Carleton University
Department of Systems and Computer Engineering
SYSC 3303A RealTime Concurrent Systems Winter 2026

Assignment 2 — Introduction to UDP

Background

In this assignment, you will build a basic three-part system consisting of:

1. A **Client** application
2. An **Intermediate Host** (sometimes called a “router” or “relay”)
3. A **Server** application

The **Client** sends gameplay requests (e.g., move, pick up loot, attack) to the **Intermediate Host**, which forwards them on to the **Server**. The **Server** processes each request (e.g., updating positions, health, or loot availability) and sends its responses back through the **Intermediate Host**, which then relays them to the **Client**. From the **Client**’s perspective, the **Host** appears to be the **Server**, and from the **Server**’s perspective, the **Host** appears to be the **Client**.

In this assignment, the **Intermediate Host** will simply forward packets without modification. However, the intermediate host could be extended to change packets and thus become an error simulator (e.g., introducing packet delays or drops) for the system.

This assignment requires:

- Understanding of **UDP/IP** basics
- Use of Java’s **DatagramPacket** and **DatagramSocket** classes
- Proper **serialization** and **deserialization** of data (converting between strings, bytes, or custom objects)

This assignment **does not** require any knowledge of Java threads or TFTP.

Specification

1. Overview of the Required “Battle Royale” Simulation

Create a simple **real-time Battle Royale**–style system with the following minimal features:

- **Multiple Players** can join. Each player is assigned a unique ID upon joining.
- The **Server** maintains a global “game state,” which tracks:
  - Positions (x,y) and health for each player.
  - Positions and types of loot boxes scattered in the world.
- **Clients** can send requests to the Host, which relays them to the Server. Examples:
  - “JOIN:playerName” (create a new player)

- "MOVE:playerId:dx:dy" (move the player)
- "PICKUP:playerId:lootId" (pick up a loot box)
- "STATE" (request the full game state for debugging or UI display)

- **The Host** listens on a known UDP port (e.g., **5000**) for incoming client requests. It forwards them to the Server's known UDP port (e.g., **6000**). When the Server responds, the Host relays that response back to the **originating client**.

Below are more specific details on how each component should behave.

2. Client Algorithm

1. **Create** a DatagramSocket (bound to any ephemeral port).
2. **Connect/Join**: Prompt the user for a player name and send a "`JOIN:playerName`" request to the Host's known port.
   - Wait for the response packet. If it indicates success ("`JOINED:playerId`"), store your `playerId`.
3. **Gameplay Loop** (text-based UI):
   - Read user commands (e.g., "`MOVE dx dy`", "`PICKUP lootId`", "`STATE`").
   - Construct a corresponding request string ("`MOVE:playerId:dx:dy`") (e.g. "`MOVE:100:5:5`")
   - Send the request (as bytes) to the Host.
   - **Wait** for the Server's response, printing or processing the response upon receipt.
4. **Repeat** until the user quits.
5. **Close** your socket and terminate.

Notes for the Client

- You should print out the requests you send and the responses you receive (for debugging and grading purposes).
- Ensure your data conversions to/from bytes are correct (avoid mixing text vs. raw byte issues).

3. Intermediate Host Algorithm

1. **Create** a DatagramSocket on a well-known port (e.g., 5000) to receive from the Client(s).
2. **Create another** DatagramSocket (or reuse the same one, if done carefully) to send to the Server's known port (e.g., **6000**) and to receive the Server's responses.
3. **Repeat forever** (or until shutdown):
   1. **Wait** for a request from any Client on port 5000.
   2. Print out the information you have received (both the string form and the raw bytes).
   3. **Forward** the exact data to the Server's port (6000).
   4. **Wait** for the Server's response.

5. Print out the response data (both string form and raw bytes).
6. **Send** the Server's response back to the **originating client** on the correct IP and port.

Notes for the Intermediate Host

- The Host **must** forward the **identical** byte array it receives to the Server.
- The Host **must** deliver the **identical** response from the Server back to the Client.
- In the course project, you can create a packet error simulator. You could modify the bytes in flight, drop some packets, or add artificial delays.

