ARIZONA STATE UNIVERSITY

CSE 434, SLN 11110 — Computer Networks — Spring 2022

Instructor: Dr. Violet R. Syrotiuk

Socket Programming Project

Available Sunday 01/30/2022; Milestone due Sunday 02/13/2022; Full project due Sunday 03/06/2022

The purpose of this project is to implement your own peer-to-peer application program in which processes communicate using sockets to play the game of "Six Card Golf."

- 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 Sunday, 02/06/2022. 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 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 commit changes to your repository on a regular basis.

The rest of this project description is organized as follows. Following an overview of the game of "Six Card Golf" and its application architecture in §1, the requirements of the game's peer-to-peer protocol are provided in §2. The requirements for the milestone and full project submissions are described in §4.

1 A Peer-to-Peer Six Card Golf Game Application

1.1 Rules of the Game of Six Card Golf

The Pack. The game of Six Card Golf uses a standard 52 card deck as depicted in Figure 1. There are 4 suits in a deck: Diamonds, Clubs, Hearts, and Spades. Face cards include Jacks, Queens, and Kings. Black cards include all Clubs and Spades, while red cards include all Hearts and Diamonds.

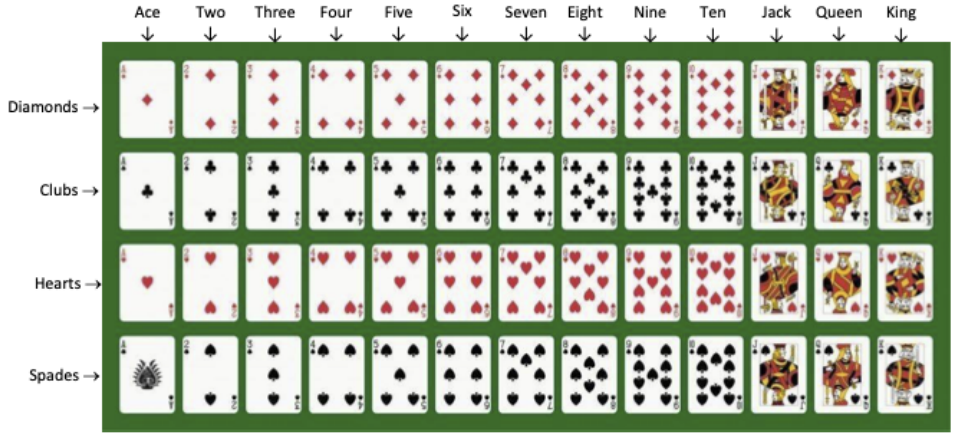

Figure 1: The standard 52 card deck.

Players. Six Card Golf is a game for 2, 3, or 4 players.

The Deal. Each player is dealt 6 cards face-down from the deck. The remainder of the cards are placed face-down, and the top card is turned up to start the discard pile beside it. Players arrange their 6 cards in 2 rows of 3 in front of them and turn 2 of these cards face-up. The remaining cards stay face-down and cannot be looked at.

The Play. The object is for players to have the lowest value of the cards in front of them by either swapping them for lesser value cards or by pairing them up with cards of equal rank.

Beginning with the player to the dealer's left, players take turns drawing single cards from either the stock or discard piles. The drawn card may either be swapped for one of that player's 6 cards, or discarded. If the card is swapped for one of the face-down cards, the card swapped in remains face-up. The round ends when all of a player's cards are face-up.

A game is nine "holes," and the player with the lowest total score is the winner.

Scoring. Each ace counts 1 point. Each 2 counts minus 2 points. Each numeral card from 3 to 10 scores face value. Each jack or queen scores 10 points. Each king scores zero points. A pair of equal cards in the same column scores zero points for the column (even if the equal cards are twos).

1.2 Application Architecture for the Game of Six Card Golf

The architecture of the Six Card Golf application is illustrated in Figure 2. The manager process is a server used for managing players and games. Before it does anything else, each peer process must register with the manager.

