Timely

Group 7

Team Members

Zhongyu Xu (z367xu)
Sophie Yuan (s58yuan)
Yi Lu (y424lu)
Jingtong Hu (j278hu)
Zijian Feng (z63feng)
Zhongqi Yue (z3yue)

Github Repo Link

https://github.com/sophieysf/timely

Architectural Description

Client-Server

The Client-Server architecture style consists of two components: server (implemented in the timely-cloud/cloudAPI module) and client (implemented in CloudClient.java in the time-common module), and one connector: the network. In our project, the server stores the application data in the board (implemented in Board.java in the timely-cloud/core) including all event and user information, and contains all backend logic in the model classes (implemented in timely-common/model). The UI layer uses the API calls in the client module to communicate with the server.

How it Supports Functional Properties

Create, Edit, and Respond to Event:

When a creator creates an event, the corresponding client API will send a HTTP request, wrapped in a CreateEventRequest object, to the network. After receiving the restful call from the network, the Event Controller will call the corresponding method in the board, where an event object is created and stored on the board data for future operations. After successfully creating the object, the cloud server will return the created event, and the UI layer can store a local copy of the created object for rendering and other actions. Similarly, when the creator edits the event, or when an attendant responds to the event on the UI layer, the client will either use the according client API call, such as vote or upload user's availability timeslot, to let the cloud server directly makes the change in the board data, or use the updated event object from the UI layer and use the client API call to update the server data accordingly.

Smart Suggestion:

Our system supports two types of smart suggestions: populating poll results and suggesting timeslot, and both types took the same approach. For example, for a particular poll, all of its poll options with their corresponding voters' userID are stored on the server. When a creator clicks the endPoll button on the UI layer, the Poll model will run its endPoll function to find the most voted option. The client side will then use a client API call to update the corresponding data stored on the board in the server. Through this approach, we can make sure both the client and the server side store the most updated version of event data.

Notification:

When there is a need to send notification to all attendants, such as when the time slot of an event is populated, the client side will use a client API call to send a HTTP request to the network, wrapping the corresponding eventID and notification details such as message content. The server will then add the notification to the corresponding event. When a newer version of event data is received from the server using the retrieveEvent client API call, the UI Layer will display the notification to other attendants.

How it Supports Non-Functional Properties

Availability:

1. *Under 90% of the testing circumstances, the app can launch within 4 seconds:*

Client-server architecture supports this property by providing an event library in the client side, which locally stores the IDs of events the current user is in. Upon launching the app, rather than retrieving all

events and then deciding which events belong to the current user, one request containing the user's event IDs can be sent to the server to only retrieve relevant events, making the launching process faster.

2. *After an event is edited, event participants can get notifications within 2 minutes under good Internet:*
The Client-Server architecture allows the client to constantly communicate with the server via a heartbeat, always fetching the latest version of events, so anytime a poll ends or an event is deleted, upon retrieval of the newest version, current event notifications list will be compared with the newest event version's notification list and push new notifications as soon as a new event version is detected. This constant communication between client and server allows new notifications to be retrieved within seconds – much less than 2 minutes.

Usability
1. *80% of the new users can create events intuitively without any instructions:*
Intuitive and easy-to-use UI are crucial for user experience, and these can be achieved by MVVM which separates app's presentation logic from the business logic and intermediate logic. This allows our team members to be divided into two groups, with one group focusing on a responsive, adaptable, and flexible UI design. The accomplishment of a user-friendly interface enables the app to satisfy this NFP requirement.

2. *User can join an event from the home page within 3 taps (paste and join):*
As mentioned in previous NFP, MVVM architecture enables the creation of an intuitive UI. By separating the UI logic from the business logic, the app is elaborately designed to require minimal steps to complete a task, which for this NFP allows users to join an event quickly and easily within 3 taps, i.e., navigate to join fragment, enter event ID and click join button.

MVVM

The MVVM architecture consists of model, i.e., cloud server (implemented in the timely-cloud/cloudAPI module), view models (implemented in timely-client/ui/dashboard/MainViewModel.java and CreateViewModel.java) and views (timely-client/ui/operations).

How it Supports Functional Properties

Create, Edit and Respond to Event:
MVVM architecture consists of Model, View Models, and Views. Model in our app is the cloud server, which saves our app data like events and users, and processes data operations such as creation and editing. We utilized a Java framework that follows MVVM principles as our View Model to store temporary data, and to act as an abstraction between the View and Model. MVVM enables us to create several user-friendly Views/UI for inputting and displaying data. Then by leveraging the View Models, the generated data can be shared among different UI components, which will then be updated and displayed to users. For example, when creating events, inputs from users will be temporarily stored in View Model and then be sent to the server to process. Once the View Model is updated, all UI components related to this user action will be updated immediately to provide feedback to users. Similarly, when the server finishes processing, it will update the View Model accordingly and the Views will be updated as well.

