ARIZONA STATE UNIVERSITY

CSE 434, SLN 10999 — Computer Networks — Spring 2024

Instructor: Dr. Violet R. Syrotiuk

Socket Programming Project

Available Sunday 02/04/2024; Milestone due Sunday 02/18/2024; Full project due Sunday 03/17/2024

The purpose of this project is to implement your own application program in which processes communicate using sockets to first build a distributed hash table (DHT) and then process queries using it.

- You may write your code in `C/C++`, in `Java`, or in `Python`; no other programming languages are permitted. Each of these languages has a socket programming library that you **must** use for communication.
- This project may be completed individually or in a group of size at most two. Whatever your choice, you **must** join a `Socket Group` under the `People` tab on Canvas before the milestone deadline of Sunday, 02/18/2024. This group can be the same as or different from the group used in Lab 1.
- Each group **must** restrict its use of port numbers to prevent the possibility of application programs from interfering with each other. As described in §3, port numbers are dependent on you group number.
- You **must** use a version control system as you develop your solution to this project, *e.g.*, GitHub or similar. Your code repository **must** be *private* to prevent anyone from plagiarizing your work. It is expected that you will create this repository by 02/11/2024 and commit changes to it on a regular basis.

1 DHT: A Distributed Hash Table with Hot Potato Query Processing

In a *distributed hash table* (DHT), the contents of a hash table are distributed across a number of hosts in a network rather than storing the entire table at a single host. In this project, a *ring* topology will be used for management of the DHT, while query processing will be done using a *hot potato* style of protocol. If you are interested, you can learn about more complex topologies used in real DHT implementations such Chord [2], Kademlia [1], and Tapestry [3].

The architecture of the DHT application is illustrated in Figure 1. It shows a DHT manager maintaining state information about the system. In this example, five peers are organized in a logical ring topology. Together they implement the DHT with each storing a subset of the records of the DHT in a local hash table. All management of the

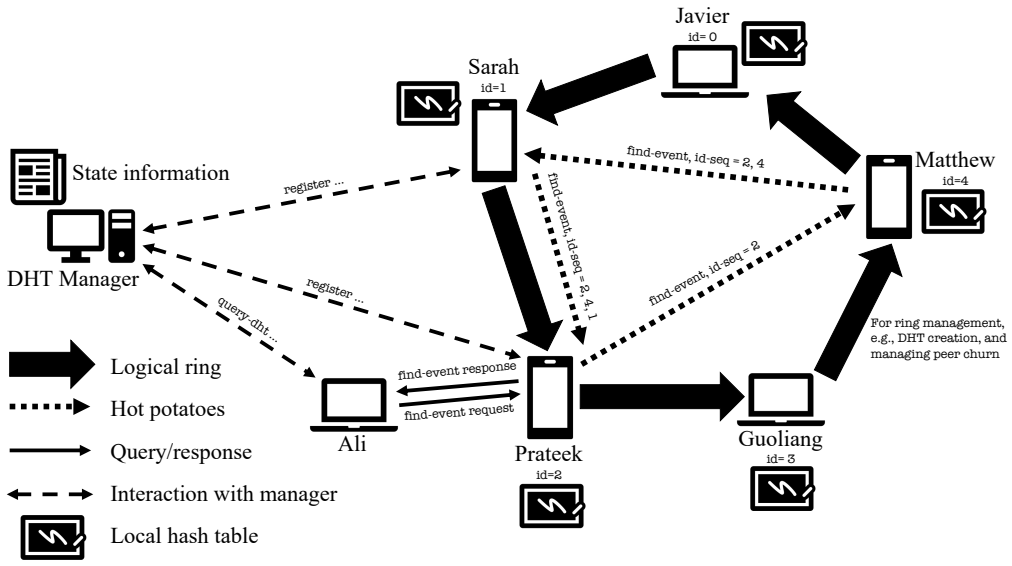

Figure 1: Architecture of the DHT application.

