ARIZONA STATE UNIVERSITY

CSE 434, SLN 60461 — Computer Networks — Fall 2024

Instructor: Dr. Violet R. Syrotiuk

Socket Programming Project

Available Sunday 09/15/2024; Milestone due Sunday 09/29/2024; Full project due Sunday 10/20/2024

The purpose of this project is to implement your own multi-threaded peer-to-peer (p2p) application program in which processes communicate using sockets to play a card game called "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.
- The application programs that form your solution to this project **must** be multi-threaded.
- 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, 09/22/2024. This group can be the same as or different from the group used in Lab 1.
- Each group **must** restrict its use of port numbers to prevent the possibility of application programs from interfering with each other. As described in §3.3, 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. You **must** commit changes to your repository on a regular basis. You will be required to provide a full commit history to demonstrate both the effort invested in this project and the provenance of your code.

Following an overview of the game of "Six Card Golf" and its application architecture in §1, the requirements of the game's `tracker` and `player` programs are provided in §2. Implementation details, such as the requirement of multi-threading, are given in §3. 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.

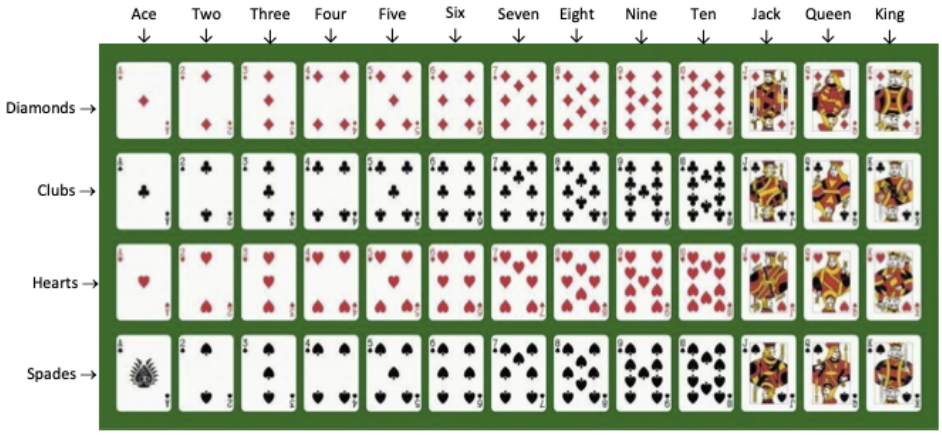

Figure 1: The standard 52 card deck.

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 examined.

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.

By default, 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 tracker process is a server used for tracking players and games. Before it does anything else, each peer process must register with the tracker.

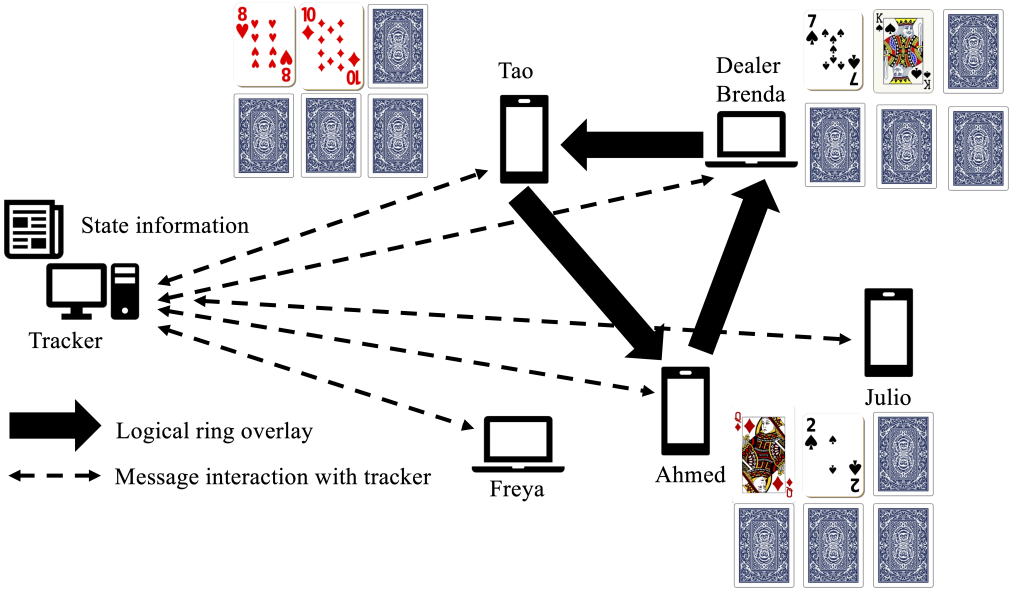

Figure 2: Architecture of the Six Card Golf application; stock and discard pile not illustrated.