Additionally, our Views provides a straightforward UI that allows users to modify events, and correspondingly, View Models can handle the logic for gathering and storing data changes including creation of polls, modifications of event details and participants information, and changes that are made by users' actions such as responding to events. Then they communicate with the Model and finally generate processed data back to Views via data binding.

Smart Suggestions:
The logic of smart suggestions is implemented on the cloud side. By utilizing Views' data collecting logic, View Models' data generating logic and Model's data processing, smart suggestion and auto-populated event details with poll results can be achieved.

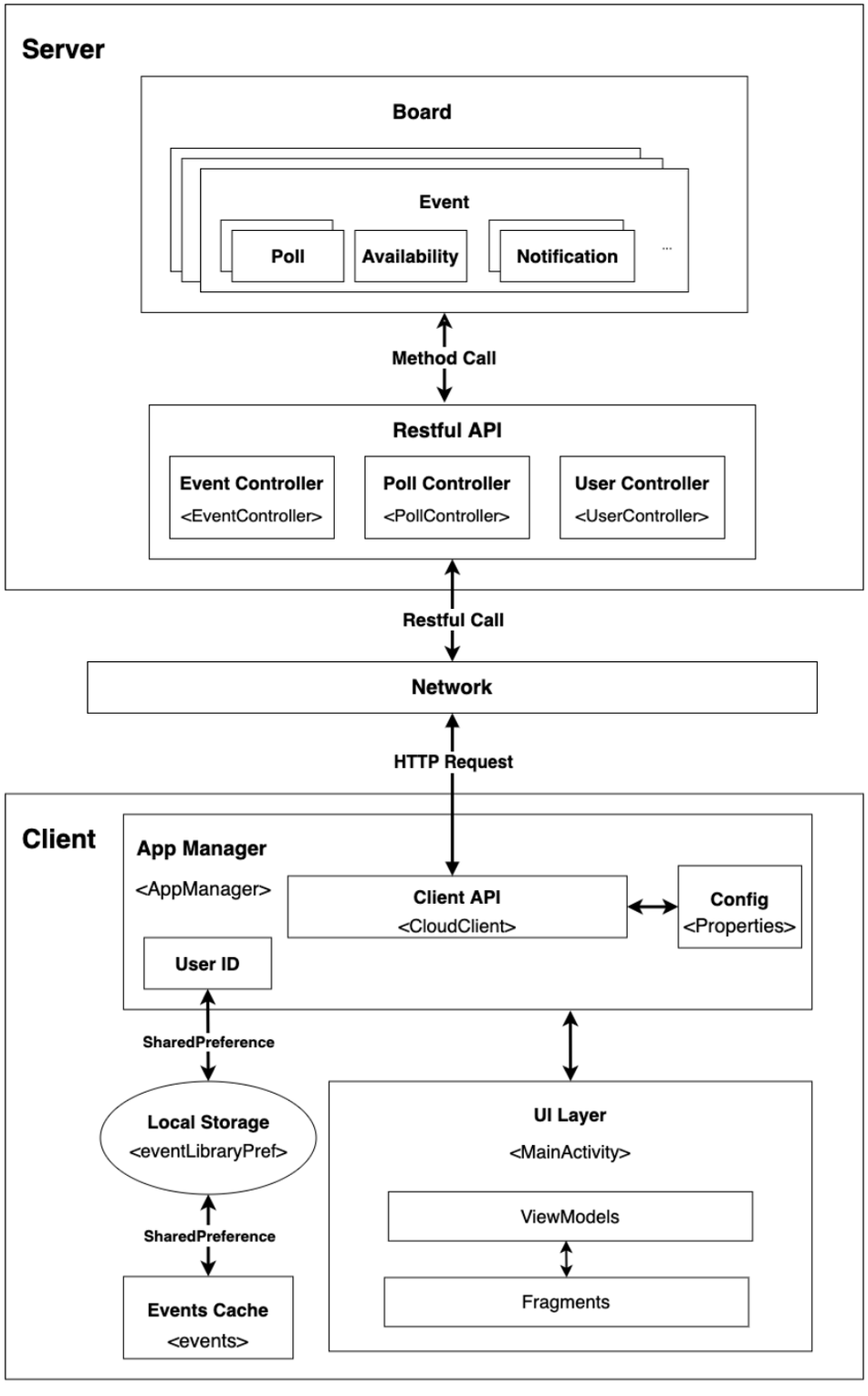
Notification:
View Models are capable of handling data updates and interacting with Views to push notifications promptly, including results of polls, event cancellation and smart suggestions whenever the Model has done data processing.

