# COSC 3P91 – Assignment 4 – 6920201 & 6973523

BRETT TERPSTRA & MICHAEL BOULOS, Brock University, Canada

## 1 Info

The attached project is a strategy game played on the system console, which acts as the interface between a player and the game world. The game runs on a runnable `gameEngine` object which facilitates game events, keeps track of game events, and defines the top-level behaviour between game objects. Appropriately, execution of the game begins within the main function, which instantiates the game engine and calls its `run` function.

Within the `run` function is the game loop for this strategy game. The game loop involves seven main user-triggered main events, and one main underlying update event:

| **User-Triggered Event** | **Event Description** |
|---|---|
| Build | When this event is triggered, the engine builds the building specified in the command line arguments associated with the command, adding it to the user map contained in the engine members. |
| Train | When this event is triggered, the engine trains/produces the inhabitants specified in the command line arguments associated with the command, adding them to the list of inhabitants in the associated user map. |
| Upgrade | When the upgrade event is triggered, the engine upgrades the unit specified by the index argument and type argument associated with the command, the engine will upgrade the building accordingly to the next allowed stage given the index is valid within the list of units in the map and when the village resource requirements are met. |
| Explore | When the explore event is triggered, the engine will display the next possible map to attack, generated based on find a suitable village relative to the player village's stats. |
| Attack Explored | When the attack event is triggered, the engine will utilize the stats of both the player village and the last non-player village explored to generate an attack result. |
| Generating Army | When the generate army event is triggered, the engine will generate an independent enemy army for testing the defences of the player village. |
| Test Village | Using the previous triggered generate army event, the engine will have armies attack the player village for testing the defensive capability of the player village. |
| **Background Event** | **Event Description** |

| | |
|---|---|
| Update | The update event calls every game loop; it utilizes an in-map timer object, specified in its own utility timer class to facilitate the timing on when certain resources will be mined/produced and other time sensitive game events. |

## Networking Info

The strategy game featured in the project uses networking APIs in Java to facilitate a client/server model execution of the above game loop. The general architecture of the project features a utility class `Server`, which manages the network state of various clients and communicating with multiple game clients. The main method creates a new instance of the `Client` which allows a player to communicate with the server, however, multiple clients can be handled by the server as well. User input is entered through the input stream on the client side, which is then sent to the server that processes the input request and executes the event related to the request, after which, the server sends a response to the client – which the client interprets and outputs on the console.

## 2 Game Usage

To use the game project, a game server must exist and be running, so to begin, run the main method on the `MainServer` class in the source directory – this starts up the game server which uses the console to log requests and responses.

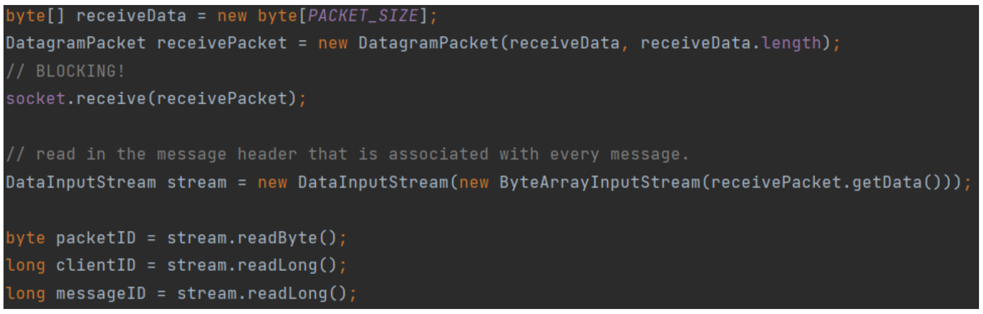
Then, to play the game through a game client, run the `MainClient` class (the Main class is just for demonstration, but any executable Java class with a main function can just create a `Client` object and talk to the server); this creates a game client instance which connects to the server.

After which, you can enter in any of the common game commands from the previous non-networked versions of the game, ie. 1 <building name> to build, 2 <inhabitant name> to train, etc.

