Intro to Nets 2025 Hackathon

Assignment - Blackijecky 

Version 1.0

Introduction

Last year, Nave traveled with a group of his students to an international networks conference about “TCP, UDP and everything in between”. One night, despite strict rules, the students secretly slipped out of the hotel and sneaked into a nearby casino to play blackjack - finally, something that didn’t involve sockets or packet formats. They rushed to a table, excited to try their luck, only to freeze when the dealer looked up and revealed himself… as Nadav, wearing a dealer’s vest and shuffling cards with suspicious expertise. Without missing a beat, he dealt them each two cards and calmly asked, “Hit or stand?”. In order to prevent future midnight casino adventures, We decided that every student would now be required to implement their own blackjack system from scratch - and that is how this assignment was born. Good luck - and may your protocol never bust.

You are asked to develop a client-server application that brings a simple blackjack game to life. The server will host the game and handle all decisions while the client connects to the server and allows the user to play through a short blackjack session.

Each team on the course will write both a client application and a server application, and everybody’s clients and servers are expected to work together with full compatibility.
Please include plenty of comments in the code to improve readability.

Logistics

The Hackathon will be held on **Wed 14/Jan/2026** on -1 floor of building 96, starting from 18:00. Attendance is not mandatory, but you must hand in the coding assignment on Moodle by the end of the day.

For real-time coordination please follow this online document:
An updated link will be attached here later

Game Rules - Simplified Blackjack

In this assignment you will implement a **very simple version of Blackjack** between a **server (dealer)** and **clients (players)**.
Deck and Card Values

- The dealer uses a **standard 52-card deck**.
- Cards have the following values:
    - Number cards (2-10): their numeric value.
    - Face cards (J, Q, K): 10 points.
    - Ace (A): 11 points.
- There is **no special "blackjack" rule**, no betting, and no splitting.
- Each card has one of 4 suits: Heart, Diamond, Club or Spade.

Round Flow

Each **round** between the server and a client follows this sequence:

1. **Initial deal**
    - The dealer (server) shuffles the deck (or uses a fresh, shuffled deck for each round).
    - The client (player) receives **2 cards face-up**.
    - The dealer receives **2 cards**, but:
        - The client **sees only the first card** (face-up).
        - The second dealer card is **hidden** until the dealer's turn.
2. **Player turn (client)**
    - The client can repeatedly choose between:
        - **"Hit"** - ask for another card.
        - **"Stand"** - stop taking cards.
    - After each **Hit**, the server sends the new card to the client and updates the sum.
    - If at any point the client's card sum is **over 21**, the client **busts** and immediately loses the round.
3. **Dealer turn (server)**
    - If the client did **not** bust:
        - The dealer reveals the **hidden second card** to the client.
        - The dealer then **draws additional cards** (and reveal each to the client) until:
            - The dealer's total is **17 or more**,
            - Or the dealer **busts** (sum > 21).
    - The dealer's logic:
        - If sum < 17 → hit.
        - If sum ≥ 17 → stand.
4. **Deciding the winner**
    - If the client busts → **dealer wins**.

- Else if the dealer busts → **client wins**.
- Else compare totals:
  - Client total > Dealer total → **client wins**.
  - Dealer total > Client total → **dealer wins**.
  - Equal totals → **tie** (you may count this as neither win nor loss for the client).

5. **End of round**
   - The server sends the result to the client: win / loss / tie.
   - The client updates its statistics (e.g., number of wins) and moves on to the next round, until the requested number of rounds is completed.

Example Run

1. Team Tzion starts their server. The server prints out "`Server started, listening on IP address 172.1.0.4`" and starts automatically sending out "offer" announcements via UDP broadcast once every second.
2. Team Adnan starts their server, which prints out "`Server started, listening on IP address 172.1.0.88`" and starts automatically sending out "offer" announcements over UDP once every second.
3. Team Joker starts their client. The client asks the user for the number of rounds he wants to play
4. Teams Kanada, Oved and Benjamin similarly start their clients. The clients all print out "`Client started, listening for offer requests...`".
5. All the clients get the Tzion announcement first, and print out ""`Received offer from 172.1.0.4`"
6. The Team Joker client connects to the Team Tzion server over TCP. After the connection succeeds the client sends the number of rounds over TCP connection, followed by a line break ('\n').
7. All other clients similarly send their requests to the server.
8. The server responds to the TCP request by starting a round of the game (instructions above)
9. For each connection, the client and server are supposed to print all different stages of the game

10. When all rounds are completed, the client should print out: "`Finished playing {x} rounds, win rate: {win_rate}`", close the TCP connection and immediately return to step 4.