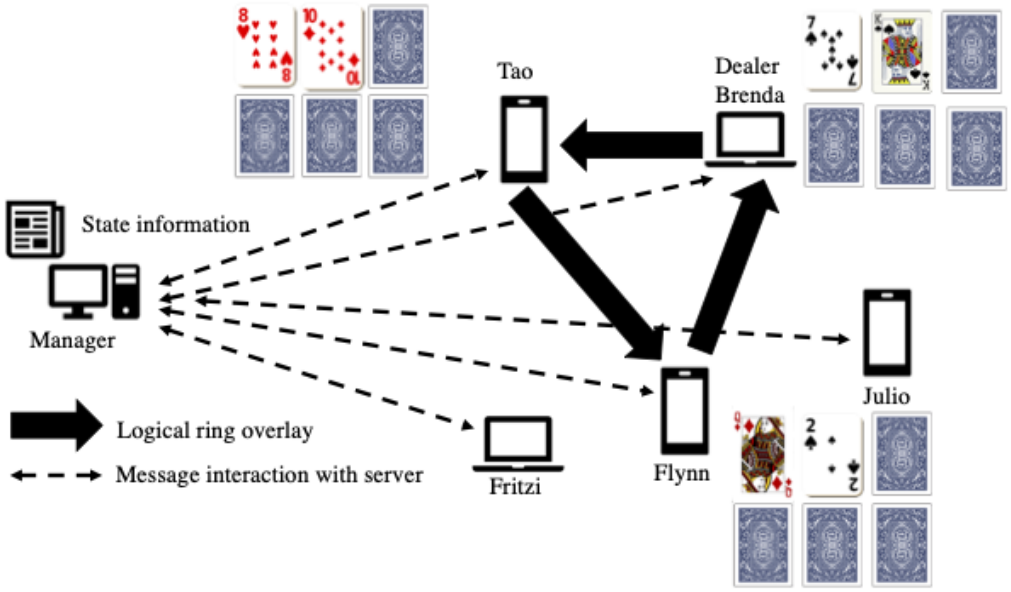

Figure 2: Architecture of the Six Card Golf application.

When a peer wants to start a game of Six Card Golf, it requests $1 \leq k \leq 3$ other peers from the manager to be players in the game. The manager selects k players, returns their information to the peer, and stores all participant information for the game. The peer starting the game acts as the dealer. Using the information returned by the manager, the dealer treats the $k + 1$ players (including itself) in the game as logically organized in the topology of a ring.

The dealer shuffles the cards and then deals each player 6 cards face down, starting with the player to the dealer's left in the ring. The game then proceeds according to the rules described in §1.1 Once all "holes" are played, the dealer announces the winner and the game terminates.

Figure 2 shows a scenario in which five peers have registered with the manager. Only one game of Six Card Golf is ongoing in this scenario, with Brenda as the dealer, and Tao and Flynn as the other players in the game.

Concurrent games Six Card Golf are permitted.

2 Requirements of the Socket Programming Project

This project to implement a peer-to-peer application in which processes communicate using sockets to play the game of Six Card Golf involves the design and implementation of two programs:

1. One program, the `manager`, maintains state information about the players in the game of Six Card Golf and ongoing games. The `manager` must be able to process all commands issued from a `player` via a text-based user interface. (No fancy UI is required!) Your `manager` should read at least one command line parameter, the first giving the port number (from your range of port numbers) at which it listens for commands. The messages to be supported by the `manager` are described in §2.1.

2. The second program, the `player`:

   (a) interacts with the `manager` as a client, and

   (b) interacts with other `player` processes as peers in the game of Six Card Golf as either a dealer or a player. Your process should read at least two command line parameters, the first being a string representing an IPv4 address in dotted decimal notation of the end-host on which the `manager` process the running, and the second being an integer port number at which the `manager` is listening. (This port number should match the port number used in starting the `manager` process.) The messages to be supported by a peer interacting with the `manager` are described in §2.1. You are to design the protocol for the players as described in §2.2.

Depending on your design decisions, you may add additional command line parameters to your programs.

2.1 The Six Card Golf Protocol: The Manager

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 `manager` maintains a "database" of players, and the participants involved in all ongoing games of Six Card Golf. A single `manager` process must be started before any `player` processes. It runs in an infinite loop, repeatedly listening for a message on the port bound to the UDP socket sent to it from a `player`, handles the request, and responds back to the `player`.

At least two `player` processes must be started, either on the same end-host as the `manager` 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 `manager`, the `player` may start a game of Six Card Golf or be selected as a participant of the game started by another player.

During play, any time that the cards of a `player` are written to `stdout`, you should arrange the 6 cards in 2 rows of 3 columns. One way to do it is as follows. For cards that are face-up, print the card number or name (J for Jack, Q for Queen, and K for King) followed by its suit (D for Diamonds, C for Clubs, H for Hearts and S for Spades). If there is a pair of equal face-up cards, they should be printed in the same column. Cards that are face-down should be printed as `***`; use a field of width 3 so that the cards line up nicely. For example, the initial hand of cards of the players Tao, Brenda, and Flynn in Figure 2 would be printed as:

```
 8H 10D ***          7S  KS ***          QD  2S ***
*** *** ***         *** *** ***         *** *** ***
```

You should output a well-labelled trace the messages transmitted and received between `player` and `manager` processes so that it is clear what is happening in your game of Six Card Golf application program.

