**Overall system design**

Overall architecture is separated into two sections, client and server. Both operate through passing GameplayRequest and GameplayResponse JSON objects. On the server side the request objects are mapped into Java objects as they are received and then back into a JSON compatible mapping format when submitting a message back to the client. The client only handles JSON object format (well Javascript objects). The structure of the server side objects is enforced by Java whereas the Javascript does not enforce object conventions such as Inheritance, encapsulation and method polymerisation. It is important that the attempt at object structuring on the client side is adhered to as closely as possible to simplify overall design.

The flow and control of data is as follows;

* Clients allow users to perform actions in game. Minor checks are run on client side to give a very quick validation as to whether the client is able to perform this action (e.g. Can a building be placed in this spot?). The purpose of this client side validation is to reduce the amount of data being transmitted between server and client. A malicious user would most likely be able to find a way around these checks if they wanted to submit invalid actions to the server (buy a building they cannot afford). Once an action is submitted to the server action processing ends here and users will continue with their game, not seeing any changes to the overall game (e.g. buildings will not begin construction, units will not begin moving, money will not be taken from the user).
* The server receives requests from clients in the form of GameplayRequest objects. The objects are passed in JSON format to the server and interpreted for use as a Java object by the Tomcat server. The player who sent the request is not included in the transmitted object but instead is retrieved from the socketSessionId. Request objects are passed from the SocketServer.java 🡺 RequestProcessor.java 🡺 GameLobby.java 🡺Engine.class where it is finally processed to produce a GameplayResponse which backtracks (as a ‘return’ value) through the chain of calls to the Socket Server object. Once here the GameplayResponse object is broadcast to all clients who are in the socket room; identified by the Game Lobby’s lobbyId.
* Clients have a socket session established with the server at ‘all’ times. These sockets are constantly waiting for retrieval of GameplayResponse objects from the server. This object indicates a change in the game state (such as construction of a new building). The objects themselves contain information required for clients to perform updates to their game state without prior knowledge of the gameplay request submitted to the server.

The system works perfectly… in theory. However socket timeouts seem to occur every time the second hand on the metaphorical clock ticks. Proposals to fix this issue include:

* Shifting a larger amount of the work to the client side to reduce dependency on the server and therefore requiring sockets to be utilised less frequently. This approach would reduce the security of the system.
* Include receipt messages to (important) gameplayResponse messages from the server. This will allow the server to send out repeated messages to clients who did not receive the first message and keep check of which clients are experiencing issues with connectivity (perhaps display this in game as a measure of ‘ping’).

GameplayRequest objects arrive in the Engine object at a method consisting of a single switch statement. The switch statement uses the requestCode to determine which private method to call for further processing. Before calling the method all parameters required for correct processing are confirmed to be present. GameplayResponse objects are constructed in these private methods and either are valid responses which will trigger game state changes, or invalid responses which currently do nothing. The plan is to include a unique communication code; generated, in part, on the server (per client basis) and in part on client side (uniquely identify each message). The code will be passed in the gameplayRequest and returned in subsequent gameplayResponses and will allow the system to generate error messages for users (e.g. not enough cash) which were not detected and displayed initially during checking prior to gameplayRequests. Explanations of planned and existing actions for the system follow.

**Placement and construction of New Buildings**

*Implemented without any purchasing system*

**Purchasing of New Buildings**

**Movement of Units to a Position**

Using A\* algorithm to calculate a path to the target. The map will be split up into squares where one square is occupied by one building/defence. A list of square coordinates between a unit’s current position (square) and target square will be generated. This forms the path which the unit will transverse and will be kept on both client and server side. The client will send updates regarding a unit’s position as they move from coordinate to coordinate. This keeps track of the current state of units on the map. As a new square is entered the previous square will be removed from the server’s path record. If a defence or building is planted on a square which is in the path of a unit, the unit will be notified and a new path will be generated. If the obstruction is enemy occupied the unit may opt to find an alternate route or attack the enemy obstacle. If the obstacle is friendly, the unit will look for an alternate path or return to the centre of its last valid square. Multiple units WILL be able to occupy the same square for now.

**Movement of Unit Groups to a Position**

**Movement of Units to Attack a Target**

**Attacking of Units/Turrets by Turrets/Units**