ARIZONA STATE UNIVERSITY

CSE 434, SLN 70516 — Computer Networks — Fall 2022

Instructor: Dr. Violet R. Syrotiuk

Socket Programming Project

Available Sunday, 09/11/2022; Milestone due Sunday, 09/25/2022; Full project due Sunday,10/23/2022

The purpose of this project is to implement your own peer-to-peer application program in which processes communicate using sockets to implement TWEETER, a highly-simplified social media service.

- 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.
- Each group **must** restrict its use of port numbers to prevent the possibility of application programs from interfering with each other. See §3.3 to determine the port numbers assigned to your group.
- 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 architecture of the TWEETER application in §1, the requirements of the peer-to-peer protocol are provided in §2. Some issues for you to consider in your implementation are found in §3. If you are considering an honours contract for this course, some ideas are provided in §4. The submission requirements for the milestone and full project are described in §5.

1 The Peer-to-Peer TWEETER Application Architecture

The architecture of the TWEETER application is illustrated in Figure 1. It shows a tracker process used for managing the users running TWEETER. Before it does anything else, each user (peer process) must register with the tracker.

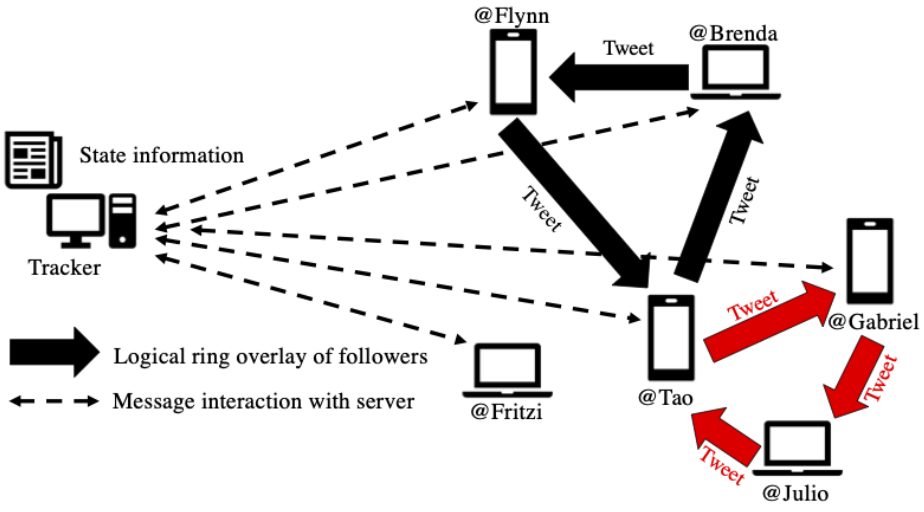

Figure 1: Architecture of the TWEETER application.

Users query the tracker and choose to follow tweets of other users (for simplicity, we are not going to store tweets, or tag them). When a user wants to tweet, it queries the tracker for a list of its followers and constructs a logical ring including the user and its followers. Once the logical ring is set up, the tweet is propagated around the ring and output

by each user. The logical ring may grow or shrink in size as the number of followers of a user grows or shrinks. The logical ring is only torn down if the user exits the system.

Each user may be part of many logical rings. Specifically, each user is part of a logical ring of every user it chooses to follow, and part of a ring involving its own followers.

Figure 1 shows a scenario in which six users (peers) have registered with the tracker. In this scenario, @Brenda has two followers, @Flynn and @Tao. When she tweets, the tweet propagates around the logical ring of size three to her followers in alphabetical order. @Tao also has two followers, @Gabriel and @Julio; his tweet propagates around a different logical ring of size three. In this scenario, @Tao is part of two logical rings, one as a follower of @Brenda, the other including his own followers.