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

1. register ⟨user⟩ ⟨IPv4-address⟩ ⟨port⟩, where user is an alphabetic string of length at most 15 characters. The IPv4-address is a string giving an IPv4 address in dotted decimal notation. This address need not be unique because one or more processes may run on the same end-host. However, the port (or ports) registered for communication by each process must be unique; see §2.3.1 for suggestions on implementation.

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

   Otherwise, 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. query players, to query the players currently registered with the manager. This command returns a return code equal to the number of registered players, and a list of tuples associated with each player. If there are no players registered, the return code is set to zero and the list is empty.

3. start game ⟨user⟩ ⟨k⟩, to initiate a game of Six Card Golf with $1 \leq k \leq 3$ additional players with user as dealer of the game. This command results in a return code of FAILURE if:

   - The user is not registered.
   - k is not in the proper range; Six Card Golf is a game for 2, 3, or 4 players only.
   - There are not at least k other users registered with the manager.

   Otherwise, the manager sets a return code of SUCCESS, and assigns a new game-identifier for the game, with user as the dealer. It then selects at random k other users from those registered as players in the game. The manager also returns a game-identifier and a list of k users that together will play a game of Six Card Golf. Starting with the dealer, and then or each of the k other players, the manager returns the tuples stored in the "database" for each user. This includes, at a minimum, the user name, IPv4-address, and port. (While the dealer's tuple is given first, the tuples of the players can be given in any order.). Receipt of SUCCESS at the dealer involves several additional steps to accomplish the set-up and play of the game, as described in §2.2.

   Recall that concurrent games of Six Card Golf are allowed. If your program is single-threaded, you should ensure that the sets of players in different games are disjoint. If your program is multi-threaded, each player could participate in multiple games. This single- vs. multi-threaded decision may alter the state information the manager maintains, and how you evaluate the last FAILURE condition.

4. query games, to query the games of Six Card Golf currently ongoing. This command returns a return code equal to the number of ongoing games, and a list that includes information for each game, including at least the game identifier, user name of the dealer, and the user names for each other player in the game. If there are no games ongoing, the return code is set to zero and the list is empty.

5. end ⟨game-identifier⟩ ⟨user⟩, to indicate that the game of Six Card Golf with game-identifier initiated by dealer user has completed. A message containing this command should be sent from the dealer to the manager when a winner of the game has been declared. This command returns SUCCESS if the the game identifier and dear match that stored by the manager and returns FAILURE otherwise. On success, the manager deletes the game information from the list of ongoing games.

6. de-register ⟨user⟩, to remove state information about the user at the manager and exit the application. This command returns SUCCESS if and only if the peer with the given name is not involved in any ongoing game of Six Card Golf, as a dealer or as a player. In that case, the tuple associated with the peer is deleted from the "database" maintained by the manager and the peer can safely exit the Six Card Golf application. If the peer is involved in an ongoing game, this command returns FAILURE.

2.2 The Six Card Golf Protocol: The Players

If a `start game` command issued by a `player` process named `user` is successful, then it becomes the dealer in a game of Six Card Golf. In this case, the `manager` also returned $\texttt{k}+1$ tuples to the dealer as Table 1 shows. Assuming the tuple is a triple, the first row is a triple consisting of the user name, IPv4 address, and port number of the dealer. (If you choose to use multiple sockets for communication (see §2.3.1) then rather than a triple, you may return a larger tuple.) The subsequent `k` rows are triples for each of the other `k` players in the game.

Table 1: The $\texttt{k}+1$ triples returned in as part of a successful `start game` command.

$\texttt{user}_0$	$\texttt{IP-addr}_0$	$\texttt{port}_0$
$\texttt{user}_1$	$\texttt{IP-addr}_1$	$\texttt{port}_1$
$\texttt{user}_2$	$\texttt{IP-addr}_2$	$\texttt{port}_2$
$\vdots$	$\vdots$	$\vdots$
$\texttt{user}_k$	$\texttt{IP-addr}_k$	$\texttt{port}_k$

The dealer then follows these steps to play a game of Six Card Golf. Recall that a game consists of playing 9 "holes" of golf. For each hole:

1. **Shuffle the deck of cards.** Form a random permutation of the first 52 integers, and arrange the deck of cards according to it.

2. **Deal 6 cards to each player (including the dealer).** Using the information in the tuples (*e.g.*, Table 1), cycle through the players 6 times to deal 6 cards from the deck; always follow the same order when cycling through players. Players should arrange their 6 cards in 2 rows of 3 in front of them and turn 2 of these cards, at random, face-up.

   Now, the dealer forms the remaining cards into 2 piles, a stock pile and a discard pile. The discard pile is initialized to the top card on the deck after dealing each player 6 cards. The stock pile consists of all remaining cards.

