ARIZONA STATE UNIVERSITY

CSE 434, SLN 70569 — Computer Networks — Fall 2021

Instructor: Dr. Violet R. Syrotiuk

Socket Programming Project

Available Sunday 09/12/2021; group formation by Sunday, 09/19/2021
Milestone due Sunday 09/26/2021; full project due Sunday 10/17/2021

The purpose of this project is to implement your own application program in which processes communicate using sockets to maintain a distributed hash table (DHT) dynamically, and answer 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 people. Whatever your choice, you **must** join a *Socket Group* under the `People` tab on Canvas before Sunday, 02/09/2020. This group can be the same as or different from the group used in Lab #1. **Please join groups consecutively.**
- Each group **must** restrict its use of port numbers to prevent application programs from interfering with each other. As described in in §2, your 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 commit changes to your repository on a regular basis.

The rest of this project description is organized as follows. Following an overview of the DHT application and its architecture in §1, the requirements of the DHT's client-server protocol, and its peer-to-peer protocol are provided in §1.1 and §1.2, respectively. §3 describes the requirements for the milestone and full project deadlines.

1 DHT: A Distributed Hash Table with Ring Overlay

As the name suggests, in a *distributed hash table* (DHT), the contents of a hash table are distributed across a number of nodes in a network rather than storing the entire table at a single node. In this project, a *ring overlay* is imposed on the nodes implementing the DHT, *i.e.*, these nodes are logically organized in the topology of a cycle. If you are interested, you can learn about the more complex overlay topologies used in real DHT implementations such in Chord [3], Kademlia [2], and Tapestry [4].

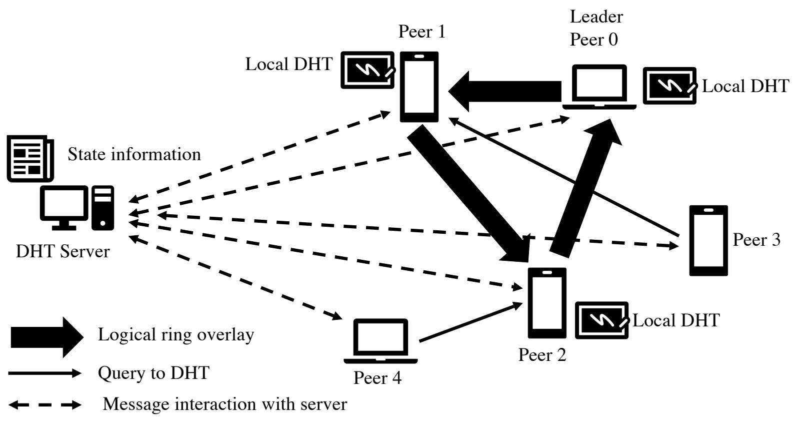

Figure 1: Architecture of the DHT application.

The architecture of the DHT application is illustrated in Figure 1. The DHT server maintains state information about the system. Peers interact with the server to establish the DHT. In this example three peers, 0, 1, and 2, are organized in a logical ring and together they implement the DHT with each storing a subset of the records of the hash table locally. After the DHT is constructed, other users, such as peers 3 and 4, may query the DHT. 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" server of a client-server protocol to support management of the peer processes implementing the DHT, among other functionality. Your server should read one integer command line parameter giving the port number (from your range of port numbers) at which the server listens. The messages to be supported by the server are described in §1.1.
2. The second program implements the client-side of the client-server protocol, as well as the peer-to-peer protocol to construct the DHT, and to query it. Your process should read at least two command line parameters, the first being a string representing the IPv4 address of the server in dotted decimal, and the second being an integer port number at which the server is listening. Depending on your design decisions, you may add additional command line parameters. The messages to be supported by a peer interacting with the server, and with other peers are described in §1.2.

1.1 The DHT Client-Server Protocol

Recall that a *protocol* defines the format and the order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event [1].

The server process is started on an end-host with an ⟨`IPv4-address`⟩ and given a ⟨`port`⟩ number. It runs in an infinite loop listening to the given port for messages incoming from a client/peer/user process. As peers send messages, the server maintains state about the DHT application. Peers that register with the server 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 transition between states as commands are processed. Peers forming the DHT may not query the DHT.

