I managed to implement the replaying functionality but there are few hiccups:

- When there is more than one client connected, replaying speed can hardly reach high as expected. Say normal frame rate is 60, to achieve double, it has to be 120. In reality, each *PApplet* can only make it to around 80 even though it has been set to be in 120. I could not figure out why but to blame architecture's efficiency. To make speeding work, I have to set three replay speed as 15, 30, and 60 fps.

- As suggested in document, an event manager class is required in event management system. Besides registration, raising and handling, I think it would be helpful to keep some world information like current state (recording, replaying...) of this manager, frame rate. and add display to each end. But this was unexpectedly problematic.

  Considering there will be one and only one class in the runtime, I set *EventManager* as a singleton class. It would be easy to store information, fetch globally by players, and then render to the screen. However, it turns out it was not quite "global". My *Server* does get a instance of *EventManager* and its information by calling *getInstance()* but clients does not. I guess it was because clients and servers are recognized as separated applet, so when clients trying to get instance, it will get a new and unique instance of *EventManager*, and thus state changes will not get reflected in each client's screen.

  To correct clients' information display, I must not rely on the event manager, I eventually end up to move my frame rate information and replaying state into each player class and make use of their render method to display. It does make things work but on the one hand, the system needs to loop the whole list to change every player's information one by one, which can be expensive, on the other hand, player class structure become even more inflexible, besides its motion State, it now has three additional state to indicate which mode (recording, replaying...) this player is on.

  To get this perfectly solved, it would be the best to turn to networking to set up communication between applets. Similar to what we do to send and receive key input, we send and receive package containing server's setting such as frame rate.

- Players need to be sent back to its state and location after recording ends prior to replaying. At first, I simply keep a copy of speed and location as static in Event Manger but then I realized that, though the character did resume to previous state, moving platforms are not being saved and keeping moving around. To save and load all object's information, instead of keeping a global map, I added *savePosAndSpeed()* and *recoverPosAndSpeed()* to all movable interface object. When a saving order was given from input handlers, movables' position and speed will be saved inside of their own class, rather then event manager.

- To access *PApplet*'s frame rate, I at first spend a long time thinking about setting up a connection between a client applet and player it is controlling. Thanks to processing

build in function, changing replay speed can be simply achieved by *frameRate()*. But how do I know which applet this player is on? I tried to figure out a way to keep an instance of client within a player class but it turns out that was difficult and unnecessary. Finally I found player's *render()* method, which owns applet as a parameter. This reminds me of that, when dealing with situation like separated thread or applet, we can always trace all the way back to functions on the upper level rather than to implement a complicated and costly composition relationship.

- For time and timeline, there are not too much about it. It contains *getLocalTime(),* which is used for client time display and *getWorldTime()*, which is for the server. Moreover, it has *startCount()* and *endCount()*. They are set to count frames elapsed during the recording. More than simply trigger replaying after player pressed stop recording, the event management has to remember how long the recording is, so the manager knows where to stop replaying and resume everything back to normal state before recording and replaying.

  Event Management System Design:

  Character collision, death, input, spawn and replay are events extends class *Event*. There are a handle lock and a replay lock to mark whether this event has been handled or replayed. Event also contains an integer field represent the time it has been handled, so called "timestamp". To construct an event requires a caller, which is normally a player.

  Handlers such as *CharacterCollisionListener*, *DeathListener*, *Inputlistener*, *ReplayListener* and *SpawnListener* all implement the interface *EventListener*, which provides a handle method. This handle method accepts an event as a parameter, and in order to interact with world time and local time, an applet is also required. For example, when a player press button to record, an input event prompting *startRecording* will be sent to *InputListener* and *InputListener* will immediately wake *Timeline* to memorize applet's current frame count then wait until another input event stops the recording.

  EventManager controls the whole registration system. It keeps two thread-safe and ordered queue: *EventQueue* and *RecordingQueue*. While *EventQueue* keeps every event happened in the gameplay, *RecordingQueue* added events only when the system is in recording mode. *RecordingQueue* will also be reset after each record and replay.

  In *EventManager*, the function *attach* mainly does two jobs: to attach an event to an event listener, and add the event to the registration or/and recording queue. When this system is running replay, only input events to change replay speed are allowed. Then in every rendering interval, *update* iterates over every element in the *EventQueue* and ask those events to look for their own listener to handle.

  For a game engine, record and replay in slow motion function is helpful for debugging when interaction between objects are becoming complicated. Besides that, it can also

be added as a feature to the game. Now that we have an event queue and every event are time stamped, we can replay events in reversed order and direction so it looks like the character was traveling back in time. This idea actually can be seen in some racing games, in which players are able to resume to state before cars crush.