DHT is performed using the ring, including creation of the DHT, and handling churn of peers managing it. One peer is shown querying the DHT. A query may result in messages among peers in the DHT using a hot potato style of protocol to respond to the query. To simplify the application, any request to query the DHT while it is under construction or modification is denied.

This socket programming project involves the design and implementation of two programs:

1. The first program implements an "always-on" manager to support management of the peers implementing the DHT, among other functionality. Your manager should read one command line parameter, an integer giving the port number at which the server listens. The messages to be supported by the manager are described in §1.1.
2. The second program implements the peer protocol that supports both construction and management of the DHT, as well as the ability to query the DHT. Your peer should read two command line parameters, the first a string giving the IPv4 address of the manager in dotted decimal, and the second an integer port number at which the manager is listening. The messages to be supported by peers interacting with the manager are described in §1.1. Messages to be support by peers interacting with each other are described in §1.2.

1.1 The DHT Manager Protocol

The manager process is started on a host with a provided `port` number. Once started it runs in an infinite loop, listening to the given port for messages incoming from peers. As peers send messages, the manager maintains state of the DHT application, including the state of the peers. Peers that register with the manager may be in one of 3 states:

1. `Free`, a peer able to participate in any capacity,
2. `Leader`, a peer that leads the construction of the DHT, and
3. `InDHT`, a peer that is one of the members of the DHT.

Peers may transition between states as commands are processed.

All commands are read from `stdin` – no fancy UI is expected! A peer reads in a command and constructs a message to be sent to the manager or to another peer over a UDP socket. Messages to the manager always come in pairs; *i.e.*, after sending a message to the manager, the peer waits for a response before issuing its next command.

In the following, <> bracket parameters to a command, while the other strings are literals. The manager must support messages corresponding to the following commands from a peer:

1. `register` ⟨`peer-name`⟩ ⟨`IPv4-address`⟩ ⟨`m-port`⟩ ⟨`p-port`⟩, where `peer-name` is an alphabetic string of length at most 15 characters. This command registers a peer with the manager. All peers must be registered prior to issuing other any other commands to the manager.

   On receipt of a command to `register`, the manager stores the name, IPv4 address, and two ports associated with that peer in a state information base. The `m-port` is strictly for communication between the peer and manager, while the `p-port` is for communication between peers. The state of the peer is set to `Free`.

   If the parameters are unique, the manager responds with a return code of `SUCCESS`. Specifically, each `peer-name` may only be registered once. The `IPv4-address` of a peer need not be unique, *i.e.*, one or more processes may run on the same end host. However, the ports used for communication by each process must be unique.

   The manager takes no action and responds to the peer with a return code of `FAILURE` in the case of a duplicate registration, or some other problem.

2. `setup-dht` ⟨`peer-name`⟩ ⟨`n`⟩ ⟨`YYYY`⟩, where $\texttt{n} \geq 3$. This command initiates the construction of a DHT of size `n` using data from year `YYYY`, with `peer-name` as its leader. In this project, only one DHT may exist at any time.

   This command results in a return code of `FAILURE` if:

   - The `peer-name` is not registered.
   - `n` is not at least three.
   - Fewer than `n` users are registered with the manager.
   - A DHT has already been set up.

Otherwise, the manager sets the state of `peer-name` to `Leader`, and selects at random `n-1` `Free` users from those registered updating each one's state to `InDHT`. The manager returns a return code of `SUCCESS` and a list of `n` peers that together will construct the DHT. The `n` peers are given by 3-tuples consisting of the `peer-name`, `IPv4-address`, and `p-port`, with the 3-tuple of the leader given first (the other tuples can follow in any order). Receipt of `SUCCESS` at the leader involves several additional steps to accomplish the setup of the DHT, as described in §1.2.1.

After responding with `SUCCESS` to a `setup-dht`, the manager waits for `dht-complete`, returning `FAILURE` to any other incoming messages.