Commands are read from `stdin` at a client (no fancy UI is expected!). A client processes a command by constructing a message to be sent to the server or to another peer over a UDP socket. Messages to the server always come in pairs; *i.e.*, after sending a request to the server, the client waits for a response before issuing its next command. Messages sent among peers are described in §1.2.

The server must support messages corresponding to the following commands from a client:

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

   On receipt of a command to register, the server stores the name, IPv4 address, and one or more ports associated with that user in a state information base. The state of the user is initialized to `Free`.

   If the parameters are unique, the server responds to the client with a return code of `SUCCESS`. Specifically, each ⟨user-name⟩ may only be registered once. The ⟨IPv4-address⟩ of a user need not be unique, *i.e.*, one or more client/peer processes may run on the same end-host. However, the port(s) used for communication by each process must be unique; see §1.1.1 for suggestions on implementation.

   The server takes no action and responds to the client with a return code of `FAILURE` indicating failure register the user due to a duplicate registration, or any other problem.

2. `setup-dht` ⟨`n`⟩ ⟨`user-name`⟩, where $n \geq 2$. This command initiates the construction of a DHT of size `n`, with `user-name` as its leader. In this project, only one DHT may exist at any time (though in principle this need not be the case).

This command results in a return code of FAILURE if:

- The user-name is not registered.
- n $\ngeq$ 2.
- There are not at least n users registered with the server.
- A DHT has already been setup.

Otherwise, the server sets the state of user-name to Leader, and selects at random n−1 Free users from those registered updating each one's state to InDHT. The server returns a return code of SUCCESS and a list of n users that together will construct the DHT. The n users are given by 3-tuples consisting of the user-name, IPv4-address, and port, with the 3-tuple of the leader given first (the other tuples can be given 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 server waits for dht-complete, returning FAILURE to all incoming requests until the DHT has been established.

3. dht-complete ⟨user-name⟩. Receipt of a dht-complete indicates that the leader has completed all the steps required to setup the DHT. If the user-name is not the leader of the DHT then this command returns FAILURE Otherwise, the server responds to the leader with SUCCESS. The server may now process any other command except setup-dht (because we assume at most one DHT exists in our system).

4. query-dht ⟨user-name⟩. This command is used to initiate a query of the DHT. It returns FAILURE if the DHT setup has not been completed, the user is not registered, or the user is registered but is not Free. Otherwise the server chooses, at random, one of the n users maintaining the DHT and returns a 3-tuple corresponding to its user-name, IPv4-address, and port to the user wishing to initiate a query the DHT. The return code is also set to SUCCESS. Receipt of SUCCESS at the client involves several additional steps to process the query, as described in §1.2.2.

5. leave-dht ⟨user-name⟩. In general, a DHT is maintained by a set of processes that is dynamic. The leave-dht initiates the steps of user-name leaving the DHT, *i.e.*, the size of the DHT decreases from n to $(n-1) \geq 1$ processes. This command returns FAILURE if the DHT does not exist, or user-name is not involved in maintaining the DHT, *i.e.*, its state is not either Leader or InDHT. Otherwise, the server responds to the user with SUCCESS and stores the user-name making the request. The server waits for dht-rebuilt, returning FAILURE to all other incoming requests until the DHT has stabilized. Receipt of SUCCESS at the client involves several steps to rebuild the DHT, as described in §1.2.3.

6. dht-rebuilt ⟨user-name⟩ ⟨new-leader⟩, where new-leader is the user name of the leader of the rebuilt DHT. Receipt of dht-rebuilt indicates that all the steps of removing user-name from maintaining the DHT have been completed. If the user-name is not the same as that initiating the leave-dht then return FAILURE. Otherwise, the server responds to the user with SUCCESS and sets the state of user-name to Free. Rebuilding the DHT may have required the assignment of a new leader. If the Leader of the DHT is not the user new-leader, then the state of the old leader is set to InDHT, the state of user new-leader is set to Leader, and the state of user-name is set to Free. The server may now resume processing any other requests (except setup-dht).

