The Event Queue

Overview:
In order to handle the possibly large amounts of information being sent between the client and the server, we created our idea of an event queue. Each event which occurs on the client (such as moving a device from one network to another, interacting with an RDT, etc) is compiled into an event, following the structure outlined below, and added to that client's event queue. The events are stored in the event queue on the client side based on the time which they occurred in order to allow for synchronous handling of the events.

Upon syncing with the server, the client wraps the event queue in JSON and sends it to the server. The server then un-encodes the event queue from the client, and inserts the events into its own events based on chronological order. This ensures that the events are handled in the correct order based on time.

Conflict Resolution:
As is common with online systems in which multiple users are editing the same data-structure, many conflicts between events can arise. We needed a way to handle these conflicts. The event queue gave allows us to handle this simply and elegantly. By handling events based on the time at which they occurred on the client, we can process the events occurring first, and then if any conflicts occur, we discard the events occurring later, ensuring that the client which propagated the event first is the one which succeeds.

Luckily, in our tests, such conflicts rarely occur. This is due to the fact that when each event occurs on the client it attempts to contract the server in order to send it the event queue. This ensures the fewest number of conflicts possible.

Local Storage:
The event queue for a particular client (device) is stored in the client's local storage. We chose to store the event queue in this fashion for this particular reason: Suppose that the client became disconnected from the server (by for example, their internet failing), he or she would then lose all of the events which which have occurred on their device which have not yet been propagated to the server. Or even worse, the client does not realize that they have become disconnected from the server, and continues to create events, but does not realize that they are not being propagated to the server. If this was not saved in the local storage of the client, then all of this information is lost. Saving this information allows it to be propagated to the server upon re-connection, as described below.

Offline Capabilities:
A bonus of our event queue system is that, in the case that your internet connection fails, but you are still within a simulation, you are still able to propagate events. These events are stored in the event queue in your local storage, and will be sent to the server when you next connect to the simulation. The conflicts that this might create are handled as stated above, allowing for a semi-functional offline mode to our application.

Event Structure:
The events in our event queue are structured as follows:

{route: string, event_data: {}, time_stamp: string}

Route: A string in the form of a pre-specified url, which informs the server of how to handle this event. An example of a route is “/create/Simulation” which tells the server to use the event_data in order to create a new simulation. A list of all of the routes which our simulation uses can be found in the document “Project API”.

event_data: An object containing all of the data needed to perform this event. A detailed description of all events and all event_data necessary for these events can be found in the document “Project API”.

time_stamp: The time stamp at which the event occurred on the client. This is used in order to order the events and determine when they occurred.