Packet Formats

There are three packet types: offer (server to client, UDP), request (client to server, TCP) and payload (both directions, TCP).

This is the format of the **offer** message:

- Magic cookie (4 bytes): 0xabcddcba. The message is rejected if it doesn't start with this cookie
- Message type (1 byte): 0x2 for offer (server to client)
- Server TCP port (2 bytes): The port on the server that the client is supposed to connect to TCP requests.
- Server Name (32 bytes): fixed-length name. If shorter than 32 characters: pad with 0x00, If longer: truncate to 32 bytes

This is the format of the **request** message:

- Magic cookie (4 bytes): 0xabcddcba. The message is rejected if it doesn't start with this cookie
- Message type (1 byte): 0x3 for request
- Number of rounds (1 byte): The number of rounds the clients want to play.
- Client team Name (32 bytes): fixed-length name. If shorter than 32 characters: pad with 0x00, If longer: truncate to 32 bytes

This is the format of the **payload** message (note: client and server have different structure):

- Magic cookie (4 bytes): 0xabcddcba. The message is rejected if it doesn't start with this cookie
- Message type (1 byte): 0x4 for payload
- **Client** - player decision (5 byte): send in text the string "Hittt" or "Stand"
- **Server** - Round result (1 byte): win (0x3) / loss (0x2) / tie (0x1) / round is not over (0x0)
- **Server** - Card value (3 byte): The card the client/server pulled from the deck
  Rank encoded 01-13 in first 2 bytes, Suit encoded 0-3 in second byte (HDCS)

Tips and Guidelines

- Please pick a creative name for your team - there will be a contest.
- Both server and client applications are supposed to run forever, until you quit them manually.
- Make sure you map between the card itself (1-13 when the dealer send it to the client) to its value in the game

- It's best to do the development on your own personal phone hotspot. During testing all the teaching staff (Yossi, Nadav, Nave and Erez) will be sharing a testing network. Be prepared to experience and tolerate interference from other teams being tested at the same time!
- The server does not have to listen to any particular port, since this is part of the offer message.
- The client needs to listen for the offer message on 13122 UDP port (It needed to be hardcoded)
- **Do not use busy-waiting (e.g. while-continue loops).** As a general guideline, if your program consumes more than 1% CPU on your own computers, you're probably doing something wrong.
- Think about what you should do if things fail - messages are corrupted, the server does not respond, the clients hang up, etc. Your error handling code will be tested.
- If you try to run two clients on the same computer, they won't be able to listen on the same UDP port unless you set the SO_REUSEPORT option when you open the socket, like this:

```
>>> s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

>>> s.bind(('',13117))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 98] Address already in use

>>> s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
>>> s.bind(('',13117))
```

- The assignment will be written in Python 3.x. You can use standard sockets or the scapy package for network programming. The struct.pack functions can help you encode and decode data from the UDP message.
- Please set up a git repository for your own code and make sure to commit regularly, since the file system on the hackathon computers is unstable. Visual Studio Code has git support so it's quite easy.

To Get Full Points on Your Assignment

If you are coming to the hackathon and show everything works you already have 85. If you want to be competitive, you can ask for an in-depth examination which can raise or lower your grade. We will publish more detailed excellence criteria later, but here are sometimes to get you started:

- Work with any client and any server
- Write high-quality code (see below)
- Have proper error handling for invalid inputs from the user and from the network, including timeouts
- Supporting more complicated versions of the game
- Make your output fun to read
- Collect and print interesting statistics

Code Quality Expectations

How to Submit

Please submit to Moodle a link to your GitHub repository (make sure it's not private). You can commit as much as you want, but only the last commit before the deadline will be considered.

Static Quality

- Code has proper layout, meaningful function and variable names
- Code has comments and documentation for functions and major code blocks
- No hard-coded constants inside code, especially IP addresses or ports

Dynamic Quality

- Return values of functions are checked
- Exceptions are handled properly
- No busy waiting!
- Network code is isolated to a single network

Source control

- Code is hosted in a GitHub repository
- Commits were made by all members of the team
- Proper use of commit messages and branches

LLM Policy

You are, of course, allowed to use Copilot/ChatGPT/etc. to improve your work, but you are expected to be fully responsible for your code and be able to explain each line of code and each high-level architectural decision.

Good luck!