# COMP SCI 7411 Event Driven Computing Project Plan

Tinson Lai
a1812422

## 1 Preliminary Notes

I've included a batch script to be run on my own Windows machine without altering system environment variables. For Debian-based distributions Linux, they should contain gradle, openjdk-8 and android-sdk packages in their official APT repository. However, I am not quite sure if this would also apply to other distributions as I personally prefer Ubuntu over other distributions. As long as Cordova can run correctly, it should be fine to build and emulate this app. Both Cordova and Backend parts need prior compilation before deploying or running, and I am using Node.js 14+.

## 2 Architecture

The architecture is designed to be centralised, meaning that the client side is essentially a remote controller plus a remote display. All game logics are implemented in and distributed by the backend server. This makes the program suffers from:

- The input received from the client side will be at a very high rate (1 event per 20 ms). This is also the reason I forfeit the plan of deploying it to a public cloud provider. Even large cloud clusters based in Sydney provided by companies like Google or Microsoft will still has a latency of more than 50ms. However, the Dockerfile is still kept in the repository. Containerising it will also cause extra latency.
- The server may not be capable to deal with a large number of clients, partly because of JavaScript itself is quite slow. Also the data transferred between the server and clients are not optimised, so redundant transfer of data may exist in the communication.

## 3 Game Logic

The game will start an endless round when the server is started. There is no automatic termination of the round. Clients may entre and leave the server freely. The basic rules are:

1. The first player entres the server will automatically be selected to the "it" (the one chasing others).
2. All subsequently joining players will be normal players (those who are avoiding being caught by the "it") from the start.

3. If the current ”it” leaves the game, the server will automatically and randomly select a player in the game to be the ”it”.
4. When the current ”it” catch someone, the caught people will respawn randomly in the game, and it will be the new ”it”. The original ”it” is no longer the ”it”.

For simplicity, the game mechanics are:

- All items are squares so that the collisions measurement will be easier.
- There are only four directions (left, right, up, down) to move. This will be controlled by moving the mouse in the browser, or touching the screen on the phone.
- The vision is locked to the player as the centre (mostly similar to the Agar.io game).
- The status of the player is displayed as the border surrounding the square in the game. Each square represents a player. Colours of the square can help players to distinguished their controlled one from others (navy vs blue, darkred vs red).
- To avoid the synchronisation problem caused by socket communication, all updates will be determined by the server before the rendering, meaning that the client side will not do any computations. The process of re-rendering will only base on updates propagated by the server.

The two power-ups, as required, are called capsules in the game, where:

- Yellow capsule will make the ”it” invisible for 15 seconds, or make other invincible for 15 seconds, when the player approach the capsule. The capsule will be regenerated after 20 seconds.
- Black capsule will only have effect if the ”it” eat it. It will stun all players except the ”it” itself for 10 seconds. Others can eat this capsule to prevent being stunned by the ”it”. It will be regenerated for every 30 seconds.

Notice that all effects will be cleared when the player is going to be respawned in the game. The implementation only uses a little bit of RxJS to pipe and repeat the input. Most of the chaining operations are done by `lodash` or JavaScript's built-in functions/methods instead. This is intentional since using the original interface can have better typing support compared to using RxJS, and it will be more efficient for computation.

# 4 Game Input

The game input is gathered directly from the `mousemove` event fired by the canvas element. The marble diagram is too complicated to draw since it involves the usage of `ReplaySubject`. But the process is:

1. The ReplaySubject will only contains a buffer size of 1, then repeatedly emit the last value received in the buffer every 20 ms. The direction will be computed in this step, meanings the direction will be sent to the server instead of letting the server to complete the calculation step.
2. The buffer mentioned before will be updated by a subscription to the `mousemove` event as discussed. To limit the rate of refresh, this subscription also contains a `throttleTime` to reduce the number of updates needed.

The reason I used this in the implementation is that I don't want to put extra computations to the server, thus I let the client repeat the user's last action by default.

# 5 Notes

I've included several lines of comments which are actually part of the code to set up the connection. I didn't comment out the Content Security Policy for Cordova, so it is also necessary to update the EJS template to match the server connection set by the entry script.