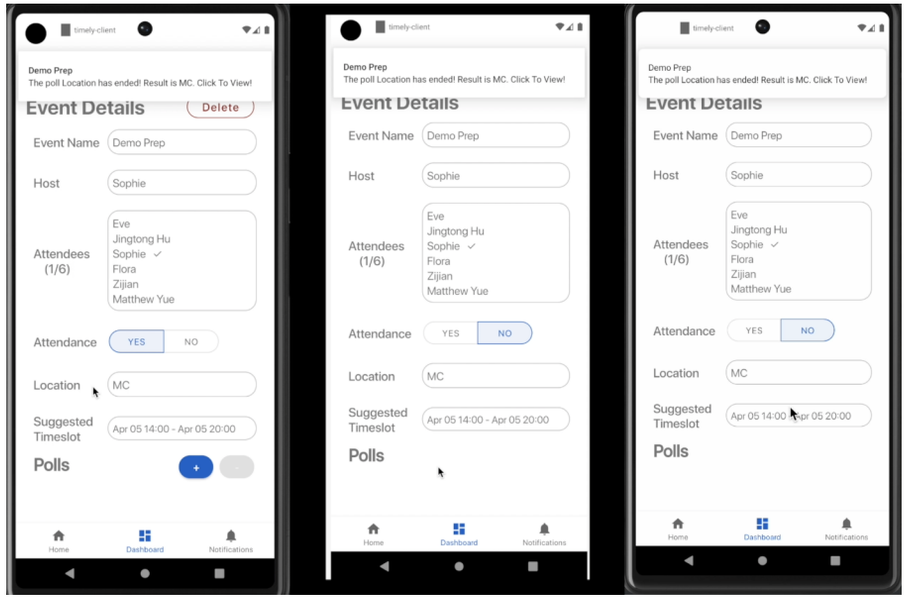
How it Supports Non-Functional Properties

Availability:
1. *Under 90% of the testing circumstances, the app can launch within 4 seconds:*
MVVM architecture supports this property by providing a separation of business logic and UI logic. The separation of concerns facilitated by the MVVM architecture allows the app to reduce the time required for loading UI components during start-up. This can be achieved due to the characteristics of MVVM that UI and related resources that are necessary for display will be loaded first, and other operations and data loading will be processed in the background while not affecting UI's work. This results in improved performance and faster launch times to meet this NFP requirement, which in this case, within 4 seconds.

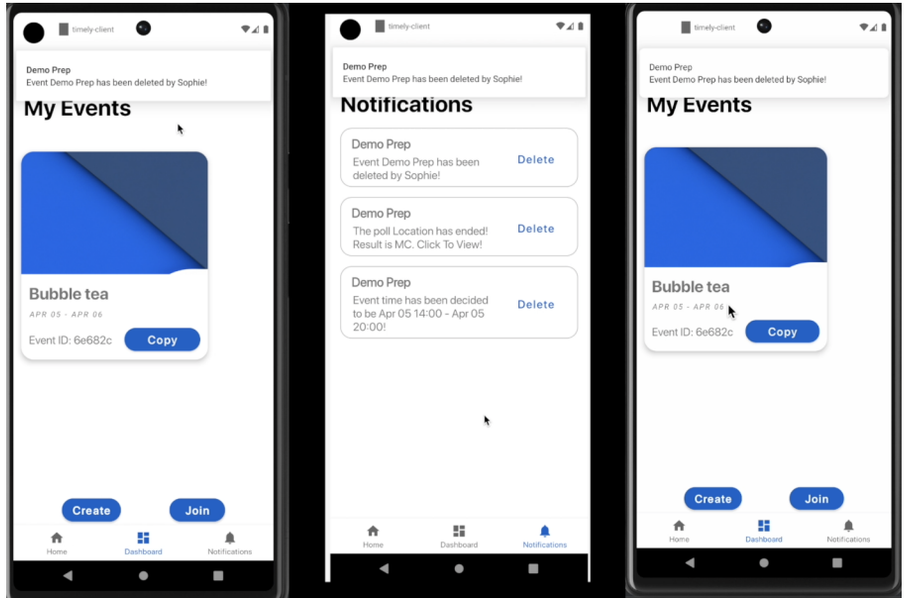
2. *After an event is edited, event participants can get notifications within 2 minutes under normal internet conditions:*
With the View Models in MVVM architecture, changes that made by users from UI side (i.e., Views) will be actively updating and promoting information to Model (i.e., Cloud) via View Models, as View Models provides a layer of abstraction that enables data binding and communication between the Model and the Views. Meanwhile, MVVM utilizes asynchronous programming techniques, which prevents the UI from getting blocked while waiting for data from the cloud, ensuring the app's responsiveness. By employing MVVM, it enables the app to send notifications promptly whenever an event is edited, and distribute notifications in a timely manner, thereby fulfilling this NFP requirement.