3. **Play the game of Six Card Golf.** Your job is to design the protocol to play the game of Six Card Golf. Similar to the commands listed for the `manager` in §2.1, define the format and the order of messages exchanged between the dealer and each player, as well as the actions taken on the transmission and/or receipt of a message or other event. Players must follow the rules of the game and work to minimize the value of their face-up cards.

   Cycle through the players. Each time, a player draws a card from either the top of the stock or the top of the discard pile. It can be swapped for one of the player's 6 cards, or discarded. If the card is swapped for one of the face-down cards, the card swapped-in is placed face-up.

   The round for the "hole" ends when all of a player's cards are face-up. The dealer accumulates the score on that hole for all 9 holes in the game.

Once 9 holes have been played, the player with the lowest score is declared the winner. The dealer then sends an `end` ⟨`game-identifier`⟩ ⟨`user`⟩ message with appropriate parameters to the `manager` to indicate that the game is over.

You should output a well-labelled trace the messages transmitted and received between `player` processes so that it is clear what is happening in your game of Six Card Golf application program. In particular, each time a player draws a card, the card drawn, the decision made, and the resulting hand should be written to `stdout`.

2.2.1 Player Extension

Design an extension to the play of the game of Six Card Golf that involves more than a player just interacting with the dealer, *i.e.*, the player needs to communicate with other players, in addition to the dealer to decide its move. For example, besides drawing a card from either the top of the stock or discard pile, the player could "steal" (swap) the most advantageous face-up card from another player. Just be sure that your strategy does not introduce an infinite loop. This could be done by, *e.g.*, forcing a "stolen" card to be swapped for a face-down card only.

2.3 Defining Message Exchanges and Message Format

As part of this project, you must to define the protocol for the players. This includes defining the format of all messages used between a `player` and the `manager` and between peer players. This may be achieved by defining a structure with all the fields required by the commands. 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, 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 between peer players using a request/response format as between a `player` and `manager` or you may make the exchange more complex if you like.

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

2.3.1 Implementation Suggestions

A peer process communicates with the `manager` as a client, and as a peer with other processes in the game. 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 triple, you may decide to register additional ports, *e.g.*, if you wanted to register a total of three ports with the manager, the format of your `register` command may be:

$$\texttt{register}\ \langle\texttt{user}\rangle\ \langle\texttt{IPv4-address}\rangle\ \langle\texttt{port}_0\rangle\ \langle\texttt{port}_1\rangle\ \langle\texttt{port}_2\rangle$$

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 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()` for details; be sure to pay attention to the return codes.

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 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.

4 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, 02/13/2022. See §4.1 for requirements.
2. The full project is due on Sunday, 03/06/2022. 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`, `query players`, `query games`, and `de-register`,.

Submit electronically before 11:59pm of Sunday, 02/13/2022 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 game of Six Card Golf 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 game of Six Card Golf 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 game of Six Card Golf application for the milestone must include:

   (a) Compile your `manager` and `player` programs (if applicable).
   (b) Run the freshly compiled programs on at least two (2) distinct end-hosts.
   (c) First, start your `manager` program. Then start three (3) `player` processes that each `register` with the server.
   (d) Have one `player` issue a `query players` command.
   (e) Have a different `player` issue a `query games` command.
   (f) Exit the peers using `de-register`; terminate the server process. Graceful termination of your application is not required at this time.

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

   Be sure that 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 game of Six Card Golf application program.

4.2 Submission Requirements for the Full Project

For the full project deadline, you are to implement the all commands to the server listed in §2.1. This also involves the design of the protocol between `player` processes, as described in §2.2.

Submit electronically before 11:59pm of Sunday, 03/06/2022 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 your game of Six Card Golf 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 designed for `player` process.
   (b) Clearly describe your idea for player extension (see §2.2.1).
   (c) Include a time-space diagram for each command implemented, including your player extension, 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.
   (d) Describe your choice of data structures and algorithms used, implementation considerations, and other design decisions.
   (e) Include a snapshot showing commits made in your choice of version control system.
   (f) 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 game of Six Card Golf 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 game of Six Card Golf application that illustrates:

   (a) Compile your `manager` and `player` programs (if applicable).

(b) Run the freshly compiled programs on at least four (4) distinct end-hosts.
(c) Registration of sufficient `player` processes to start two concurrent games of Six Card Golf, with and without the player extension component, with least two players each.
(d) Use another `player` to query both players and games.
(e) Design scenarios to illustrate both successful and unsuccessful return codes of each command to the `manager`.
(f) Graceful termination of your application, *i.e.*, de-registration of `player` processes. Of course, the server process needs to be terminated explicitly.

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 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 game of Six Card Golf application program.

References

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