3. `dht-complete` ⟨`peer-name`⟩. Receipt of a `dht-complete` indicates that the leader has completed all the steps required to set up the DHT. If the `peer-name` is not the leader of the DHT then this command returns `FAILURE` Otherwise, the manager responds to the leader with `SUCCESS`. The manager may now process any other command (except `setup-dht`).

4. `query-dht` ⟨`peer-name`⟩. This command is used to initiate a query of the DHT. It returns `FAILURE` if the DHT set up has not been completed, the peer is not registered, or the peer is registered but not `Free`. Otherwise, the manager chooses at random one of the `n` peers maintaining the DHT and responds with a 3-tuple consisting of its `peer-name`, `IPv4-address`, and `p-port`. The return code is also set to `SUCCESS`. Receipt of `SUCCESS` to the peer making the `query-dht` involves several additional steps; see §1.2.2.

5. `leave-dht` ⟨`peer-name`⟩. A DHT is maintained by a set of processes that is dynamic. The `leave-dht` initiates the process of `peer-name` leaving the DHT. This command returns `FAILURE` if the peer is not one maintaining the DHT, or the DHT does not exist. Otherwise, the manager responds with `SUCCESS` and stores the `peer-name` making the request. The server waits for `dht-rebuilt`, returning `FAILURE` to any other incoming messages. Receipt of `SUCCESS` at the peer involves several steps to rebuild the DHT; see §1.2.3.

6. `join-dht` ⟨`peer-name`⟩. This command initiates the process of `peer-name` joining the DHT. This command returns `FAILURE` if the peer is not `Free`, or the DHT does not exist. Otherwise, the manager responds with `SUCCESS` and stores the `peer-name` making the request. The server waits for `dht-rebuilt`, returning `FAILURE` to any other incoming messages. Receipt of `SUCCESS` at the peer involves several steps to rebuild the DHT; see §1.2.4.

7. `dht-rebuilt` ⟨`peer-name`⟩ ⟨`new-leader`⟩, where `new-leader` is the name of the leader of the rebuilt DHT. Receipt of `dht-rebuilt` indicates that all the steps of managing the peer churn have been completed. If the `peer-name` is not the same as that initiating the `leave-dht`, or `join-dht`, then return `FAILURE`. Otherwise, the manager responds to the peer with `SUCCESS` and sets the state of `peer-name`, as appropriate. Rebuilding the DHT may require the assignment of a new leader; set states accordingly.

8. `deregister` ⟨`peer-name`⟩. This command removes the state of a `Free` peer from the state information base, allowing it to terminate. This command returns `FAILURE` if the peer state is `InDHT`. Otherwise, the peer's state information is removed, and the manager responds with `SUCCESS`. The user process then exits the application, *i.e.*, it terminates.

9. `teardown-dht` ⟨`peer-name`⟩, where `peer-name` is the leader of the DHT. This command initiates the deletion of the DHT. This command returns `FAILURE` if the peer is not the leader of the DHT. Otherwise, the manager responds with `SUCCESS`. Receipt of `SUCCESS` at the peer involves several steps to delete the DHT; see §1.2.5. The manager waits for `teardown-complete`, returning `FAILURE` to any other incoming messages.

10. `teardown-complete` ⟨`peer-name`⟩, where `peer-name` is the leader of the DHT. This command indicates that the DHT has been deleted. The manager returns `FAILURE` if the peer is not the leader of the DHT. Otherwise, the manager changes the state of each peer involved in maintaining the DHT to `Free`, and responds to the with `SUCCESS`.

1.2 The DHT Peer-to-Peer (P2P) Protocol

1.2.1 Creation of the DHT, `setup-dht`

The creation of a DHT is initiated by a `setup-dht` command sent by a peer to the manager, to create a DHT maintained by `n` peers in total. On receipt of `SUCCESS`, `n` 3-tuples consisting of peer name, IPv4 address, and peer port number are also included; see Table 1. The first 3-tuple is that of the process that sent the `setup-dht` command. This process serves as the leader of the DHT, assigning itself an identifier of zero. In the example in Figure 1, Javier is the leader.