When a peer wants to start a game of Six Card Golf, it requests $1 \leq \mathtt{n} \leq 3$ other peers from the tracker to be players in the game. The tracker selects $\mathtt{n}$ 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 tracker, the dealer treats the $\mathtt{n}+1$ 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 tracker. Only one game of Six Card Golf is ongoing in this scenario, with Brenda as the dealer, and Tao and Ahmed 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 `tracker`, maintains state information about the players in the game of Six Card Golf and ongoing games. The `tracker` must be able to process all commands issued from a `player` via a text-based user interface. (No fancy UI is required!) Your `tracker` 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 `tracker` are described in §2.1.

2. The second program, the `player`:

   (a) interacts with the `tracker` 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 four command line parameters, the first being a string representing an IPv4 address in dotted decimal notation of the end-host on which the `tracker` process the running, and the second being an integer port number at which the `tracker` is listening. Of course, this port number should match the port number used in starting the `tracker` process. The third and fourth command line parameters are the port numbers used for communication with the `tracker` and other `peer` processes, respectively. The messages to be supported by a peer interacting with the `tracker` 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 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 tracker located at a fixed IP address and port, to track processes running the application. In this project, the `tracker` maintains a "database" of players, and the participants involved in all ongoing games of Six Card Golf. A single `tracker` 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. Each one reads commands from `stdin` (again, no fancy UI is expected!) until it exits the application. After registering with the `tracker`, 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 (C for Clubs, D for Diamonds, 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 of 3 so that the cards line up nicely. For example, the initial hand of cards of the players Tao, Brenda, and Ahmed in Figure 2 would be printed as:

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

You must output a well-labelled trace the messages exchanged between `player` and `tracker` processes so that their interaction is 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 `player`:

1. `register` ⟨`player`⟩ ⟨`IPv4`⟩ ⟨`t-port`⟩ ⟨`p-port`⟩, where `player` is an alphabetic string of length at most 15 characters. `IPv4` is a string giving an IPv4 address in dotted decimal notation. This address need not be unique because ≥ 1 processes may run on the same end-host. The `t-port` is used strictly for communication between this player and the tracker, while the `p-port` is strictly for communication with other players.

   Each `player` may only be registered once. This command returns `SUCCESS` if the player's name is not a duplicate among all peers registered. In this case, the `tracker` stores a tuple associated with the player in a "database." It also sets the state of the player to `free` indicating availability to play a game.

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

2. `query players`, to query the players currently registered with the tracker. 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` ⟨`player`⟩ ⟨`n`⟩ ⟨`#holes`⟩, to initiate a game of Six Card Golf with $1 \leq$ `n` ≤ 3 additional players with `player` as dealer of the game. The game is parameterized by the number $1 \leq$ `#holes` ≤ 9 to play; if not specified, by default a game has 9 holes.

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

   - The `player` is not registered.
   - `n` is not in the proper range; Six Card Golf is a game for 2, 3, or 4 players only.
   - There are not at least `n` other players registered with the `tracker`.
   - `holes` is not in the proper range; $1 \leq$ `holes` ≤ 9.

   Otherwise, the `tracker` sets a return code of `SUCCESS`, and assigns a new `game-identifier` for the game; it also stores state recording that `player` is the dealer of the game. The `tracker` also selects `n` `free` players at random from those registered as players in the game. Also returned is to the dealer is the `game-identifier`, and a list of the `n` players that together will play a game of Six Card Golf. The list consists of the `player` name, `IPv4-address`, and `p-port` number, of the dealer, and each of the `n` other players, stored in the "database." The `tracker` also updates the state of each player from `free` to `in-play`.

   On receipt of `SUCCESS`, the dealer performs several additional steps to accomplish the set-up and play of the game, as described in §2.2.

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, the name of the dealer, and the 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`⟩ ⟨`player`⟩, to indicate that the game of Six Card Golf with game-identifier initiated by dealer `player` has completed. A message containing this command should be sent from the dealer to the `tracker` after a winner of the game has been declared. This command returns `SUCCESS` if the game identifier and dealer match that stored by the `tracker` and returns `FAILURE` otherwise. On success, the `tracker` deletes the game information from its list of ongoing games and changes the state of the players from `in-play` to `free`.

6. `de-register` ⟨`player`⟩, to remove state information about the `player` at the `tracker` and exit the application. This command returns `SUCCESS` if and only if the given player is not involved in any ongoing game of Six Card Golf. In that case, the tuple associated with the peer is deleted from the "database" maintained by the `tracker` 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 Play

If a `start game` command issued by a `player` process is successful, then it becomes the dealer in a game of Six Card Golf. In this case, the `tracker` also returned `n` + 1 tuples to the dealer as given in Table 1. The first row consists of the dealer's name, IPv4 address, and `p-port` number of the dealer (the `t-port` is not needed for game play). The subsequent `n` rows are tuples for each of the other `n` players in the game.