2 Requirements of the TWEETER Socket Programming Project

This project implements TWEETER, a peer-to-peer social media service application in which processes communicate using sockets. This involves the design and implementation of two programs:

1. One program, the `tracker`, maintains state information about the users using TWEETER. The `tracker` must be able to process messages issued from a `user` via a text-based user interface. (No fancy UI is required!) Your `tracker` should read one command line parameter specifying the port number (from your range of port numbers) at which it listens for commands. The messages supported by the `tracker` are described in §2.1.

2. The second program, the `user`:

   (a) interacts with the `tracker` as a client, and
   (b) interacts with other `user` processes as their peer in TWEETER. Your process should read at least two command line parameters, the first is the IPv4 address in dotted decimal notation of the end-host on which the `tracker` process the running, and the second is the port number at which the `tracker` is listening. The port number should match the port number parameter of the `tracker` process. The messages to be supported by a user interacting with the `tracker` as a client are described in §2.1. Some of the messages to be supported by a user (peer) interacting with other users as peers are defined in §2.2 whereas for others you have the opportunity to design the protocol yourself.

Depending on your design decisions (see §3), you may add additional command line parameters to your programs.

2.1 TWEETER: The Tracker

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].

Every peer-to-peer application requires an always-on server, located at a fixed IP address and port, to manage processes running the application. In this project, the `tracker` is the server. It maintains a "database" of users currently running the TWEETER application, and maintains records of the followers of each user, among other information. A single `tracker` process must be started before any `user` processes are run. The `tracker` runs in an infinite loop, repeatedly listening for a message on the port bound to the UDP socket sent to it from a `user`, processes the message, and responds back to the `user` following the protocol.

To be interesting, at least two `user` processes must be started, either on the same end-host as the `tracker` or on different end-hosts. Each one reads commands from `stdin` (again, no fancy UI is expected) until it exits the application. After registering with the `tracker`, the `user` may choose users to follow. When a user tweets, the tweet is propagated to its followers around a logical ring overlay network; see §2.2 for details.

You must output a well-labelled trace the messages transmitted and received between `user` and `tracker` processes, and among `user` processes, as well as other explanatory output, so that the sequence messages sent and received by your TWEETER application is absolutely clear.

In the following, angle brackets ⟨⟩ delimit parameters to the command, while the other strings are literals. The `tracker` must support messages corresponding to the following commands from a `user`:

1. `register @⟨handle⟩ ⟨IPv4-address⟩ ⟨port⟩`, to register the `handle`, *i.e.*, the user name of the user, with the `tracker`. The `handle` is an alphabetic string of length at most 15 characters. The `IPv4-address` is the IPv4 address in dotted decimal notation of the end-host running this user process. This address need not be unique because one or more processes may run on the same end-host. The `port` is one or more port numbers associated with this user used for communication with this user; how many ports you register for each user depends on your design decisions; see §3. The port numbers must come from the set assigned to your group; see §3.3 to find this information.

   Each `@handle` must only be registered once. This command returns `SUCCESS` if the user's handle is not a duplicate among all peers registered. In this case, the `tracker` stores a tuple consisting of the user's handle, its IPv4 address, and one or more ports to use to communicate with that user in a "database." You may implement the "database" however you see fit.

   Otherwise, the server takes no action and responds with `FAILURE` to the user indicating failure to register the handle due to a duplicate registration, or any other problem.

2. `query handles`, to query the handles currently registered with the `tracker`. This command returns an integer `n` equal to the number of registered handles, and a list of the `n` handles. If there are no handles registered, then zero is returned for `n` and the list is empty.

3. `follow @⟨handle`$_i$`⟩ @⟨handle`$_j$`⟩`, to indicate that the user with `@handle`$_i$ sending the `follow` command wishes to follow the tweets of the user with `@handle`$_j$. The tracker updates the state of followers of `@handle`$_j$ to include `@handle`$_i$, *maintaining the list in alphabetical order*, and then returns `SUCCESS`. Otherwise, the tracker returns `FAILURE` indicating failure to add `@handle`$_i$ as a follower of `@handle`$_j$.

