ACalendar
Software Design Specification

Version 1
10/21/2016

Qinrong Lin (snailin@cs.washington.edu)
Lisa Qun Zhang (zq0605@cs.washington.edu)
Ruoyu Mo (rettymoo@cs.washington.edu)
Tong Shen (shent3@cs.washington.edu)
Zhengyang Gao (gaoz6@cs.washington.edu)
Ziyao Huang (ziyaoh@cs.washington.edu)

Part 1 System Architecture

1.1 Modules

1.1.1 User Account Module

The first time a user seeks access to our application, our app will register for user by asking user to provide a valid email address. Then our app will send verify message to the email that user provided. The verify message contain a verify code which user has to confirm in our application login screen before expiration. Finally, with the unique username and the password provided by user, the registration succeeds. In subsequent user logins, the app will authenticates the user by means of username and password. If user forget his/her username or password, they can find it back by verify email also.

From developer's perspective, this account module needs front-end views which can guide user through the above processes, login/register controller which take user's input, and send these information to our back-end server, server-side account model that verifies the login information and retrieves user data from database and response to client app. For security issues, it is necessary to have our server to verify login password and retrieve user's personal data instead of our app, bypasses our server, directly queries from database.

Classes in this module may include:

1. LoginView: displays login page
2. SignupView: displays signup page
3. LoginController: reads login page input and sends request to verify information
4. SignupController: reads signup page input and sends request to create new account
5. client.Account: stores account info (userId, username, email, firstname, lastname)
6. server.Account: verifies login information, creates new account and asks email module to send confirmation email, updates data to Account Table in db.

1.1.2 Email Manager Module

Same as other customer-facing applications, we need emails to send updates, verifications and notifications to users. Therefore, the email module has to accomplish at least these three different category of emails:

1. Verification Emails: Every time a new user signs up, we need the user to provide a valid email address. In order to verify that address, we send out emails with verification code. So verification emails will be sent automatically from the server.
2. Notification Emails: Notification emails are sent out when there is new event invitation, new friend request changes on existing events, or an event is about to happen. All these notifications are configurable by user in settings. Users can choose whether to receive the notifications of any type of updates. This is similar to the verification email.

3. Update(Commercial) Emails: This type of email is optional for implementation. So if in the future we have more users or we would release a new version of ACalendar.

The basic email functionalities will be implemented with JavaMail API.

Classes in this module may include:

1. Email object: create different types of email with title, contents, sender's email and recipients' email.
2. Email Manager: fetch triggered by server and send out email that is passed in.

1.1.3 Event and Event Sharing Module

The Event and Event Sharing module contains the functionality of creating events, managing event, editing events, deleting events, storing/accessing events data from database (both local and server-side).

Classes in this module may include:

1. EventView: Display events title in calendar, event page with detailed information.
2. EventOperationController: accept user input and convert it into commands in either server.Event or EventSharing.
3. EventManager: locally store all events of user.
4. client.Event: contain event related information.
5. server.Event: adding/deleting/updating Event Table in db. Public events will be accessible to all users on our server.
6. EventSharing: Update EventSharing Table in db, which involves sending event invitation, accepting/declining received invitation. Private events are could only be shared to owners' friends and only invited friends have the permission to join private events.

1.1.4 Friend Manager Module

Friend manger Module will perform 3 main functionality: search friend, send friend request and accept friend requests.

1. Search friend: User input a friend username. The server will fetch that user from user table in database.
2. Send friend request: User hit "add friend" button after input the username. In database, one tuple of (sender userid, recipient userid) will be added into friend table.
3. Accept friend request: If the recipient accepts the request, add a tuple (recipient userid, sender userid) will be added to the friend table.

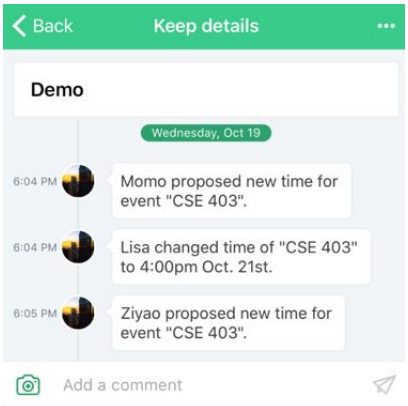
Each tuple in friend table will also include a create time. Server will constantly check the table and remove those unaccepted friend request (one way relation) if it has already expired.

Classes in this module may include:

1. FriendView: display friend list page

2. FriendController: monitor user input on friend list page
3. FriendManager: store all friends of user
4. client.Account: friends' account info
5. server.Friend: manage friends invitations, update friend relationship to Relationship Table in db

1.1.5 Message Module



For each event, a series of messages will be displayed in the format of a vertical timeline to let users know what changes have been made to that event, for example, when a participant of an event proposes a new event time or the creator of the event changes the event time. Please see the screenshot on the right for an example (this is a screenshot of the app TimeTree, but what we are aiming for in the messages section is very similar to the one displayed on the right).

