COSC 3P91 – Assignment 4 – 7376726

PARKER TENBROECK, Brock University, Canada

Traffic simulation networked and threaded

1 MVC ARCHITECTURAL DESIGN PATTERN

Having this design pattern allowed for easy separation of network and client code. It also allowed for nearly all the code to be resued from last assignment and only need very few minor adjustments. This is because networking, simulation and GUI/TUI are all implemented using 'systems' that can be added/removed at any point. This allows for the Client, Server and Integrated Client to use all of the same code but with different systems attached. The old model of a local client with no networking is still available alongside a server and client (Both Graphical and Terminal).

2 SOCKET DECISION

The communication protocol I designed relies extensively upon all data arriving to both the client and server without dropping data and upon that data being in order. With that in mind it was the obvious choice to use TCP even though it adds some additional overhead.

3 NETWORKED DATA

All integers sent over the network are big endian. These datatypes are compounded together to construct larger packets and datatypes sent over the network.

Kind	representation	bytes
byte	$v_{0..8}$	1
short	$v_{16..8}$, $v_{8..0}$	2
int	$v_{32..24}$, $v_{24..16}$, $v_{16..8}$, $v_{8..0}$	4
long	$v_{64..56}$, $v_{56..48}$, $v_{48..40}$, $v_{40..32}$, $v_{32..24}$, $v_{24..16}$, $v_{16..8}$, $v_{8..0}$	8
float	$v_{32..24}$, $v_{24..16}$, $v_{16..8}$, $v_{8..0}$	4
String	short(=n) + n*byte	2+n
array [element]	int(=n) + n*element	4+n*sizeof(element)
Serialized<Object>	*?	*?

*The last datatype is javas builtin `Serializable` methods and is inserted when an entire object is sent. The size and representation of these types are up to how java Serializes them.

Another Important part are Vehicle/Road/Intersection IDs. These IDs map unique Vehicles/Roads/Intersections to numeric IDs stored as intergers and are used throughout the network protocol.

Author's address: Parker TenBroeck, Brock University, 1812 Sir Isaac Brock Way, St. Catharines, ON, L2S 3A1, Canada.

Update date: 2 April 2024

4 CONNECTION

A walk through of the traffic and steps taken when a new Client is connecting to and communicating with a existing Server.
When a client is constructed it is provided with a controller that defines how it should control the vehicle is it assigned. This controller is the same controller you would normally assign to a vehicle. No changes are needed to be made for the controller to worked with a networked client.

4.1 Authentication and Handshake

```
struct ClientToServerAuth{
    username: String,
    password: String
}
```

```
struct ServerToClientAuth{
    // 69 if authenticated, anything else if not authenticated
    response: byte
}
```

The first step is the client trying to connect to a TCP socket at a user specified IP/URL on port 42069. Once the connection has been established the the client sends a `ClientToServerAuth` packet and waits for a `ServerToClientAuth` packet. On receiving the response the Client will check if it has been authenticated. If so it continues onto later steps, if not it can send a `ClientToServerAuth` packet again or close the connection. On the Server side a new thread is spawned when a client socket is connected and it waits for a `ClientToServerAuth` packet to be received. On receiving the packet the server checks with its `AuthenticationDatabase` instance to see if the specified user&password is in its database. On failure the server sends a `ServerToClientAuth` packet with response byte set to 0 and loops back to waiting for an incoming `ClientToServerAuth` packet. When successful the server responds with a `ServerToClientAuth` packet with a response of 69 telling the client it has sucsessfuly authenticated. It then adds the client to a list of new clients and exits the thread.