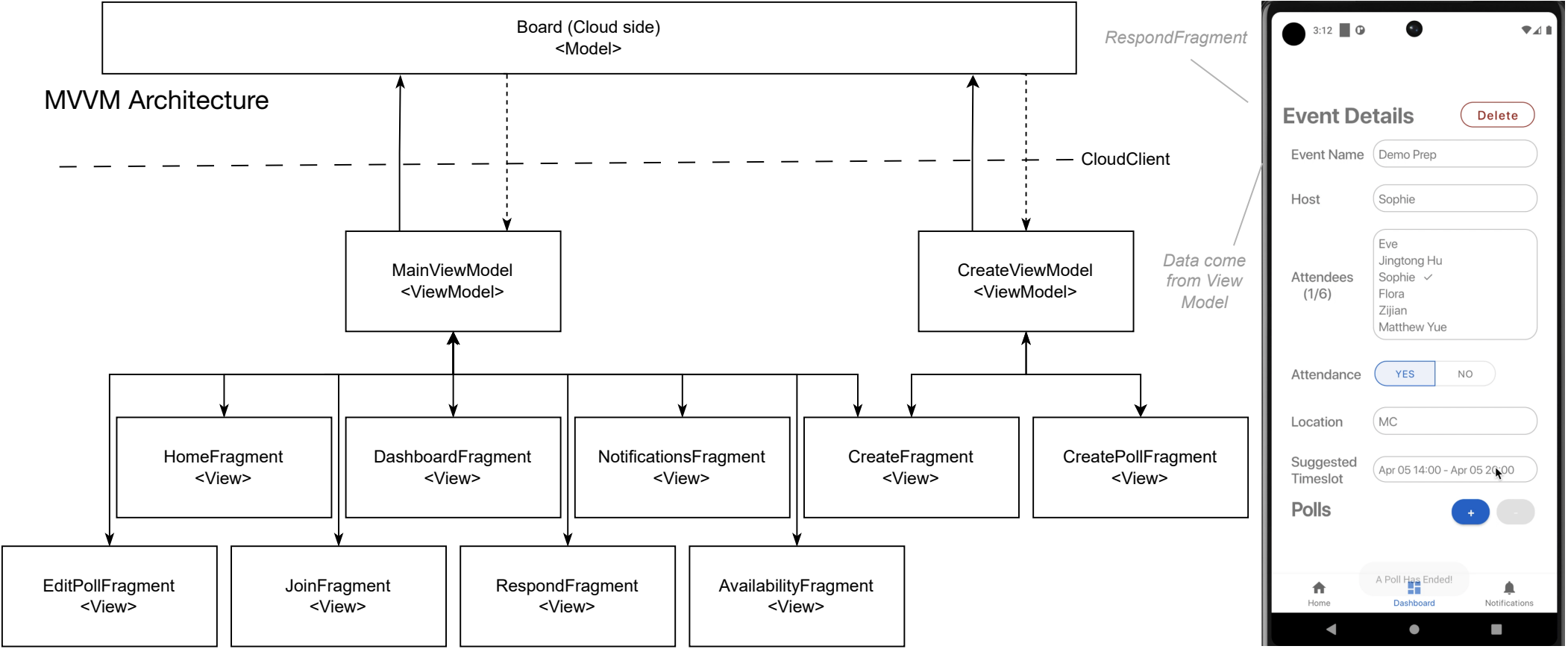
Client-Server Architecture





When a poll has ended or an event has been deleted, new event versions are retrieved from the server via heartbeat so everyone can receive notifications and the event details page are updated in real time.




Board (Cloud side)
<Model>
MVVM Architecture
CloudClient
MainViewModel
<ViewModel>
CreateViewModel
<ViewModel>
HomeFragment
<View>
DashboardFragment
<View>
NotificationsFragment
<View>
CreateFragment
<View>
CreatePollFragment
<View>
EditPollFragment
<View>
JoinFragment
<View>
RespondFragment
<View>
AvailabilityFragment
<View>
RespondFragment
Data come from View Model
3:12
Event Details
Delete
Event Name
Demo Prep
Host
Sophie
Attendees (1/6)
Eve
Jingtong Hu
Sophie
Flora
Zijian
Matthew Yue
Attendance
YES
NO
Location
MC
Suggested Timeslot
Apr 05 14:00 - Apr 05 20:00
Polls
A Poll Has Ended!
Home
Dashboard
Notifications

System Design

Cloud

Physical Location: [timely/timely-cloud]
The cloud server side uses the Restful API to receive HTTP requests, which are calls from client, the controller classes(AvailabilityController, EventController, PollController, UserController) are responsible to receive client calls, and call the corresponding interface in Board class. It corresponds to the Restful API layer in the component diagram.