**Ideally when starting, type '5' in the input to display your initial village and your options as a player.**

## 3 Sockets

The use of sockets in the project is to facilitate the communication between the server and client over the network, through the aptly named classes `Server` and `Client`. The strategy game, in particular, uses UDP Datagram Sockets to send and receive Datagram Packets back and forth from the server and participating clients.

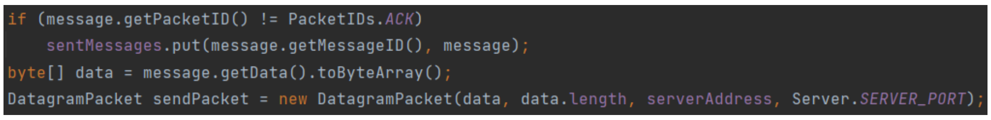


```java
byte[] receiveData = new byte[PACKET_SIZE];
DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
// BLOCKING!
socket.receive(receivePacket);

// read in the message header that is associated with every message.
DataInputStream stream = new DataInputStream(new ByteArrayInputStream(receivePacket.getData()));

byte packetID = stream.readByte();
long clientID = stream.readLong();
long messageID = stream.readLong();
```

Fig. 1. Part of the Server class, this example code shows part of the code that uses a Datagram Packet to receive input from a client over the network



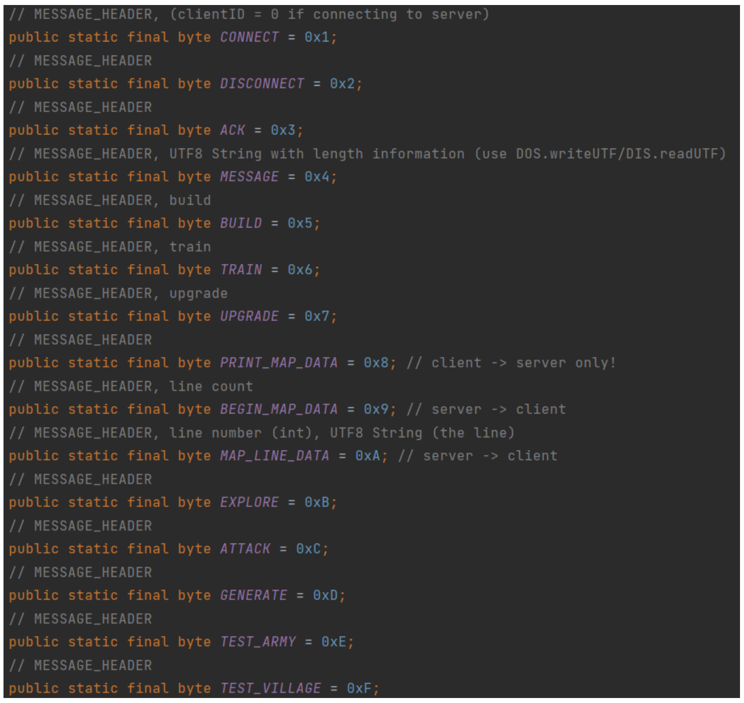
```java
if (message.getPacketID() != PacketIDs.ACK)
    sentMessages.put(message.getMessageID(), message);
byte[] data = message.getData().toByteArray();
DatagramPacket sendPacket = new DatagramPacket(data, data.length, serverAddress, Server.SERVER_PORT);
```

Fig. 2. Part of the Client class, this example code shows part of the code that uses a Datagram Packet to send input to a server over the network

The reason the study opted to use UDP, despite that TCP would have made more sense in the context of game communication not being in real time, was because there was a preference for using discrete packets instead of streams for more control. *~Also, we did it for the joke!*