Table 1: The `n` 3-tuples returned in a successful `setup-dht` command.

$\texttt{peer}_0$	$\texttt{IP-addr}_0$	$\texttt{p-port}_0$	← this is the leader's tuple
$\texttt{peer}_1$	$\texttt{IP-addr}_1$	$\texttt{p-port}_1$	
⋮	⋮	⋮	
$\texttt{peer}_{n-1}$	$\texttt{IP-addr}_{n-1}$	$\texttt{p-port}_{n-1}$	

The leader then follows these steps to set up the DHT:

1. **Assign identifiers and neighbours.** First, a logical ring among the `n` processes must be set up. The processes will be assigned identifiers $0, 1, \ldots, \texttt{n} - 1$. The leader has identifier 0. For $i = 1, \ldots, \texttt{n} - 1$:

   (a) The leader sends a command `set-id` to $\texttt{peer}_i$ at $\texttt{IP-addr}_i$, $\texttt{p-port}_i$. It also sends identifier `i`, the ring size `n`, and the 3-tuples from Table 1.
   (b) On receipt of a `set-id`, $\texttt{peer}_i$ sets its identifier to `i` and the ring size to `n`. It stores the 3-tuple of $\texttt{peer}_{(i+1) \bmod \texttt{n}}$ to use as the address of its right neighbour, *i.e.*, the next process on the cycle.

   After the assignment of identifiers, all management messages **must** always travel around the logical ring in one direction, from the process with identifier $0 \to 1 \to 2 \to \ldots \to \texttt{n} - 1$ and back to 0.

2. **Construct the local DHTs.** Now, the leader populates the DHT with data. The dataset is storm data provided by the U.S. National Weather Service (NWS). It contains a listing by state of weather phenomena. Data is available from 1950 to the present.

   For each year `YYYY`, $1950 \leq \texttt{YYYY} \leq 2019$, the input file is named `details-YYYY.csv`, and is a *comma separated value* (CSV) file containing the storm event data. Each line of the file `details-YYYY.csv` contains details of a storm event described by the following 14 fields, in order. An exception is the first line of the file which contains the field names and should be skipped.

   (a) `event_id`: An identifier assigned by the NWS for a specific storm event. Example: 383097.
   (b) `state`: The state name, spelled in all capital letters, where the event occurred. Example: GEORGIA.
   (c) `year`: The four digit year for the event in this record. Example: 2000.
   (d) `month_name`: Name of the month for the event in this record spelled out. Example: January.
   (e) `event_type`: The event types permitted are listed in Table 2, spelled out.
   (f) `cz_type`**:** Indicates whether the event happened in a county/parish (C), zone (Z), or marine (M).
   (g) `cz_name`: County/Parish, Zone or Marine name assigned to the county or zone. Example: AIKEN.
   (h) `injuries_direct`: The number of injuries directly related to the weather event. Examples: 0, 56.
   (i) `injuries_indirect`: The number of injuries indirectly related to the weather event. Examples: 0, 87.
   (j) `deaths_direct`: The number of deaths directly related to the weather event. Examples: 0, 23.
   (k) `deaths_indirect`: The number of deaths indirectly related to the weather event. Examples: 0, 4, 6.
   (l) `damage_property`: The estimated amount of damage to property incurred by the weather event. Examples: 10.00K, 0.00K, 10.00M.
   (m) `damage_crops`: The estimated amount of damage to crops incurred by the weather event.
   (n) `tor_f_scale`: Enhanced Fujita Scale describes the strength of the tornado based on the amount and type of damage caused by the tornado. Examples: EF0, EF1, EF2, EF3, EF4, EF5.