4. `drop @⟨handle`$_i$`⟩ @⟨handle`$_j$`⟩`, to indicate that the user with `@handle`$_i$ sending the `drop` command no longer wishes to follow the tweets of the user with `@handle`$_j$. The tracker updates the state of followers of `@handle`$_j$ to remove `@handle`$_i$, *maintaining the list in alphabetical order*, and then returns `SUCCESS`. Otherwise, the tracker returns `FAILURE` indicating failure to remove `@handle`$_i$ as a follower of `@handle`$_j$.

5. `tweet @⟨handle⟩ "tweet"`, sends a `tweet` consisting of ≤ 140 characters, by the user with `@handle` to its followers. This involves the construction of a logical ring of the followers of `@handle`, if it does not already exist. If the logical ring already exists, the followers of `@handle` may have increased or decreased since the last tweet, hence followers may need to be inserted into or deleted from the logical ring.

   To accomplish this, the `tracker` constructs a response with an integer `n` equal to the current number of followers of `@handle` and, for each follower $1 \leq i \leq$ `n`, returns a list of `n` tuples, $\langle$`follower`$_i$, `IP-addr`$_i$, `port`$_i\rangle$ assuming here that only one port was registered for each follower of `@handle`. The list of followers maintained by the `tracker` is in alphabetical order, hence the tuples are ordered alphabetically `follower`$_1 <$ `follower`$_1 < \ldots <$ `follower`$_n$. The return code should be set to `SUCCESS` in this case.

   If the `tracker` is unable to construct the response, it returns `FAILURE`.

   The protocol to follow to accomplish the setup or reconfiguration of the logical ring among the peers is described in §2.2. Once the logical ring is set up, the tweet sent by the user is propagated to its followers around a logical ring overlay network. After the tweet returns to the user, it sends a `end-tweet` command to the `tracker`.

   To simplify concurrency control in tweeting is to not allow it; *i.e.*, the simplest thing is for the `tracker`to reject all incoming commands until it receives the `end-tweet` command from `@handle`. How you handle concurrency depends on your design decisions.

6. `end-tweet @⟨handle⟩`, indicates that the tweet has been successfully propagated along the logical ring of followers of `@handle`. If the `@handle` does not correspond to the handle of the user that issued the `tweet` then this command returns `FAILURE`. Otherwise, the `tracker` responds to `@handle` with `SUCCESS`. Depending on your design decisions, the `tracker` may unblock, or restart processing of other commands.

7. `exit @⟨handle⟩`, is issued if the user with the given handle wishes to quit the TWEETER application. First, this command must remove `@handle` as a follower from all users it was following. If a tweet is in process involving `@handle` it must be allowed to finish before processing the `exit`.

   In addition, this command must delete of state of the logical ring of followers of `@handle`. This involves several steps to delete the state of the logical ring; see §2.2.3. The `tracker` waits for `exit-complete` from `@handle`.

   Finally, the user and its list of followers is deleted from the "database" maintained by the `tracker`.

You may consider providing a mechanism to save the state of the `tracker` in a file to facilitate testing and debugging of your application. For example, you may add another command line parameter to your `tracker` that reads the state from the named file. This may allow you to bring up your system more quickly.

2.2 TWEETER: The Users

After creation of a `user` process in the TWEETER application, it first registers a `@handle` with the `tracker`. Then it may issue any sequence of commands, following the format in §2.1.

On issuing a `tweet` command to the `tracker` the steps taken by `@handle` are:

1. Create or update the logical ring of followers of `@handle`.
2. Propagate the text of the `tweet` around the ring. Each follower prints the source (*i.e.*, handle of the previous hop in the ring) and contents of the tweet.
3. Send an `end-tweet` command to the `tracker`.

In the very least, each user must maintain state information about the logical ring of its followers and also about the other logical rings in which it participates.