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

8. Design the order of message exchanges, as well as the actions taken on the transmission and/or receipt of a message for a command join-dht that increases the size of the size of the DHT from n to $n+1$ processes, distributing the hash table records among one more process. You should be able to adapt the instructions given in §1.2.3 for leaving the DHT to joining it, and reuse the dht-rebuilt command.

9. `teardown-dht` ⟨`user-name`⟩, where `user-name` is the leader of the DHT. This command initiates the deletion of the DHT. This command returns `FAILURE` if the user is not the leader of the DHT. Otherwise, the server responds to the leader with `SUCCESS`. Receipt of `SUCCESS` at the leader involves several steps to delete the DHT, as described in §1.2.4. The server waits for `teardown-complete`, returning `FAILURE` to all other incoming queries until the DHT has been deleted.

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

1.1.1 Implementation Suggestions

A peer process communicates with the server as a client, and it may either query a peer in the DHT, or participate in maintaining the DHT in which case it sends messages to its right neighbour in the ring overlay and receives from its left neighbour. You may choose to set up a separate socket for each such communication. In this case, you must use a different port for each socket, so you should register multiple ports with the server for each user. Rather than registering a 3-tuple, you may decide to register two additional ports, *e.g.*,

$$\texttt{register}\ \langle\texttt{user-name}\rangle\ \langle\texttt{IPv4-address}\rangle\ \langle\texttt{port}_{\text{from}-\text{left}}\rangle\ \langle\texttt{port}_{\text{to}-\text{right}}\rangle\ \langle\texttt{port}_{\text{query}}\rangle$$

If you set up multiple sockets, you may consider using a different thread for handling each one. 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()` is blocking. This means that when a process issues a `recvfrom()` that cannot be completed immediately (because there is no packet to read), 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 the behaviour 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 by setting the `flags` argument of `recvfrom()` or by using the function `fcntl()`. See the `man` pages for `recvfrom()` and `fcntl()` for details; be sure to pay attention to the return codes.

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

A peer-to-peer (P2P) process can be a client of the server and also interact with other P2P nodes. Many requests sent to the server involve several additional steps taken by the process initiating the request after receipt of `SUCCESS` from the server. Specifically, each of `setup-dht`, `query-dht`, `leave-dht`, `teardown-dht`, and perhaps your design of the `join-dht`, involve additional steps, as described in the following sections.

1.2.1 Creation of the DHT

The creation of a DHT is initiated by a `setup-dht` command sent by the leader process, to be maintained by `n` processes in total. On receipt of `SUCCESS`, `n` 3-tuples are also included as Table 1 shows. The first 3-tuple is the user name, IPv4 address, and port number of the process that sent the `setup-dht` command. This process serves as the leader of the DHT, assigning itself an identifier of zero. If you choose to use multiple sockets for communication (see §1.1.1) then rather than a 3-tuple, you may return a larger tuple.

The leader then follows these steps to setup the DHT:

1. **Assign identifiers and neighbours.** First, a logical ring among the `n` processes must be set up. The processes are assigned identifiers $0, 1, \ldots, \texttt{n} - 1$ because these will be used by the hash function to determine in which local DHT to store a record. The leader has identifier 0. For $i = 1, \ldots, \texttt{n} - 1$:
   (a) The leader sends a command `set-id` to $\texttt{user}_i$ at $\texttt{IP-addr}_i$, $\texttt{port}_i$. It also sends identifier `i`, the ring size `n`, and the two 3-tuples $(i - 1) \bmod \texttt{n}$ and $(i + 1) \bmod \texttt{n}$ from Table 1.

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

$\texttt{user}_0$	$\texttt{IP-addr}_0$	$\texttt{port}_0$
$\texttt{user}_1$	$\texttt{IP-addr}_1$	$\texttt{port}_1$
$\texttt{user}_2$	$\texttt{IP-addr}_2$	$\texttt{port}_2$
$\vdots$	$\vdots$	$\vdots$
$\texttt{user}_{n-1}$	$\texttt{IP-addr}_{n-1}$	$\texttt{port}_{n-1}$

(b) On receipt of a `set-id`, $\texttt{user}_i$ sets its identifier to `i` and the ring size to `n`. It stores the 3-tuple of $\texttt{user}_{(i-1) \bmod \texttt{n}}$ to use as the address and port of its left neighbour. It stores the 3-tuple of $\texttt{user}_{(i+1) \bmod \texttt{n}}$ to use as the address and port of its right neighbour.