We define messages to be associated with specific events. We would need to create a message and save it to our backend database when an action is acted upon a specific event, and when a user views an event, we would need to retrieve the appropriate messages stored in the database.

For messages, we need two API's to create and retrieve messages: createMessage() and getMessages(). createMessage() creates and stores a single Message object in our database, whereas getMessages() can retrieve an arbitrary number of messages of an event.

Each time a user makes a change to an event, we make an API call to createMessage() and record what the change was and store it in our database. createMessage() takes in a createMessageRequest object, which contains the message category and the event id of the event that triggered this API call. createMessage() returns a createMessageResponse object, which contains the id of the message that was just created.

When a user goes to see the details of an event, we make an API call getMessages() and retrieve the newest messages (default to 10) about recent changes made to the event and display it on the UI. In order to correctly and successfully retrieve the messages, we would need the three parameters, which are stored in getMessagesRequest objects. Each getMessageRequest object has three fields: eventId, offset, and howMany. eventId tells the database which event's messages we are interested in retrieving; offset is used to facilitate pagination when retrieving messages from the database (because for better performance, we would like to only retrieve what we need from the database); howMany tells the database how many of the messages we would like to retrieve. getMessages() return a list of Message objects, which consists of messageId, messageCategory, createdBy, createdAt, and eventId.

1.2 Database Design

The following relational database schema are designed for storing data in backend.

1.2.1 User Account Info

In order to store all the user account information, we need a database to do so. Here are a sample of our database table (more attributes if more feature add)

Account								
userId	username	password	firstname	lastname	email	verified	createdTime	iconId

1.2.2 Messages (Event timeline)

One major design decision we agreed on is to keep this two-table structure for messages. The messages table stores high-level information about each messages, for instance, what type of message a specific message is (we will talk more about the different types of messages in just a bit). The second message_attribtues table gives us more flexibility in the future if more use cases are required: when an additional use case that requires a different attribute type and value is needed, instead of having to add a column to the messages table, we can instead add a row to the message_attributes table.

Another major design decision we made revolves around whether we should give users the ability to mark a certain message as read or not. Initially we hoped to give users the ability to mark each message as read/unread, but after some discussion, we realized that this would complicate the logic our messages a lot (this has to do with the fact that each event includes potentially an unlimited number of participants). Therefore, in the end, we decided against this proposal.

Furthermore, messages are categorized into different categories. Having a set set of types of notifications enables us to store and display the messages more easily. Currently we support four types of messages, which are “NEW_TIME_PROPOSED”, “TIME_CHANGED”, “USER_JOINED_EVENT”, and “USER_QUIT_EVENT” (these will be made into a Java enum class).

messages				
id	message Category (ex: New time proposed, Time changed)	createdBy (userId)	createdAt (Date)	updatedAt (Date)

message_attributes		
messageId	attributeType (enum)	attributeValue

1.2.3 Event Managers

This table is designed for a future implementation. For each event, there is one owner and one or more managers. The current plan is only give permissions to owner of modifying the events. But in the future we want to also grant this permission to the managers. Therefore the managerUid information is stored in a separate table.

Manager	
eventId	managerUid

1.2.4 Events

The EventInfo table stores all related informations of each event. OrganizerUid is the userId of the event owner. ManagerUids is a list of the uids of the users who have the permission to modify the event.

EventInfo								
eventId	ownerUid	title	createTime	startTime	endTime	description	location	isPublic

The EventSharing table maps each event the recipients of the event invitation. Each entry represents a tuple of eventId and one of the recipients. And the sentTime column stores the time when the invitation is sent; the status column marks whether this invitation is accepted by this recipient(pending/rejected/accepted).

EventSharing			
eventId	recipientUserId	sentTime	status

1.2.5 User Relationship

The basic idea of friendship table is to store a tuple of userids for each entry. For each pair of friends A and B, there will be two entries added into the table: (A, B) and (B, A). Therefore with this table we build a graph which has users as vertices and edges between each pair of friends if both tuples exist in the table.

