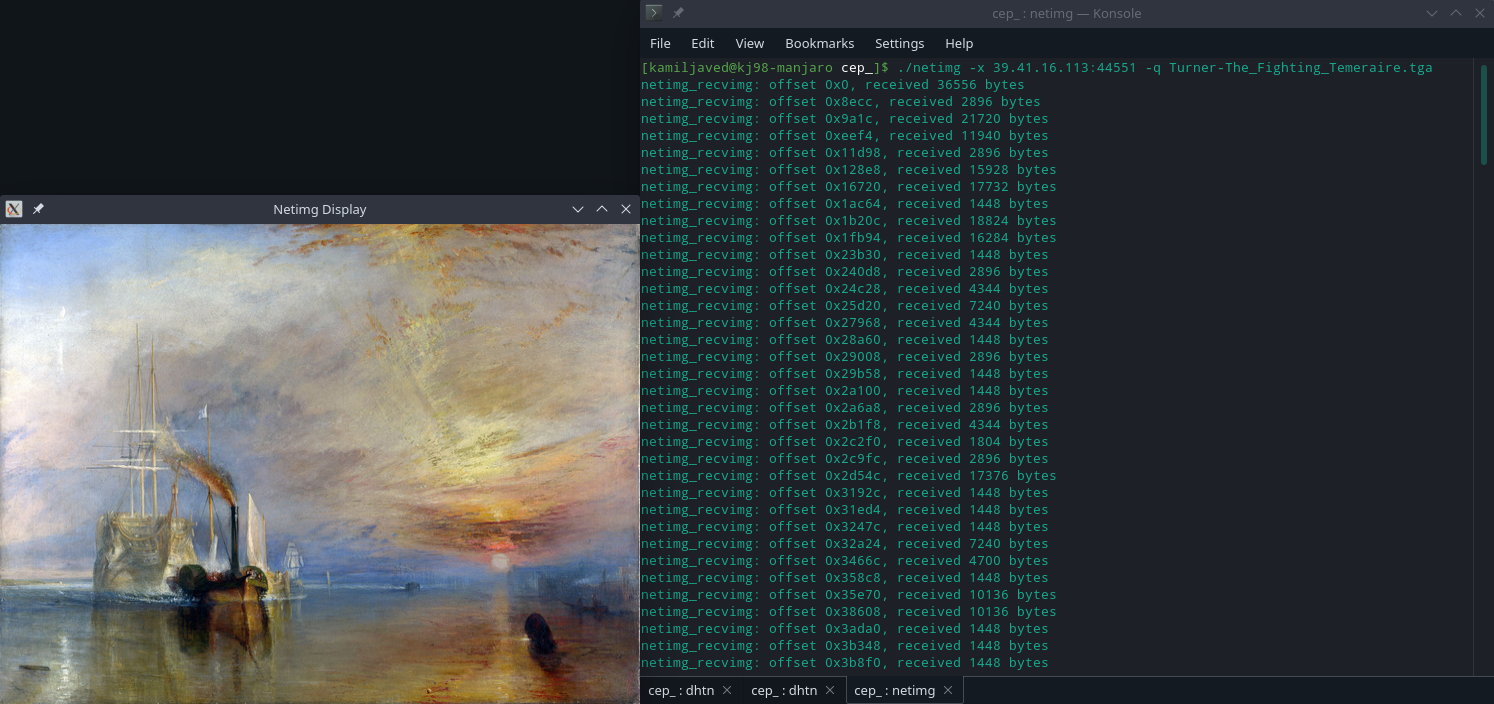
Terminal output of node 10 (*network2*) during all the nodes’ join process :



Terminal output of node 230 (*network1*) during all the nodes’ join process :



Then, an image was queried by *netimg* running on *network1* to the node 100 running on *network2* (across internet). This image was found with node 100, and hence its *imgdb* client returned the image.



Node 100 (the queried server) terminal output:



Similarly, image with ID 92 was queried using *netimg* (on *network1*) from node 230 running on *network1*, which was forwarded across the DHT (and eventually) over to node 100 (authoritative for ID 92 in this DHT) on *network2*, and node 100 replied with a RPLY message to node 230, permitting node 230 to cache and transfer the image to the querying *netimg* client.