Board class is the core of the cloud server, corresponding to the Board box in the component diagram. It manages all the event data, including the polls, availability info, participants info inside of the active events. The board class is responsible for all the event information updates, so that there is only one centralized point of update, and there will not be conflict of versions. The clients can retrieve the latest version of the events they joined periodically to update themselves.
So that the external API of the board class is called by the controller layer and exposed to the client side as an HTTP request interface. This interface is an abstraction of the operations that a client needs from the server, and the operations mainly includes three types:

- updating events, including all the create/add/remove operations to polls/availability/notifications
- retrieving the latest events (retrieveEvent(userID, eventID, version))
- Session management(getSessionID())

On board, the event data is saved in a hashmap of eventID mapping to event data, so the board can retrieve or update any event data by the eventID provided on request in O(1) time. There are also hashmaps of pollID/availabilityID mapping to eventID, so we can also find the event by the pollID/availabilityID in O(1) time.

There is a time suggesting algorithm in [timely/common/model/Availability.java] that is used by the cloud server to find the most overlapped time for an event based on the uploaded availability of every event participant. Each availability will be stored as an array with size of two, where the first element indicates start time and the second one indicates end time. The algorithm first separates all start times and end times into two sorted arrays, using two pointers technique to increase the counter whenever a start time is processed and decrease the counter whenever an end time is processed. When the counter is updated, an array with time value and counter value will be put into a priority queue for later to find out the most overlapped time period based on the counter value.

The reason that we choose this cloud server design over Monolithic architecture is because we need real time synchronization, if the server-based design we can keep the copy of the latest event on the server and only keeps it updated, clients can always retrieve the latest version from the server and this design avoids version conflict. This design is better for maintenance, since the data are managed in one position similar to a data repository style.

Communication

Synchronization Data

Physical Location: [timely/timely-common], [timely-cloud/core/Board.java], [timely-client/MainActivity.java]

Each event has a version number, when there are updates to the event, the client calls the cloud server and passes the data that needs to be updated. The cloud server keeps all the active events with the latest version, and performs the update on the latest version of events, and increments the version number. The client side uses a heartbeat to check if all the local events match the latest versions, it retrieves the latest event from the cloud server if the local version is out of date and overwrites the local version.

Connection Design

Physical Location: [timely/timely-common]

For the connection design, we use two **external libraries**, Spring MVC in cloud implementation and Jersey 3.0.3 (Jakarta Version) in client APIs. In this case, our http connection follows the Restful Aritcheture. Controllers in the cloud are the handler of a restful call. CloudClient in timely/common/cloudAPI is the wrapper of a http request. Moreover, we have a default configuration located in assets/config.properties, where the endpoint of the cloud server is defined. To maintain the consistency, the data structures used in client and cloud are all defined in timely-common/model.

Client

Physical Location: [timely/timely-client]

This subsystem is an android application using SDK API 30, integrated with the parent-library jar file 'timely-common.jar' for the connection to cloud server. According to the component diagram, this subsystem has two large components, App Manager(AppManager) and UI Layer including ViewModel such as *MainViewModel* and *CreateViewModel*, and View such as *HomeFragment, DashboardFragment, NotificationsFragment, EditPollFragment, AvailabilityFragment, CreateFragment, CreatePollFragment.* These two parts are the core implementation of UI:

- AppManager inherited from android.application. It is initialized whenever the app boots. It is responsible for maintaining global resources such as *CloudClient*, *Properties*, and UserInfo as an identifier.
- UI layer inherited from android UI sdk. MainActivity is the entry of the app inherited from *AppCompatActivity.*

Cache design: Local cache consists of a local identifier *UserInfo*, an local *SessionID*, *Events* and *Notifications* to save the last runtime status. When the app boots, if the local identifier is not found, a *UserInfo* will be generated as the identifier. If the SessionID from cloud is different from local *SessionID,* cache will be aborted since cloud has been rebooted.

Design Patterns

Singleton Pattern

The Singleton design pattern used in the controllers – EventController, PollController, UserController, and AvailabilityController (implemented in timely-cloud/controller) to restrict the instantiation of the Board. It ensures that there is only one instance of the Board class which stores all events, users, and event-user mappings. Having multiple instances of Board could lead to conflicts and data corruption, such

as not being able to find an event or user since some events are stored in one instance and some are stored in another instance. By using the Singleton pattern, we can ensure that the only instance can be accessed easily and consistently in our program, as well as making it easier to manage shared data resources.

Observer Pattern

The Observer design pattern is utilized in the Client component of our app, which is illustrated in our UML diagram. This pattern is used to facilitate seamless communication between the two View Models and all Views/Fragments. In our application specifically, MainViewModel and CreateViewModel are subjects and all fragments are observers. MainViewModel contains most of the data models, whereas data models in CreateViewModel will be renewed whenever a user is creating a new event. With Observer design, any user interactions made in views from the UI side will be captured by views and such interactions will be processed in cloud server via view models. Then view models process data changes, update its internal data models and finally notify corresponding registered observers/views, e.g., views observe mutable live data which reside in view models. This ensures that all updates in the View Model will be automatically reflected in views. Moreover, the Observer design pattern introduces a low coupling between our views and view models, providing an architecture that can be maintained and modified easily and efficiently. That is, it allows modifications to views without impacting view models, and vice versa. This leads to an optimization in performance, effectively reducing unnecessary processing. Overall, with the Observer design pattern, our app can achieve better flexibility, maintainability, and responsiveness.