2.2.1 Create the Logical Ring of Followers

On receipt of `SUCCESS` from the `tracker` to a `tweet`, one tuple for each of `n` followers, such as those given in rows 1 through `n` of Table 1 are returned. A tuple for the user (`@handle`) issuing the `tweet` is added as row zero of the table. Assuming here that only one port is registered for each user, the fields of the tuple are the handle, the IPv4 address, and port number registered with the user process. Recall that the list of followers is in alphabetical order, hence the tuples are ordered alphabetically $\texttt{user}_1 < \texttt{user}_2 < \ldots < \texttt{user}_n$.

Table 1: The `n` tuples returned by the `tracker`. A tuple for the user issuing the tweet is added in row zero.

Index	User	IPv4 Address	Port
0	$\texttt{user}_0$	$\texttt{IP-addr}_0$	$\texttt{port}_0$
1	$\texttt{user}_1$	$\texttt{IP-addr}_1$	$\texttt{port}_1$
2	$\texttt{user}_2$	$\texttt{IP-addr}_2$	$\texttt{port}_2$
	⋮	⋮	⋮
n	$\texttt{user}_n$	$\texttt{IP-addr}_n$	$\texttt{port}_n$

If a logical ring of followers already exists for `@handle` then the ring may need to be reconfigured; see §2.2.2 for details. Otherwise, the steps to set up the logical ring:

1. **Assign left and right neighbours.** First, a logical ring among the `n+1` processes, *i.e.*, the `@handle` and its `n` followers, must be set up. For $i = 1, \ldots, \texttt{n}$:

   (a) The `@handle` sends a command `setup` to $\texttt{user}_i$ at $\texttt{IP-addr}_i$, $\texttt{port}_i$. It sends its `@handle` and the two tuples $(i-1) \bmod \texttt{n+1}$ and $(i+1) \bmod \texttt{n+1}$ from Table 1.

   (b) On receipt of a `setup` command, $\texttt{user}_i$ stores the tuple of $\texttt{user}_{(i-1) \bmod \texttt{n}}$ to use as the address and port of its left neighbour, and the tuple of $\texttt{user}_{(i+1) \bmod \texttt{n}}$ to use as the address and port of its right neighbour, and associates this information with `@handle`.

The `@handle` stores state information about its own left and right neighbours in its logical ring of followers, and also its current list of followers in alphabetical order.

2. **Propagate the tweet.** Messages must travel around the logical ring in one direction, always received from the left neighbour and forwarded to the right neighbour. A tweet always starts and ends at the `@handle`.

   (a) The `@handle` sends the text of the tweet to its right neighbour on its logical ring of followers.
   (b) Each follower of `@handle` receives the tweet from the port associated with its left neighbour for this logical ring, prints the source (*i.e.*, handle that forwarded the tweet) and contents of the tweet, and then forwards the tweet on the port associated with its right neighbour for this logical ring.
   (c) Ultimately, the tweet returns to `@handle`, which stops the propagation.

   The handle of each previous hop neighbour on the logical ring must be part of your output to demonstrate that the tweet is propagating along the logical ring.

3. **The tweet is complete.** The `@handle` sends a `end-tweet` message to the `tracker`.

The user may now issue a new command.

2.2.2 Update the Logical Ring of Followers

If a logical ring of followers for `@handle` already exists, then state information that includes an ordered list of followers is known. Suppose that the list stored is $\texttt{user}_1 < \texttt{user}_2 < \ldots < \texttt{user}_k$ and has length `k`. Now, an up-to-date list of followers is returned by the `tracker`, including tuples such that $\texttt{u}_1 < \texttt{u}_2 < \ldots < \texttt{u}_n$ where `n` may be less than, equal to, or less than `k`.

Design a protocol to traverse these two lists in parallel, inserting or deleting state in the logical ring as necessary. There are two primary cases. First, a new follower of $\texttt{u}_i$ of `@handle` is not in the previous list of followers and lies alphabetically between two other followers, *i.e.*, $\texttt{user}_j < \texttt{u}_i < \texttt{user}_{j+1}$. Then the right neighbour of $\texttt{user}_j$ must be updated to the details of $\texttt{u}_i$. Similarly, the left neighbour of $\texttt{user}_{j+1}$ must be updated to the details of $\texttt{u}_i$. In addition, the left and right neighbour of $\texttt{u}_i$ must also be established.