## 4 Client / Server Communication

To make the communication between the client and the server more standardized, a `PacketIDs` class, which acts as a packet frame, contains static byte headers that the server and client can use to identify different types of packets that represent different requests and messages types being passed.



```java
// MESSAGE_HEADER, (clientID = 0 if connecting to server)
public static final byte CONNECT = 0x1;
// MESSAGE_HEADER
public static final byte DISCONNECT = 0x2;
// MESSAGE_HEADER
public static final byte ACK = 0x3;
// MESSAGE_HEADER, UTF8 String with length information (use DOS.writeUTF/DIS.readUTF)
public static final byte MESSAGE = 0x4;
// MESSAGE_HEADER, build
public static final byte BUILD = 0x5;
// MESSAGE_HEADER, train
public static final byte TRAIN = 0x6;
// MESSAGE_HEADER, upgrade
public static final byte UPGRADE = 0x7;
// MESSAGE_HEADER
public static final byte PRINT_MAP_DATA = 0x8; // client -> server only!
// MESSAGE_HEADER, line count
public static final byte BEGIN_MAP_DATA = 0x9; // server -> client
// MESSAGE_HEADER, line number (int), UTF8 String (the line)
public static final byte MAP_LINE_DATA = 0xA; // server -> client
// MESSAGE_HEADER
public static final byte EXPLORE = 0xB;
// MESSAGE_HEADER
public static final byte ATTACK = 0xC;
// MESSAGE_HEADER
public static final byte GENERATE = 0xD;
// MESSAGE_HEADER
public static final byte TEST_ARMY = 0xE;
// MESSAGE_HEADER
public static final byte TEST_VILLAGE = 0xF;
```

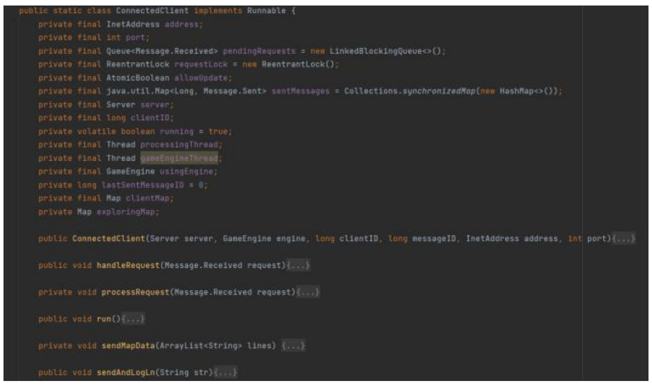
Fig. 3. The `PacketIDs` class, it contains all the possible headers for

the types of packets that can be passed by the Server and client

In addition to the packet frame, the project also employs a primitive 2-way handshake for user authentication. The ACK packet header exists to allow the server and client to make and acknowledgement of the client/server connection prior to communication.

## 5 Multithreading the Server

Activities on the server side are multithreaded to allow multiple clients/players to interact with the game; the server contains a thread for listening and threads for responding to requests from specific clients. Each time a new client connects, a thread is created running an instance of the `ConnectedClient` class that corresponds to that given client, with its own unique client ID that matches that of the connected game client.



```
public static class ConnectedClient implements Runnable {
    private final InetAddress address;
    private final int port;
    private final Queue<Message.Received> pendingRequests = new LinkedBlockingQueue<>();
    private final ReentrantLock requestLock = new ReentrantLock();
    private final AtomicBoolean allowUpdate;
    private final java.util.Map<Long, Message.Sent> sentMessages = Collections.synchronizedMap(new HashMap<>());
    private final Server server;
    private final long clientID;
    private volatile boolean running = true;
    private final Thread processingThread;
    private final Thread gameEngineThread;
    private final GameEngine usingEngine;
    private long lastSentMessageID = 0;
    private final Map clientMap;
    private Map exploringMap;

    public ConnectedClient(Server server, GameEngine engine, long clientID, long messageID, InetAddress address, int port){...}

    public void handleRequest(Message.Received request){...}

    private void processRequest(Message.Received request){...}

    public void run(){...}

    private void sendMapData(ArrayList<String> lines) {...}

    public void sendAndLogLn(String str){...}
```

Fig. 4. The `ConnectedClient` class, it allows the server to handle

multiple clients concurrently by delegating threads to ensure each client

is responded to per request.

On the client-side, there also exists a receive thread which allows the client to send a request while a previous request is awaiting a response. The server handles all the messages, eventually responding to incoming client requests sequentially by way of a 'pendingRequests' queue per `ConnectedClient` thread.

**DOCUMENT END**