Coupling

Our design of coupling is mainly reflected with three levels. On the highest level, we separate server side and client side. Both components can run independently and not affect one another. If the server is down, our UI display will remain stable and keep responding to our users. If the UI is down, all the data and operations on the server will not be affected or deleted. The next time when the UI is back up, users will still have the most up-to-date information.

The next level is on the application level. There is an application manager class that includes all the essential application related data, such as current user ID. Suppose we need to add a new feature or delete some existing implementations, we can simply edit the app manager class because it has all the information we need.

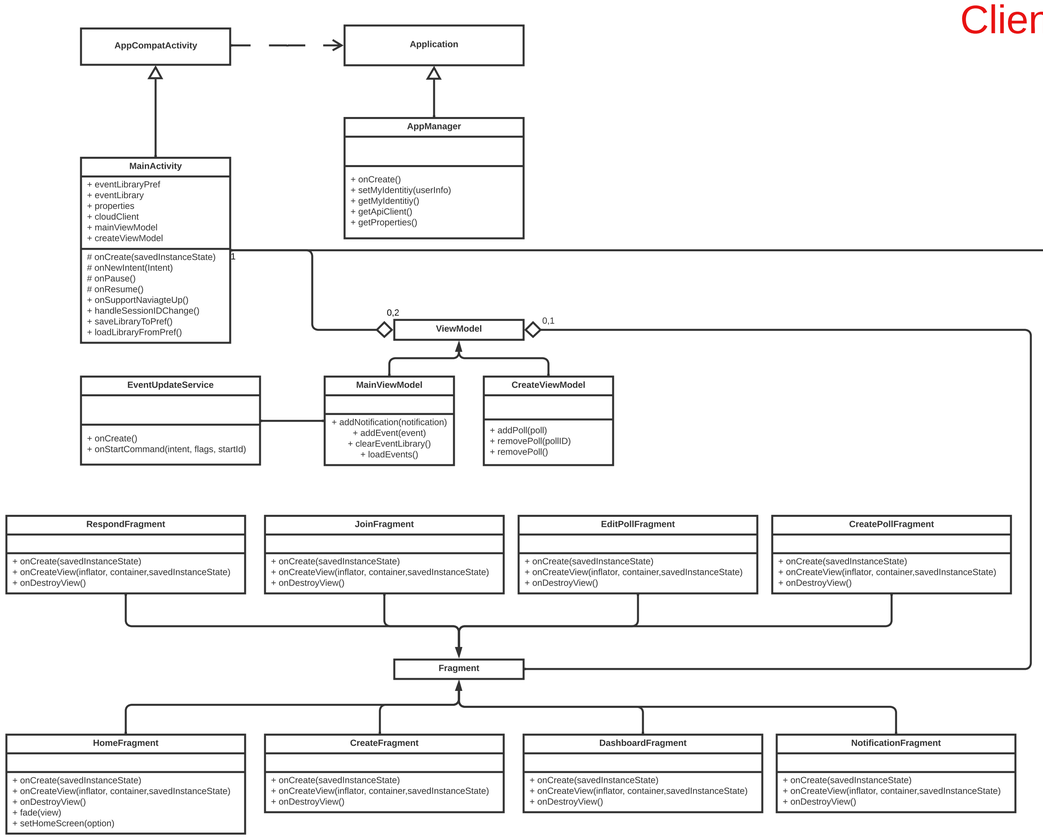
The third level is between view and view models. The view models contain the most up-to-date temporary information for users. It acts like a cache in our application. All the data displayed to users comes from the view models. If the user loses connections to the internet or the user's phone freezes, the information in view model will remain stable and will be provided to users once the connection is back. If we would like to add new features, we can simply add the information that users need to see in the view model and our views will be updated accordingly.

Imagine we have a big change in the future which requires our app to allow co-host of any events. All the existing implementations could remain unchanged. We could simply add a new co-host on the server side and pass the information to the UI through the view model. The UI will be updated accordingly and co-host information will be kept both on the server permanently and on the UI side temporarily.