Table 2: Valid Event Types

Event Name	Designator	Event Name	Designator
Astronomical Low Tide	Z	Hurricane (Typhoon)	Z
Avalanche	Z	Ice Storm	Z
Blizzard	Z	Lake-Effect Snow	Z
Coastal Flood	Z	Lakeshore Flood	Z
Cold/Wind Chill	Z	Lightning	C
Debris Flow	C	Marine Hail	M
Dense Fog	Z	Marine High Wind	M
Dense Smoke	Z	Marine Strong Wind	M
Drought	Z	Marine Thunderstorm Wind	M
Dust Devil	C	Rip Current	Z
Dust Storm	Z	Seiche	Z
Excessive Heat	Z	Sleet	Z
Extreme Cold/Wind Chill	Z	Storm Surge/Tide	Z
Flash Flood	C	Strong Wind	Z
Flood	C	Thunderstorm Wind	C
Frost/Freeze	Z	Tornado	C
Funnel Cloud	C	Tropical Depression	Z
Freezing Fog	Z	Tropical Storm	Z
Hail	C	Tsunami	Z
Heat	Z	Volcanic Ash	Z
Heavy Rain	C	Waterspout	M
Heavy Snow	Z	Wildfire	Z
High Surf	Z	Winter Storm	Z
High Wind	Z	Winter Weather	Z

Compute the number of storm events, ℓ, in the `details-YYYY.csv` file, *i.e.*, , this is one less than the number of lines in the file. For $i = 1, \ldots, \ell$, the leader:

(a) Reads storm event i from the dataset into a record. Each record will be stored in **one** local hash table.

(b) Determines the identifier of the peer that will store the record. To do so, it computes two hash functions. The first hash function computes a position, `pos`, as the `event_id` field modulo the hash table size. The hash table size, `s`, is the first prime number larger than $2 \times \ell$. The second hash function is the position modulo the ring size `n`.

$$\begin{aligned} \texttt{pos} &= \texttt{event\_id} \bmod s \\ \texttt{id} &= \texttt{pos} \bmod \texttt{n} \end{aligned}$$

Here, $0 \leq \texttt{id} \leq \texttt{n} - 1$, is the identifier of the node in the ring that is to store the record, whereas `pos` gives the position in the local hash table of node *id* at which to store the record.

If the `id` is identifier of the current node, then it stores the record in its local hash table. Otherwise, the leader sends a `store` command to its right neighbour on the ring, along with the record. The record is forwarded along the ring from the leader to node whose identifier equals *id*, where it is stored in the local hash table. In a correct implementation, a `store` command does not return to the leader.

Do not have the leader send the `store` command directly to the node *id*. In DHTs the topology of the DHT, here a ring, **must** be used for management functions.

3. **Print configuration and signal completion of DHT set up.** The leader outputs the number of records stored at each node in the ring, and sends a message for the `dht-complete` command to the manager.

1.2.2 Querying the DHT, `query-dht`

In response to a successful `query-dht` command, the manager, it returns a 3-tuple `peer-name`, `IP-v4-address`, and `p-port` of any the one of the processes involved in maintaining the DHT. That is, the query process of the sending peer `S` **can start at any node in the ring**.

The sending peer `S` issues a `find-event <event_id>` query to `peer-name` to search the DHT for the storm event with identifier `event_id`. The `find-event` query should include the 3-tuple of `S`.

1. On receipt of the `find-event` at `peer-name`, it computes `pos` and `id` as described in §1.2.1.
2. If the `id` equals the identifier of `peer-name`, it examines position `pos` in its local hash table to see if it holds the record with the given `event_id`. If so, it returns `SUCCESS`, including the entire record associated with `event_id`, and `id-seq` equal to `id`.
3. If the `id` does not equal the identifier of `peer-name`, then the `find-event` message is forwarded using a hot potato protocol. In a hot potato protocol, the identifier of the node whose local hash table is searched next is selected at random.
   (a) Let $\mathtt{I} = \{0, 1, \ldots, \mathtt{n}-1\} \setminus \mathtt{id}$. That is, `I` is the set of all identifiers from zero to $\mathtt{n}-1$ except `id`.
   (b) Initialize `id-seq` to `id`. The `id-seq` records the order that nodes in the ring are visited.
   (c) Let let $\mathtt{next} = \mathtt{random(I)}$. The `find-event` message is sent to *next*, updating the set $\mathtt{I} = \mathtt{I} \setminus \mathtt{next}$, and appending `next` to `id-seq`. Node `next` examines position *pos* in its local hash table for the event.
4. The message is forwarded until either the search is successful or it is not. If the event with the given `event_id` is found, a return code of `SUCCESS` is sent to `S`, including the associated record, and `id-seq`. The fields of the storm event, with each field labelled, should be output on a separate line at `S`. In addition, the `id-seq` giving the sequence of nodes probed to find the record should also be output.