Secondly, the up-to-date list of followers may no longer contain $\texttt{user}_j$. If $\texttt{user}_j$ lies alphabetically in the up-to-date list, $\texttt{u}_{i-1} < \texttt{user}_j < \texttt{u}_i$, then in general the right neighbour of $\texttt{u}_{i-1}$ must be updated to the details of $\texttt{user}_j$. Similarly, the left neighbour of $\texttt{u}_i$ must be updated to the details of $\texttt{u}_{i-1}$. In addition, the left and right neighbour of $\texttt{user}_j$ must also be removed from its state. (Note, that it's possible that either $\texttt{u}_{i-1}$ or $\texttt{u}_i$, or both, are also not in the previous list of followers.)

Of course, edge cases must be considered and handled appropriately. It is important that state information at the processes be updated in both cases, not to mention the up-to-date list of followers in the state information for `@handle`.

2.2.3 Exiting the TWEETER Application

When a user wishes to quit the TWEETER application, it issues the `exit @`⟨`handle`⟩ command. If a tweet involving `@handle` is being processed it must be allowed to finish before processing the `exit`.

To exit *gracefully*, several steps are involved:

1. The logical ring of the followers of `@handle` must be torn down. Design a protocol to delete the state information at each follower of `@handle`.
2. Send an `exit-complete` from `@handle` to the `tracker` and the `@handle` process terminates.
3. At the `tracker`, `@handle` must be removed as a follower from all users it was following, and the "database" entry for `@handle` must be deleted. These steps will automatically update any logical rings required by subsequent tweets.

3 Implementation Design Decisions

3.1 Number of Ports to Register and Threading

A user process communicates with the `tracker` as a client, and as a peer with other processes in TWEETER. 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 `tracker` for each user. Rather than registering a triple, you may decide to register additional ports, *e.g.*, if you wanted to register three ports with the `tracker`, the format of your `register` command would be:

`register @⟨handle⟩ ⟨IPv4-address⟩ ⟨port0⟩ ⟨port1⟩ ⟨port2⟩`

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 message to read), the process is put to sleep waiting for a message 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 message. 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()` in `C/C++` for details; be sure to pay attention to the return codes.

3.2 Defining Message Exchanges and Message Format

As part of this project, you must to define the protocol for TWEETER. This includes defining the format of all messages used between a `user` and the `tracker` and among peers. This may be achieved by defining a structure with all the fields required by the commands. For example, you could define the command as an integer field and interpret it accordingly. Alternatively, you may prefer to define the command as a string, delimiting the fields using a special character, 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 choose to define the message exchanges among peers using a request/response format as between a `user` and `tracker` or you may make the exchange more complex if you like.

It may also useful to define meaningful return codes to differentiate more specific `SUCCESS` and `FAILURE` states, among other return codes that you may introduce.

3.3 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 of these services 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.

When you assign ports to processes on the same end-host, they must be unique. If you also have processes running on different end-hosts, you may re-use port numbers because each process is addressed by a pair. It is your responsibility to assign port numbers properly to processes.

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

3.4 End-Host Availability

It you are looking for more end-hosts to use in your experimentation, one option is to `ssh` into `general4.asu.edu` and `general5.asu.edu`; as far as I know, these still have ports numbers > 1024 opened up for us. Use `ifconfig` to determine their IP addresses. (I also thought `general3.asu.edu` was available but it is currently refusing connections, so I'm not sure about it.)

In addition, the machines in the racks in BYENG 217 can run your code. Of course, you need to configure their IP addresses and masks so that they are on the same network; see Lab #1.

All of these machines run RedHat Lunix.