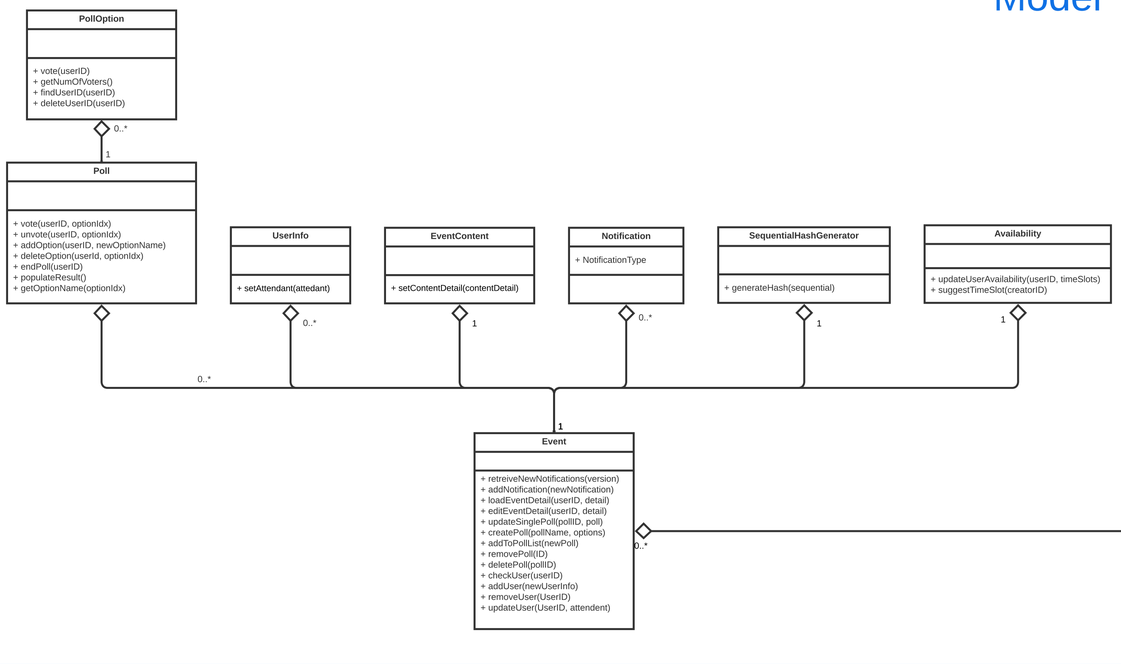
Timely UML

Group 7 | April 09

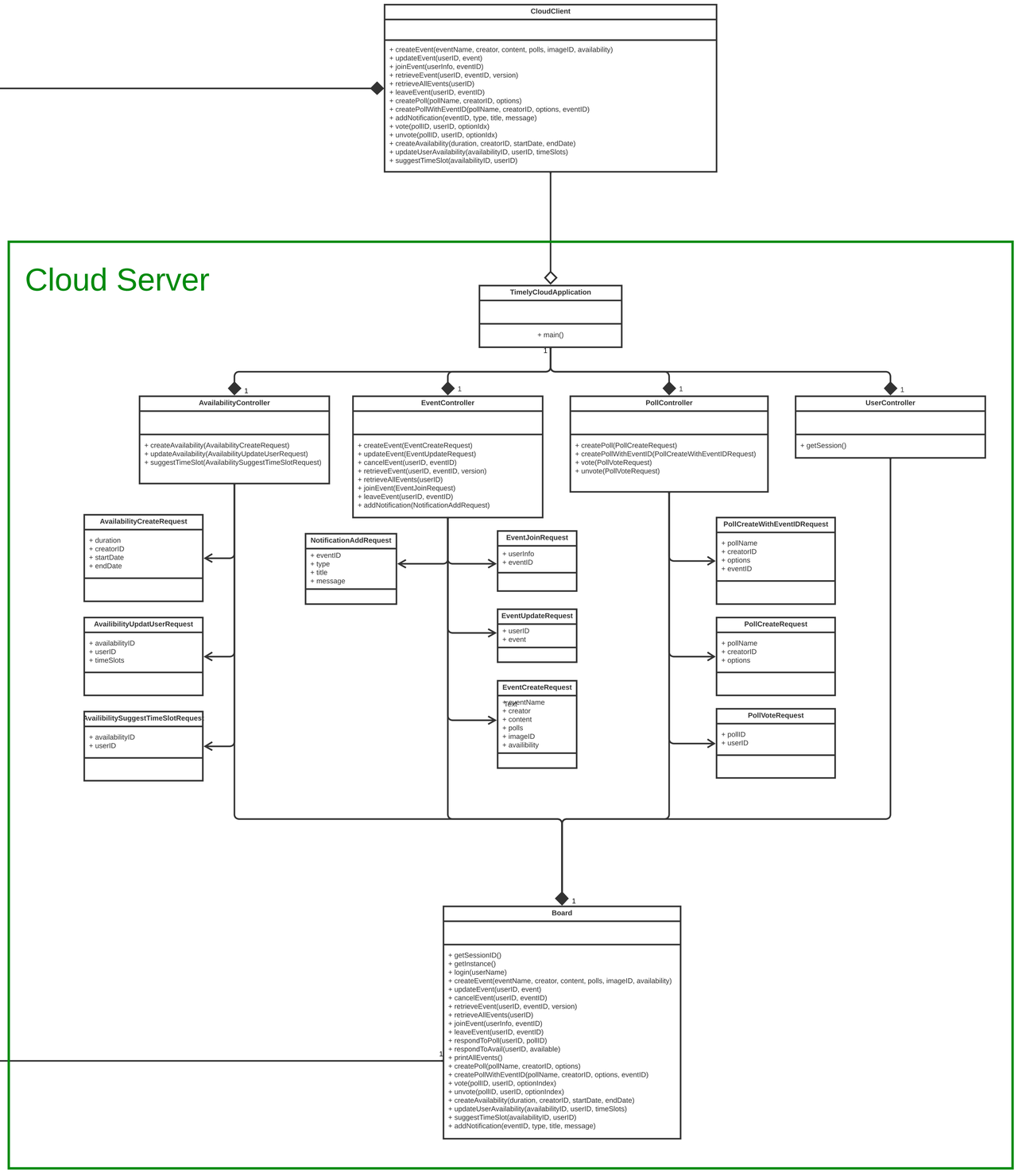