   If the event with the given `event_id` is not found, a return code of `FAILURE` is sent to `S`, which subsequently prints: `"Storm event <event_id> not found in the DHT."`

1.2.3 Leaving the DHT, `leave-dht`

DHTs are maintained by a dynamic set of processes so existing implementations of DHTs allow new nodes to join or leave the maintenance of the DHT. Suppose that a `leave-dht` command is issued to the manager by peer `u` with identifier i. This requires the DHT to be rebuilt with $\mathtt{n}-1$ nodes. One way to accomplish this is as follows:

1. Peer `u` initiates a `teardown` of the DHT (see step 1 in §1.2.5). When the `teardown` propagates back to `u`, it deletes its own local hash table.
2. A renumbering of ring identifiers is initiated by `u`. A `reset-id` command is sent to its right neighbour setting its identifier to zero, and the ring size to $\mathtt{n}-1$. Peer `u` is removed from the set of tuples. The `reset-id` should be propagated around the ring, incrementing the node identifier and resetting the ring size as it goes.

   When a `reset-id` is received by `u`, it knows the identifiers of the nodes in the ring have been renumbered, with its right neighbour becoming the leader of the new smaller ring of size $\mathtt{n}-1$.
3. Peer `u` sends a command `rebuild-dht` to its right neighbour, *i.e.*, the new leader of the smaller ring. Upon receipt of the `rebuild-dht`, the new leader follows step 2 of §1.2.1 to construct the local DHTs in the new ring of size $\mathtt{n}-1$, signalling `u` on completion.
4. Peer `u` sends a message for the `dht-rebuilt` command to the manager specifying its right neighbour as the `new-leader`.

1.2.4 Joining the DHT, `join-dht`

Design the protocol for a peer to join the DHT. The last step must include the peer sending a message for the `dht-rebuilt` command to the manager specifying the `new-leader`.

1.2.5 Deletion of the DHT, `teardown-dht`

The command `tearddown-dht` deletes the DHT. This is accomplished by the following steps:

1. The leader send a `teardown` command to its right neighbour. This causes each node to delete its local hash table as the `teardown` propagates around the ring.

2. On receipt of the `teardown` at the leader, it deletes its local hash table and then sends a message for the `teardown-complete` command to the server.

Be sure to make the implementation of `teardown` generic so that it can be used as part of the processing for the `leave-dht` command; see step 1 of §1.2.3.

2 Implementation Suggestions

2.1 Managing Sockets

For the `n` 3-tuples returned by the `setup-dht`, you should establish an array of sockets indexed by node `id` on the ring. Then node `i`'s right neighbour on the ring is found by writing to the socket at position $\texttt{i} + 1 \bmod \texttt{n}$.

You may consider using a different thread for handling socket associated with the `m-port` and `p-port` of each process. Alternatively a single thread may loop, checking each socket one at a time to see if a message has arrived for the process to handle. If you use a single thread, you must be aware that by default the function `recvfrom()` in C++ is blocking. This means that when a process issues a `recvfrom()` that cannot be completed immediately (because there is no packet), the process is put to sleep waiting for a packet to arrive at the socket. Therefore, a call to `recvfrom()` will return immediately only if a packet is available on the socket. This may not be what you want.