Table 1: The `n` + 1 tuples returned in as part of a successful `start game` command; $\texttt{player}_0$ is the dealer.

$\texttt{player}_0$	$\texttt{IP-addr}_0$	$\texttt{p-port}_0$
$\texttt{player}_1$	$\texttt{IP-addr}_1$	$\texttt{p-port}_1$
$\texttt{player}_2$	$\texttt{IP-addr}_2$	$\texttt{p-port}_2$
$\vdots$	$\vdots$	$\vdots$
$\texttt{player}_n$	$\texttt{IP-addr}_n$	$\texttt{p-port}_n$

The dealer then follows these steps to play a game of Six Card Golf. Recall that the number of `#holes` to be played in the game is parameterized. 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 order returned in the `start game` command when cycling through players.
3. **Each player turns two of its six cards face-up.** Each Pplayers should arrange their 6 cards in 2 rows of 3 in front of them and turn 2 of these cards, at random, face-up.
4. **The dealer forms the stock and discard piles.** 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.
5. **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 `tracker` 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 each `#holes` in the game.

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

You must 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.

3 Implementation Details

3.1 Managing Sockets

From the $\mathtt{n}+1$ tuples returned by a successful `start game` command, establish an array of sockets indexed by the player's position in the list. The logical ring of processes is then easily implemented: Peer `i`'s right neighbour on the ring is found by writing to the socket descriptor found at position $\mathtt{i}+1 \bmod \mathtt{n}$ in the array.

You must use a different thread for handling the socket associated with the `t-port` and `p-port` of each process. Be aware that by default the function `recvfrom()` in C++ is blocking. This means that when a process issues a `recvfrom()` that cannot be completed immediately (because there is no packet), the process is put to sleep waiting for a packet to arrive at the socket. Therefore, a call to `recvfrom()` will return immediately only if a packet is available on the socket. This may not be what you want.

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

3.2 Defining Message Exchanges and Message Format

Section §2.1 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 need to define the protocols for the game play in §2.2 . This includes defining the format of all messages used in peer-tracker and in peer-to-peer communication. This could be achieved by defining a structure with all the fields required by the command. For example, you could define the name of the command as an integer field and interpret it accordingly. Alternatively, a message can be a string, with concatenated fields separated by a delimiter. Any choice is fine so long as you are able to set and extract the fields of a message and interpret them.

You may want to define *reasonable* upper bounds on the total number of registered peers that your application supports, among others. It may also useful to define meaningful return codes, in particular to differentiate among `SUCCESS` and `FAILURE` states, among other return codes that you may introduce.

3.3 Port Numbers

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

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

4 Submission Requirements for the Milestone and Full Project Deadlines

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

- The milestone is due on Sunday, 09/29/2024. See §4.1 for requirements.
- The full project is due on Sunday, 10/20/2024. See §4.2 for requirements.

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

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

4.1 Submission Requirements for the Milestone

For the milestone deadline, you are to implement the following commands to the `tracker`: `register`, `query players`, `query games`, and `de-register`.

Submit electronically before 11:59pm of Sunday, 09/29/2024 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 Six Card Golf application programs.
   (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. You **must** use a software tool of your choice, *e.g.*, PowerPoint, to draw these diagrams.
   (c) Describe your choice of data structures and algorithms used, and other design decisions.
   (d) Provide a link to a fully accessible page showing the **full** commit history of your project in your choice of version control system, *e.g.*, using the `git log` command in GitHub.
   (e) Provide a *a link to your video demo* and ensure that the link is accessible to graders. In addition, you **must** 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 7 minutes to YouTube with no splicing or edits, and 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 `tracker` and `player` 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) `player` processes that each `register` with the tracker.
   (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 tracker process. Graceful termination of your programs are not required at this time.

   Your video will require at least four (4) windows open: one for the `tracker`, 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.

4.2 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 protocol between `player` processes, as described in §2.2.

Submit electronically before 11:59pm of Sunday, 10/20/2024 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 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, other design decisions, and **in particular, how you have used multi-threading.**
   (e) Provide a link to a fully accessible page showing the **full** commit history of your project since the milestone deadline, 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, you **must** 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.

3. **Video demo (50%).** Upload a video of length at most 20 minutes to YouTube with no splicing or edits, and with audio accompaniment. Control the `#holes` parameter to meet this time limit.

   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 that illustrates:

   (a) Compile your `tracker` and `player` programs (if applicable).
   (b) Run the freshly compiled programs on at least three (3) distinct end-hosts.
   (c) Registration of sufficient `player` processes to start two (2) 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 `tracker`.
   (f) Graceful termination of your application, *i.e.*, de-registration of `player` processes. Of course, the tracker 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 on a LAN you've configured in CloudLab, 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.

References

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