Messages always travel around the logical ring in one direction, received from the left neighbour and forwarded to the right neighbour. After the end of this step, each of the processes has a unique identifier and knows the address of its left and right neighbours.

2. **Construct the local DHTs.** Now, the leader populates the DHT. The dataset consists of a set of 241 records in a CSV file named `StatsCountry.csv`. Each record consists of 9 fields related to countries of the world: Country Code, Short Name, Table Name, Long Name, 2-alpha Code, Currency Unit, Region, WB-2 Code, and Latest Population Census. The first line of the file consists of the column headers and is not part of the data. For $\ell = 2, \ldots, 241$:

   (a) The leader reads line ℓ from the dataset into a record; you will need to define the record structure.

   (b) The leader computes the hash function using the Long Name as the key in two steps: First, as the sum of the ASCII value of each character in the Long Name field of record ℓ, modulo the local hash table size of 353 and, secondly, modulo the logical ring size `n`.

$$\begin{aligned} \mathit{pos} &= \left(\sum_{k=1}^{len(LongName)} ASCII(LongName[k]) \right) \bmod 353 \\ \mathit{id} &= \mathit{pos} \bmod \texttt{n} \end{aligned}$$

   The second modulus yields a value *id* that is the identifer, $0 \leq id \leq \texttt{n} - 1$, of the node in the logical ring that must store the record (each record is stored in exactly one node in the DHT). The first modulus gives the position in the local hash table of node *id* on the logical ring at which to store the record.

   For example, the sum of the ASCII values of the characters in the Long Name of the country `Aruba` is: $(65 + 114 + 117 + 98 + 97) = 491$. Taking this sum modulo the local hash table size 353 gives, $491 \bmod 353 = 138$, the position to store the record in the local hash table. But the local hash table is determined by the second modulus, *i.e.*, modulo the ring size.

   If the hash function returns an *id* equal to the identifier of node i, $0 \leq i \leq \texttt{n} - 1$, then it is node i that must store the record in its local hash table. For the example, if the logical ring has size $\texttt{n} = 3$, $138 \bmod 3 = 0$ yields a node identifier of 0. Hence, in this case, the leader stores the record for Aruba in its local hash table at position 138.

   If the *id* does not equal the identifier of node i, then node i forwards the `store` command to its right neighbour, along with the record. The record is forwarded around the ring from the leader to node whose identifier equals *id*, where it is then stored in the local hash table.

   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 traversed.

   On completion of this step, all of the records of the country statistics dataset will have been distributed across the nodes that are working together to maintain the DHT.

3. **DHT construction is complete.** The leader sends a `dht-complete` message to the server.

1.2.2 Querying the DHT

A query of the DHT is initiated by a `query-dht` command sent by any process `user-name` that is not involved in maintaining the `DHT`. Recall that the server may return the 3-tuple of *any* the one of the processes involved in maintaining the DHT, *i.e.*, the query process can start at any node in the logical ring. On receipt of `SUCCESS` of the `query-dht` from the server, the `user-name` process follows these steps:

1. The message returned to `user-name` from the server includes the process with which to initiate the query in the `DHT`. Therefore `user-name` sends a `query ``Long Name''` to the process at `IPv4-address` listening on `port` from the 3-tuple. The process in the DHT receiving the `query` computes the hash function of `Long Name`.
2. If the *id* equals the node's identifier on the ring, it examines position *pos* in its local DHT to see if it holds the record. If so, it returns `SUCCESS`, and includes the entire record associated with `Long Name` to `user-name`; otherwise it returns `FAILURE`.
3. If the *id* does not equal the node's identifier on the ring, then the `query` message is forwarded to its right neighbour for processing. The `query` message is forwarded around the ring until the node whose identifier matches *id* is encountered. It then examines position *pos* in its local DHT and responds as described in step 2.
4. On receipt of a `SUCCESS`, the process `user-name` issuing the query outputs the fields of the record. On receipt of a `FAILURE` it outputs that the record associated with `Long Name` is not found in the DHT.