4.2 Initialization

Every system tick the Server iterates through each Client in the newClient list. Once it is done it empties the newClient list. Each Client is initialized as follows.

```
// Maps a Vehicle object to a specific ID
struct NewVehicle{
    vehicle_id: int,
    vehicle: Serialized<Vehicle>,
}

// Maps a road whos mapid is road_map_id to an integer id road_id
struct NewRoad{
    road_id: int,
    road_map_id: String,
}
```

```
// Maps a intersection whos mapid is intersection_map_id to an integer id
     intersection_id
struct NewIntersection{
    intersection_id: int,
    intersection_map_id: String,
}

struct ServerToClientInitialization{
    player_vehicle: VehicleID,
    map: Serialized<RoadMap>,
    all_vehicles: [NewVehicle],
    all_road_ids: [NewRoad],
    all_intersection_ids: [NewIntersection],
}
```

Upon client initialization the server will create a new vehicle that will represent the Client this is known as the player_vehicle(referenced in the packet). The server will then construct and send a `ServerToClientInitialization` packet to the client, who is waiting to receive one (still in the simulation initialization phase). Once the client receives the packet it initializes itself and the simulation with the provided values. Most importantly it sets the simulation map to the provided map in the received packet. It is important to save the ID mappings for later as they are referenced in update packets. Once the Client finishes its initialization the simulation can continue initializing remaining systems and start running the system. Once the Server finishes sending the initialization packet it attaches the Client as a controller to the vehicle and queues it to be spawned in a Source intersection.

4.3 Updates

```
// updates and status about the current simulation
struct SimData{
    // the current tick of the simulation
    tick: int,
    // the current frame delta at time of sending this update
    delta: float,
    // the current time in nanoseconds of the simulation
    sim_nanos: long
}

// represents a unique lane
struct Lane{
    // the id of the road this lane is apart of
    road_id: int,
    // the lane index of this lane.
    lane: int
}

// information about changing lanes
struct LaneChange{
    // index of the vehicle behind ourself(player) in the lane to the
    right
    // -1 if none
```

```
    right_lane_back_index: int,
    // index of the vehicle behind ourself(player) in the lane to the
    left
    // -1 if none
    left_lane_back_index: int,
    // index of ourself in the current lane.
    // -1 if none (not currently spawned / placed on a road)
    current_lane_index: int,

    // the lane the vehicle in currently in. fields set to -1 if not in a
     lane.
    current_lane: Lane,
}

// information about turning information and data.
struct TurnData{
    // the road id of the road we are currently being prompted to turn
    off.
    // -1 if none
    turn_road_id: int,
    // the lane number of the road we are currently being prompted to
    turn off.
    // -1 if none
    turn_lane: int,
    // The intersection ID that the vehicle is currently at and can turn
    out of. -1 if none.
    intersection_id: int
}

// information about the status of the player vehicle
struct PlayerData{
    health: float,
    // our actual speed calculated by the simulation
    actual_speed: float,
    reputation: float,

    // information and data relating to changing lanes
    lane_change_data: LaneChange,

    // the lane the vehicle was just placed in during the last update.
    // fields set to -1 if it was not placed in a new lane.
    put_in_lane: Lane,

    // information and data related to turning.
    turn_data: TurnData,
}

// new vehicles that have been added to the road network since the last
    update.
struct NewVehicle{
    vehicle_id: int,
    vehicle: Serialized<Vehicle>
```

```
}


struct VehicleData{
    vehicle_id: int,
    distance: float
}

// how far each vehicle in this lane is along (this) Lane.
struct LaneData{
    vehicle_data: [VehicleData]
}

// How far each vehicle on this road is along (this) Road.
struct VehicleRoadPositions{
    // what road the child vehicles are on,
    road_id: int,
    // index into this array indicates lane # on road
    lane_data: [LaneData]
}

struct ServerToClientUpdate{
    sim_data: SimData,
    player_data: PlayerData,
    // a list of new vehicles that have been spanwed since last update.
    new_vehicles: [NewVehicle],
    // array over all roads in the simulation
    vehicle_positions: [VehicleRoadPositions]
}
```

```
struct ClientToServerUpdate{
   // the desired speed multiplier of the vehicle
   speed: float,
   // -1 for no chosen turn. a valid index to indicate a turn is desired
   turn_index: int,
//   -3,        -2,      -1,        0,      1,          2,        3
//ForceLeft,NudgeLeft, WaitLeft, Nothing,WaitRight,NudgeRight,ForceRight
// respectively
   lane_change: byte
}
```