Client



Model

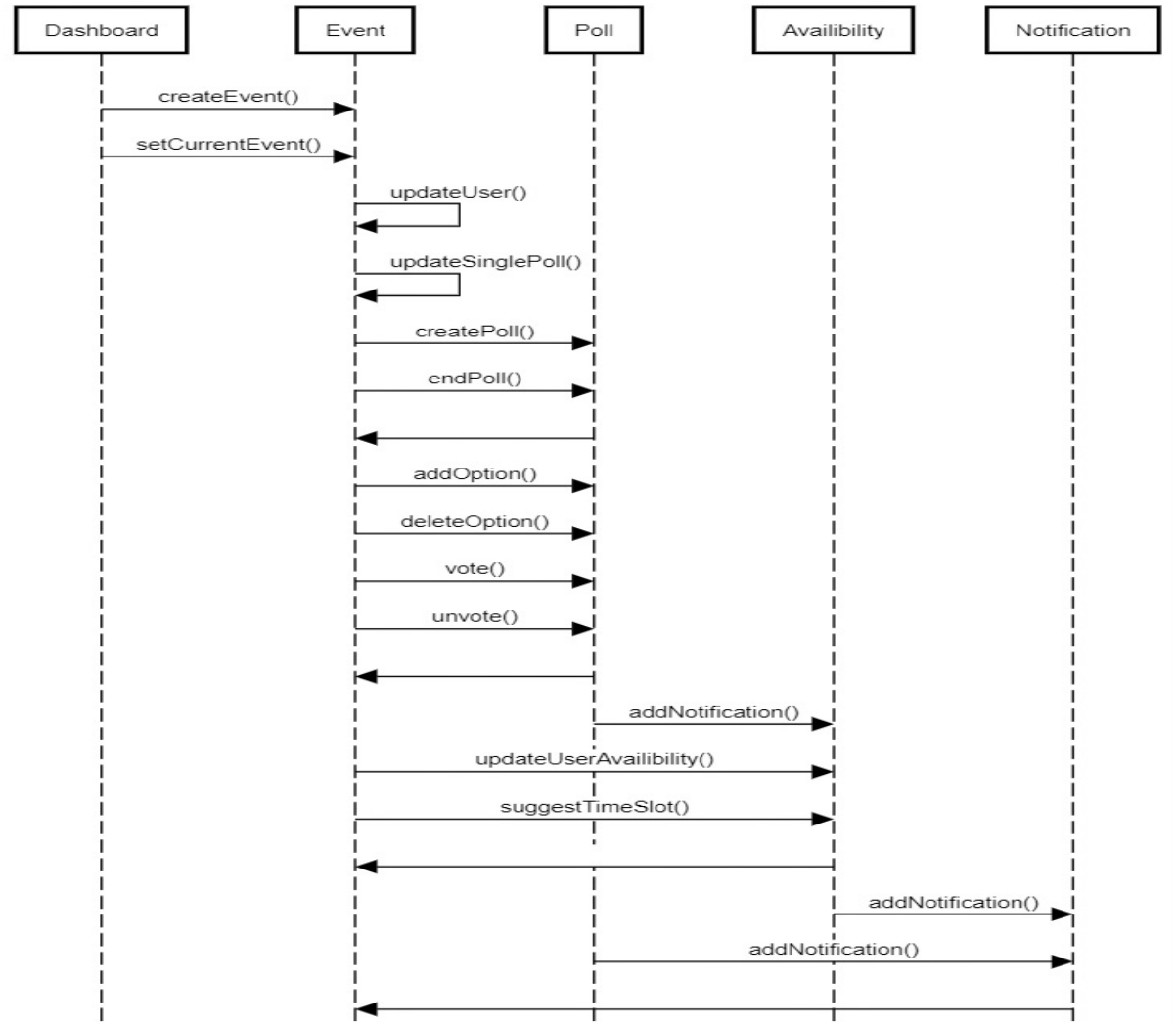


Cloud Server



Sequence Diagram

Creator



Participant