The process `user-name` **must not** compute the identifier of the node to handle the `query` directly. The topology of the DHT, here a ring, **must** be traversed to answer the query by starting with the process returned by the server.

1.2.3 Leaving the DHT

DHTs are maintained by a dynamic set of processes. Existing implementations of DHTs allow new nodes to join the maintenance of a DHT or leave the DHT. Therefore, when a `leave-dht` command is issued to the server by `user-name` with node identifier i this requires the DHT to be rebuilt with $\mathtt{n}-1$ nodes. To accomplish this requires the following steps:

1. User `user-name` retains the 3-tuple of its right neighbour.
2. User `user-name` initiates a `teardown` of the DHT (see step 1 in §1.2.4). When the `teardown` propagates back to `user-name`, it deletes its own local DHT.
3. A renumbering of identifiers is now initiated by `user-name`. A `reset-id` command is sent to its right neighbour setting its identifier to zero, and the ring size to $\mathtt{n}-1$ (but not resetting the neighbours). The `reset-id` propagates around the ring, with each node sending to its right neighbour, incrementing the node identifier when it forwards the message. When a `reset-id` is received by `user-name`, 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 $\mathtt{n}-1$.
4. User `user-name` now must reset one neighbour of each of its own left and right neighbours to remove itself from the logical ring. This involves resetting the left neighbour of `user-name`'s right neighbour to `user-name`'s left neighbour, and resetting the right neighbour of `user-name`'s left neighbour to `user-name`'s right neighbour. New commands `reset-left` and `reset-right` should be introduced to accomplish the removal of `user-name` from the logical ring.
5. User `user-name` 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 the steps in step 2 of §1.2.1 to construct the local DHTs in the new ring of size $\mathtt{n}-1$.
6. On completion of rebuilding the DHT, `user-name` now sends a message for the `dht-rebuilt` command to the server with the `new-leader` specified as its right neighbour.

The state of the process issuing the `leave-dht` is `Free`. It is therefore now able to query the `DHT` or exit the application.

1.2.4 Deletion of the 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. After each node deletes its local DHT, its identifier, and its neighbour information, it forwards the `teardown` message to what was its right neighbour. Hence the `teardown` propagates around the ring, with each node deleting its local DHT, until it returns back to the leader.
2. On receipt of the `teardown` at the leader, it follow suits, and then sends a message for the `teardown-complete` command to the server.

Consider making the implementation of `teardown` generic so that it can be used as part of the processing for the `leave-dht` command; see step 2 of §1.2.3.

1.3 Defining Message Exchanges and Message Format

The previous 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 will ultimately have to define these steps for the `set-id` and `store` commands that are part of the `setup-dht` command, the `query` command that is part of the `query-dht` command, the `reset-id`, `reset-left`, `reset-right`, and `rebuild-dht` commands that are part of the `leave-dht` command, the `teardown` command that is part of the `teardown-dht` command, and the command `join-dht`.

In addition, you will need to define the format of all messages used in client-server and in peer-to-peer communication. This may 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, you may prefer to define the command as a string that you then parse. Indeed, any choice is fine so long as you are able to extract the fields from a message and interpret them.

You may want to define an upper bound on the size a local DHT as 353 (a prime number), and *reasonable* upper bounds on the total number of registered users that your application supports, among others. It may also useful to define meaningful return codes to differentiate `SUCCESS` and `FAILURE` states.

2 Port Numbers

Both TCP and UDP use 16-bit integer port numbers to differentiate between processes. Both also define a group of well-known ports to identify well-known services. For example, every TCP/IP implementation that supports FTP assigns well-known port of 21 (decimal) to the FTP server.

Clients on the other hand, use ephemeral, or short-lived, ports. These port numbers are normally assigned to the client. Clients normally do not care about the value of the ephemeral port; the client just needs to be certain that the ephemeral port is unique on the client host.

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. When possible, the same port is assigned to a given server for both TCP and UDP. For example, port 80 is assigned for a Web server for both protocols, though all implementations currently use only TCP.
- *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. As with well-known ports, the same port is assigned to a given service for both TCP and UDP.
- *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 $[2500, 2999]$, group 4 has range $[3000, 3499]$, and so on.

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

3 Submission Requirements for the Milestone and Full Project Deadlines

All submissions are due before 11:59pm on the deadline date.