Every simulation system tick a `ServerToClientUpdate` is sent to every currently connected client and every Client responds with a `ClientToServerUpdate` packet. A Client processing the packet does as follows. It first updates the simulation with the values in the `SimData` portion of the packet. It reads `PlayerData` and saves it for later. It processes all of `new_vehicles` inserting and deserializing any new vehicles and remembering their respective IDs. It then updates the entire RoadMap with the new positions present in `vehicle_positions`. Once the update to the map and simulation is complete it relays the data present in `PlayerData` to the `Controller` provided to itself during its construction. It then relays the returned results back to the server with the `ClientToServerUpdate` packet.

5 MULTIPLE CLIENTS

Multiple clients work without any additional consideration. When another client connects it spawns, looks and interacts with everything else like a regular vehicle. When a client disconnects it simply disappears from the road network like magic.

6 NO CLIENTS

Again because Clients aren't special there is no need for any Clients to be connected for the simulation to run.

7 SERVER MULTITHREADING

The `RoadMap` tick method has been expanded to use a `ThreadPool`. Many small tasks (one for each road) are dispatched to the `ThreadPool` every simulation tick and then spawner thread joins the `ThreadPool` and wait until all tasks have been handled and finished. This improves performance significantly when dealing with many saturated roads. The `ThreadPool` is initialized statically and has # of cores - 1 threads (because the spawning thread helps execution when it calls join) to ensure that the CPU is used as effectively as possible.

As the roads are stored in a list. and the road ticks don't access anything outside the current road there are no special considerations when updating roads in parallel. This also means when the update method is done no matter how many threads the end result of the road updates will always be the same and will be consistent.

NOTES

The build script is intended to be used with a java 17 or higher compiler. The program assumes that when ran in terminal mode it is running on a linux system with stty available and is being ran on a semi modern terminal. When running (especially over network) it might take some time for the server to spawn the player on a road as it doesn't prioritize connected player clients over NPVs(non player vehicles) when spawning entities.

There are a few 'accounts' that are registered by default to the servers Authentication-Database. the accounts respective username and passwords (case matters for BOTH username and password) are as follows
username: '`Parker`' password: '`bruh`'
username: '`TA`' password: '`supersecret`'
username: '`James`' password: '`you_are_my_only_sunshine`'
username: '`Brett`' password: '`femboys:3`'
u sername: '`Robson`' password: '`nice`'
An attempt to login with any other account will result in the server not authenticating you and prompting you to try again.

There are multiple run scripts available. When using the default `run.sh` a single argument specifying how/what will be ran (server/client(gui/tui)/integrated(gui/tui)). If a invalid/no argument is given it will print all the options available. Scripts to run specific parts are also provided for ease of use.

The TUI is setup to use raw mode in terminal. This mean the cursor will not be visible and entered text will not show. This is intended, the ui is designed to be interacted with the arrow keys and using modifier keys (ctrl, shift) to change what they mean.

Left/Right is changing lanes using WaitLeft/WaitRight respectively. Shift + Left/Right changes lanes using NudgeLeft/NudgeRight respectively. and Ctrl + Left/Right changes lanes using ForceLeft/ForceRight respectively. Up/Down move through the turn menu (It is only visible when at a turn). Pressing enter/space when the turn menu is shown will select that turn (no matter if you can fit or not so be carful!). Using Ctrl + Left/Right will increase/decrease the speed of the vehicle. All these options are explained / printed to the screen to ensure the controls are known at all times. An additional quality of life is using 1-9 keys when the turn menu is shown to select the first(1) through nth element up to 9 quickly.