4. Server Algorithm

1. **Create** a DatagramSocket to **receive** on a well-known port (e.g., **6000**).
2. **Loop forever:**
    1. **Wait** for a request packet from the Host.
    2. **Parse** the incoming message ("`JOIN`", "`MOVE`", "`PICKUP`", "`STATE`", and "`QUIT`").
    3. **Update** the global `GameState` accordingly. This may involve:
        - Adding a new `Player` if "`JOIN`".
        - Moving an existing `Player` if "`MOVE`".
        - Checking if a `Player` is standing on a `LootBox` if "`PICKUP`".
        - Returning the entire state if "`STATE`".
    4. **Construct** an appropriate response (e.g., "`JOINED:playerId`", "`MOVE_OK`", "`PICKUP_OK`", "`PICKUP_FAIL`", or the serialized `GameState`).
    5. **Send** that response back to the **Host**, using the **Host**'s IP address and **source port** from the request packet.
    6. **Print out** the request and the response for debugging.

Notes for the Server

- You must use the **`GameState`**, **`Player`** and **`LootBox`** classes provided on Brightspace. These classes will be used to store players and loot boxes with positions.
- If a request is **invalid** (e.g., incomplete format), you may choose to ignore it, send an error message back, or throw an exception.
- You can create new sockets for each response or reuse a single socket, as long as you remain consistent with the assignment requirements (must use UDP).

Hints

- The Echo ClientServer example discussed in class and posted on the web site will be useful.
- The APIs for the DatagramSocket, DatagramPacket, and String classes, as well as information on Java arrays which are available through the https://docs.oracle.com/javase/8/docs/api/ help facility, may also prove useful.

- For this assignment and for the project you need to be able to run multiple main programs (projects) concurrently. Each of the client, host and server must have their own main() method and you must run them in the following order: Server, Host then Client.
- Ensure that you follow the specification above. Your datagram sockets (the total number, ports used, send and/or receive) and packet formats must be exactly as described. In the project you will be following a similar specification. Your implementation is considered to be wrong if you do not follow the specification!
- The TAs will mark your assignments in the lab environment. It is your responsibility to ensure that your code works in that environment, and that any software required for viewing any text/diagrams is also present in the lab. You must use the IntelliJ IDE.

Work Products

Your submission must include:

1. "`README.txt`"
   - Briefly describe how to compile and run each component (Client, Host, Server) in **IntelliJ.**
   - List the files in your submission (Java classes, diagrams, etc.).
2. One UML Sequence Diagram
   - Depict the **typical flow** (e.g., a player picking up loot).
   - Show how the Client, Host, and Server interact with the relevant function calls/packet exchanges.
3. One or More UML Class Diagrams
   - Show the internal structure of your system (`Client`, `Host`, `Server`, `GameState`, `Player`, and `LootBox`).
   - You may submit multiple diagrams if you prefer to separate concerns (e.g., one for game logic classes and one for networking classes).
4. Source Code for all three parts of the system
   - `Client`, `Host`, and `Server` Java files.
   - Any additional helper classes (e.g., `GameState`, `Player`, `LootBox`).
   - Ensure your code uses good programming style (meaningful variable names, proper formatting, comments, etc.).
5. Additional Diagrams/Documentation (optional)
   - If you wrote test classes or used advanced architectural patterns, you may include them, but clearly explain them in your README.

For parts 2 and 3, hand drawn scanned diagrams are acceptable, as long as they are neatly drawn and your handwriting is legible, and the software required to view them is present in the lab. As an alternative, you can use Violet UML.

Submitting Assignments

Assignments are to be submitted electronically using Brightspace. Emailed submissions will not be accepted. See the course outline for the procedure to follow if illness causes you to miss the deadline.

Please see the master schedule on the main course website (PDF), or the Course Schedule for the lab section you are enrolled in (click on “Full Schedule”), for due dates. Assignments are due at 9:00pm on the Saturday. Submit early and submit often!