You can change `recvfrom()` to be non-blocking, *i.e.*, it will return immediately even if there is no packet. This can be done in C++ by setting the `flags` argument of `recvfrom()` or by using the function `fcntl()`. See the `man` pages for `recvfrom()` and `fcntl()` for details.

2.2 Defining Message Exchanges and Message Format

Sections §1.1 and §1.2 have described the order of many message exchanges, as well as the actions taken on the transmission and/or receipt of a message. As part of this project, you may need to add details for the `setup-dht`, `query-dht`, `leave-dht`, `join-dht`, and the `teardown-dht` commands.

In addition, you need to define the format of all messages used in peer-manager and in peer-to-peer communication. This could be achieved by defining a structure with all the fields required by the command. For example, you could define the name of the command as an integer field and interpret it accordingly. Alternatively, a message can be a string, with concatenated fields separated by a delimiter. Indeed, any choice is fine so long as you are able to extract the fields of a message and interpret them.

You may want to define an upper bound on the size a local hash table as the first prime number larger than $2 \times \ell$ (see §1.2.1), and *reasonable* upper bounds on the total number of registered peers that your application supports, among others. It may also useful to define meaningful return codes to further differentiate `SUCCESS` and `FAILURE`.

3 Port Numbers

RFC 1700 contains the list of port number assignments from the Internet Assigned Numbers Authority (IANA). The port numbers are divided into three ranges:

- *Well-known ports:* 0 through 1023. These port numbers are controlled and assigned by IANA.
- *Registered ports:* 1024 through 49151. The upper limit of 49151 for these ports is new; RFC 1700 lists the upper range as 65535. These port numbers are not controlled by the IANA.
- *Dynamic* or *private ports:* 49152 through 65535. The IANA dictates nothing about these ports. These are the ephemeral ports.

In this project, each group $G \geq 1$ is assigned a set of 500 unique port numbers to use in the following range. If $G \bmod 2 = 0$, *i.e.*, your group number is even, then use the range: $\left[\left(\frac{G}{2} \times 1000\right) + 1000, \left(\frac{G}{2} \times 1000\right) + 1499\right]$. If $G \bmod 2 = 1$, *i.e.*, your group number is odd, then use the range: $\left[\left(\left\lceil \frac{G}{2} \right\rceil \times 1000\right) + 500, \left(\left\lceil \frac{G}{2} \right\rceil \times 1000\right) + 999\right]$. That is, group 1 has range $[1500, 1999]$, group 2 has range $[2000, 2499]$, group 3 has range $[2500, 2999]$, and so on.

Do not use port numbers outside your assigned range, as otherwise you may send messages to another group's process by accident and it is unlikely it will be able to interpret it correctly, causing spurious crashes.

4 Submission Requirements for the Milestone and Full Project Deadlines

Submit electronically before 11:59pm on the deadline date a zip file named `Groupx.zip` where `x` is your group number. Do not use any other archiving program except `zip`.

1. The milestone is due on Sunday, 02/18/2024. See §4.1 for requirements.
2. The full project is due on Sunday, 03/17/2024. See §4.2 for requirements.

It is your responsibility to submit your project well before the time deadline!!! Late projects are not accepted. Do not expect the clock on your machine to be synchronized with the one on Canvas!

An unlimited number of submissions are allowed. The last submission will be graded.

4.1 Submission Requirements for the Milestone

For the milestone deadline, you are to implement the following commands to the manager: `register`, `setup-dht`, and `dht-complete`. This also involves implementation of commands that may be issued among peers associated with these commands; see, *e.g.*, §1.2.1.