4 Honours Contract Ideas

If you are interested in an honours contract for this course, usually it involves extensions to the socket project. Some ideas could be to implement additional command features in TWEETER, for example:

- `message @`⟨`handle`⟩, in a way to send a direct message to the user with `@handle`. Similar to the format of tweets, direct messages are up to 140 characters long, but they are sent privately to a TWEETER user. Think of them as private messages that cannot be followed.
- `block @`⟨`handle`⟩, blocks receipt of tweets from the user with `@handle`.

I am open to your ideas for extending the project, and am especially interested in extensions that involve communication among peer processes. Before you submit a contract, please discuss your ideas with me.

5 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, 09/25/2022. See §5.1 for requirements.
2. The full project is due on Sunday, 10/23/2022. See §5.2 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.

5.1 Submission Requirements for the Milestone

For the milestone deadline, you are to implement the following commands to the `tracker`: `register`, `query handles`, `follow @handle`, `drop @handle`, and a non-graceful termination using `exit`.

Submit electronically before 11:59pm of Sunday, 09/25/2022 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 TWEETER application program *in this order*:

   (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, provide a list of timestamps in your video at which each step 3(a)-3(g) is demonstrated.

2. **Code and documentation (25%).** Submit well-documented source code implementing the milestone of your TWEETER application.

3. **Video demo (25%).** Upload a video of length at most 7 minutes to YouTube with no splicing or edits. You must provide audio accompaniment to explain your demo.

   This video must be uploaded and timestamped *before* the milestone submission deadline.

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

   (a) Compile your `tracker` and `user` programs (if applicable).
   (b) Run the freshly compiled programs on at least two (2) distinct end-hosts.
   (c) First, start your `tracker` program. Then start three (3) `user` processes that each `register` with the `tracker`.
   (d) Have one `user` issue a `query handles` command.
   (e) Issue some number of `follow @⟨ handle ⟩` commands.
   (f) Have one `user` issue a `drop @⟨ handle ⟩` command.
   (g) Exit the peers using `exit`; terminate the `tracker` process. Graceful termination of your application does not involve much at this time.

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

   In addition to your audio accompaniment, you must output a well-labelled trace the messages transmitted and received between `user` and `tracker` processes, as well as other explanatory output, so that the sequence messages sent and received by your TWEETER application is absolutely clear.

5.2 Submission Requirements for the Full Project

For the full project deadline, you are to implement the all commands to the `tracker` listed in §2.1. This also involves the design of the protocols between `user` processes, as described in §**??**.

Submit electronically before 11:59pm of Sunday, 10/23/2022 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 design of your TWEETER application program to include details for the remaining commands implemented for the full project. Provide *in this order*"

   (a) Include a description of your message format for each command designed for `user` process.
   (b) Include a time-space diagram for each command implemented, explaining your design of the update of a logical ring and exiting the system, illustrating 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, provide a list of timestamps in your video at which each requirement 3(a)-3(f) is satisfied.

2. **Code and documentation (20%).** Submit well-documented source code implementing the milestone of your TWEETER application.

3. **Video demo (50%).** Upload a video of length at most 20 minutes to YouTube with no splicing or edits. You must provide audio accompaniment to explain your demo.

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

   Design an experiment to demonstrate the full functionality of your TWEETER application. It must:

   (a) Compile your `tracker` and `user` programs (if applicable).
   (b) Run the freshly compiled programs on at least four (4) distinct end-hosts.
   (c) Register a sufficient number of `user` processes to create two logical rings of followers size three (3). The rings must intersect in at least one user.
   (d) Each command must be demonstrated at least once.
   (e) Design scenarios to illustrate both successful and unsuccessful return codes of each command to the `tracker`.
   (f) Gracefully terminate your application, *i.e.*, exit each `user` processes. Of course, the `tracker` process needs to be terminated explicitly.

   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, in addition to your audio accompaniment, you must output a well-labelled trace the messages transmitted and received between `user` and `tracker` processes, and among `user` processes, as well as other explanatory output, so that the sequence messages sent and received by your TWEETER application is absolutely clear.

References

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