1. The milestone is due on Sunday, 09/26/2021. See §3.1 for requirements.
2. The full project is due on Sunday, 10/17/2021. See §3.2 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.

3.1 Submission Requirements for the Milestone

For the milestone deadline, you are to implement the following commands to the server: `register`, `deregister`, `setup-dht`, `dht-complete`, and `query-dht`. This also involves implementation of commands that may be issued among peers that are associated with these commands; see §1.2.1 and §1.2.2.

Submit electronically before 11:59pm of Sunday, 09/26/2021 a zip file named `Groupx.zip` where `x` is your group number. **Do not** use any other archiving program except `zip`.

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.
   (c) Describe your choice of data structures and algorithms 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 graders. In addition, give a list of timestamps in your video at which each step 3(a)-3(f) is demonstrated.
2. **Code and documentation. 25%** Submit well-documented source code implementing the milestone of your DHT application.

3. **Video demo. 25%** Upload a video of length at most 10 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 server and peer programs (if applicable).
   (b) Run the freshly compiled programs on at least two (2) distinct end-hosts.
   (c) First, start your server program. Then start three (3) peers that each `register` with the server.
   (d) Create a `DHT` of size three, *i.e.*, use command `setup-dht` with $\texttt{n} = 3$.
   (e) Register another peer process and have it issue a `query-dht` command to the `DHT`. Illustrate a successful and an unsuccessful query.
   (f) Exit the peers using `deregister`; terminate the server process. Graceful termination of your application is not required at this time.

   Your video will require at least five (5) windows open: one for the server, three for the processes involved in creating the `DHT`, and one more for the process querying the `DHT`. Ensure that the font size in each window is large enough to read!

   The output of your commands must be a well-labelled trace the messages transmitted and received between processes so that it is clear what is happening in your DHT application program.

3.2 Submission Requirements for the Full Project

For the full project deadline, you are to implement the all commands to the server 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, and any new command introduced in your design of the `joing-dht` command.

Submit electronically before 11:59pm of Sunday, 10/17/2021 a zip file named `Groupx.zip` where `x` is your group number. **Do not** use any other archiving program except `zip`.

Your `zip` file must contain:

1. **Design document in PDF format. 30%** Extend the design document for the milestone phase of your DHT application program to include details for the remaining commands implemented for the full project.

   (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.
   In particular, for the `join-dht` command of your own design provide more detail similar to that provided for `leave-dht` in this specification.
   (c) Describe your choice of data structures and algorithms 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 graders. In addition, give a list of timestamps in your video at which each step 3(a)-3(f) is demonstrated.

2. **Code and documentation. 20%** Submit well-documented source code implementing your DHT application.

3. **Video demo. 50%** Upload a video of length at most 20 minutes to YouTube with no splicing or edits, with audio accompaniment.

   This video must be uploaded and timestamped before the full project submission deadline.

Design an experiment to demonstrate the functionality of your DHT application that illustrates:

(a) Compilation of your server and peer programs (if applicable).
(b) Creation of processes on at least three (3) distinct end-hosts.
(c) Registration of sufficient peer processes to first create a DHT of size at least three, *i.e.*, `n` ≥ 3, and illustration of successful and unsuccessful queries to it.
(d) Decreasing the size of the DHT, *i.e.*, execution of the command `leave-dht`, followed by successful and unsuccessful queries to the smaller DHT by one or more peers.
(e) Increasing the size of the DHT, *i.e.*, execution of the command `join-dht`, followed by successful and unsuccessful queries to the larger DHT by one or more peers.
(f) Graceful termination of your application, *i.e.*, tearing down the DHT via `teardown-dht`, and deregistration of peer processes. Of course, the server process needs to be terminated explicitly.

For the end-hosts, consider using the machines on the racks in BYENG 217, or installing your application on VMs in GENI, or using any other end-hosts available to you for the demo.

However many windows you choose to open for your video demo, ensure that the font size in each window 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 clear what is happening in your DHT application program.

References

[1] James Kurose and Keith Ross. *Computer Networking, A Top-Down Approach.* Pearson, 7th edition, 2017.

[2] 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.

[3] 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.

[4] 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.