For the milestone, your `zip` file must contain:

1. **Design document in PDF format (50%).** Describe the design of your DHT application program.
   (a) Include a description of your message format for each command implemented for the milestone.
   (b) Include a time-space diagram for each command implemented to illustrate the order of messages exchanged between communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event. Use a software tool of your choice, *e.g.*, PowerPoint, to draw these diagrams.
   (c) Describe your choice of data structures used, implementation considerations, and other design decisions.
   (d) Include a snapshot showing commits made in your choice of version control system.
   (e) Provide a *a link to your video demo* and ensure that the link is accessible to our graders. In addition, give a list of timestamps in your video at which each step 3(a)-3(d) is demonstrated.
2. **Code and documentation (25%).** Submit your well-documented source code implementing the milestone of your DHT application.
3. **Video demo (25%).** Upload a video of length at most 7 minutes to YouTube with no splicing or edits, with audio accompaniment. This video must be uploaded and timestamped *before* the milestone submission deadline.

   The video demo of your DHT application for the milestone must include:

   (a) Compile your `manager` and `peer` programs (if applicable).
   (b) Run the freshly compiled programs on at least two distinct end-hosts.
   (c) Start your `manager` program. Then start three `peer` processes that each `register` with the `manager`.
   (d) Have one `peer` issue a `setup-dht` command to build a DHT of size `n` $= 3$ using the `YYYY` $= 1950$ dataset. This should output the number of records stored at each peer in the ring and finish by sending a `dht-complete` to the `manager`.

Graceful termination of your application is not required at this time.

For the end-hosts, consider using `general{3|4|5}.asu.edu`, the machines on the racks in BYENG 217, or installing your application on VMs on a LAN you configure in CloudLab, or using any other end-hosts available to you for the demo.

Your video will require at least four (4) windows open: one for the `manager`, and one for each `peer`. Ensure that the font size in each window is large enough to read!

Be sure that the output of your commands are a well-labelled trace of the messages transmitted and received between processes so that it is easy to follow what is happening in your DHT application program.

4.2 Submission Requirements for the Full Project

For the full project deadline, you are to implement the all commands to the manager listed in §1.1. This also involves implementation of all commands issued between peers that are associated with these commands as described in §1.2.

For the full project submission, your `zip` file must contain:

1. **Design document in PDF format (30%).** Extend the design document for the milestone of your DHT application to include details for the remaining commands implemented for the full project. Provide all items listed in step 1 of §4.1, except that the list of timestamps in the video must be for each step 3(a)-3(i) listed below.
2. **Code and documentation (20%).** Submit well-documented source code implementing your DHT application.
3. **Video demo (50%).** Upload a video of target length at most 15 minutes to YouTube with no splicing or edits, with audio accompaniment. This video must be uploaded and timestamped *before* the full project deadline.

   The video demo of your DHT application for the full project must include:

   (a) Compile your `manager` and `peer` programs (if applicable).
   (b) Run the freshly compiled programs on at least four distinct end-hosts.
   (c) Start your `manager` program. Then start six `peer` processes that each `register` with the `manager`.
   (d) Select one `peer` to issue a `setup-dht` command to build a DHT of size `n` $= 5$ and `YYYY` $= 1996$.
   (e) Have the remaining `peer` issue a sequence of `query-dht` commands, with `event_id` taken from the set $\{5536849, 2402920, 5539287, 55770111\}$.
   (f) Select one `peer` in the DHT issue a `leave-dht` command. Then issue at least one query by the `peer` that left the DHT (with any `event_id`).
   (g) Select one `peer` of the two outside the DHT to issue a `join-dht` command. Then issue at least one query by the remaining `peer` (with any `event_id`).
   (h) Have the leader of the DHT issue a `teardown-dht`.
   (i) Gracefully terminate of your application, *i.e.*, `de-register` all peers and have them exit. Of course, the `manager` process needs to be terminated explicitly.

   Ensure that the font size in each window you open is large enough to read!

   As before, the output of your commands must be a well-labelled trace the messages transmitted and received between processes so that it is easy to follow what is happening in your DHT application program.

References

[1] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *International Workshop on Peer-to-Peer Systems (IPTPS'02), LNCS Volume 2429*, pages 53–62, 2002.

[2] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[3] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications (JSAC)*, 22(1):41–53, 2004.