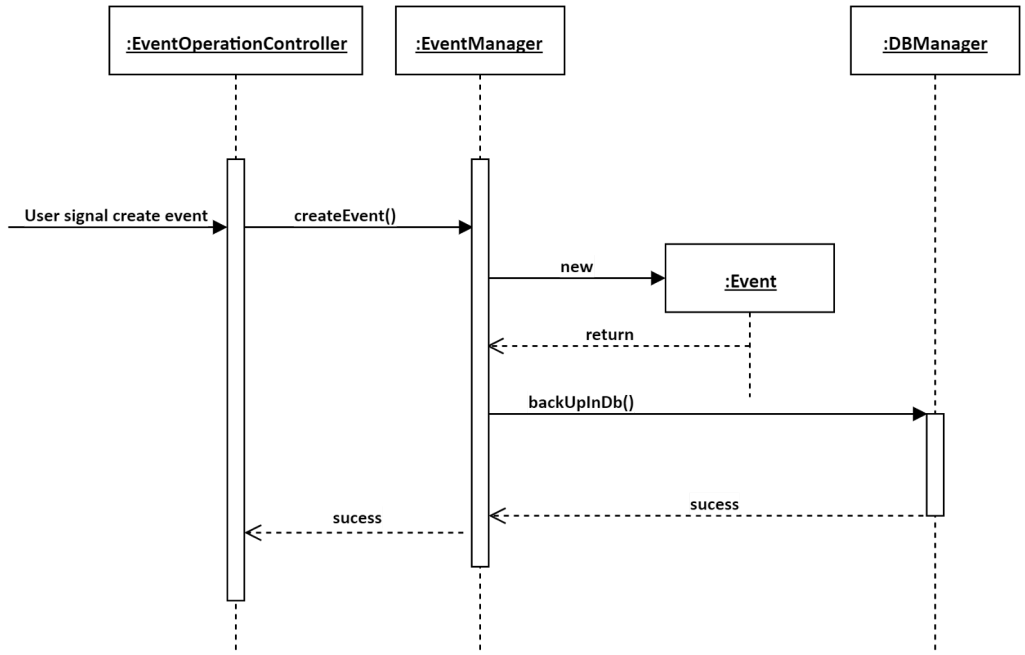
When a friendship request is sent out from A to B, we add an entry (Aid, Bid, RequestTime) into the table. And when the request is accepted by B, we add another entry (Bid, Aid, AcceptTime) into the same table. We only consider the friendship exists if both entry is present in the table.

One a while we go through the table and deleted those unaccepted friendship tuples that have already expired.

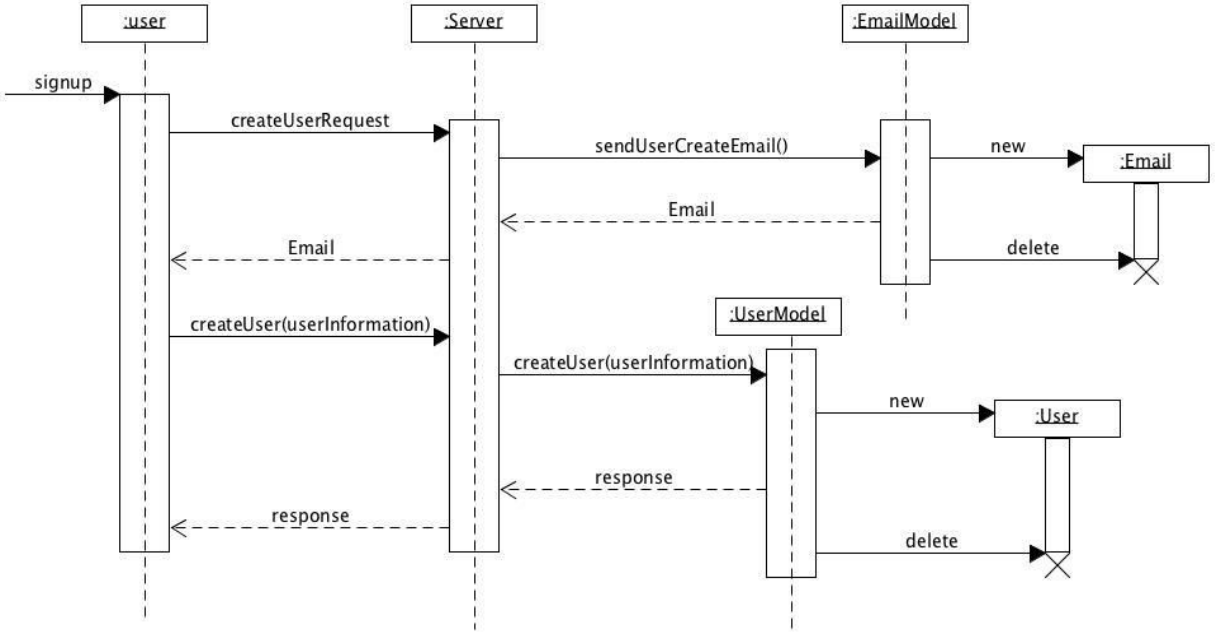
Relationship			
userId1	userId2	createTime	status

1.3 UML Sequence Diagrams

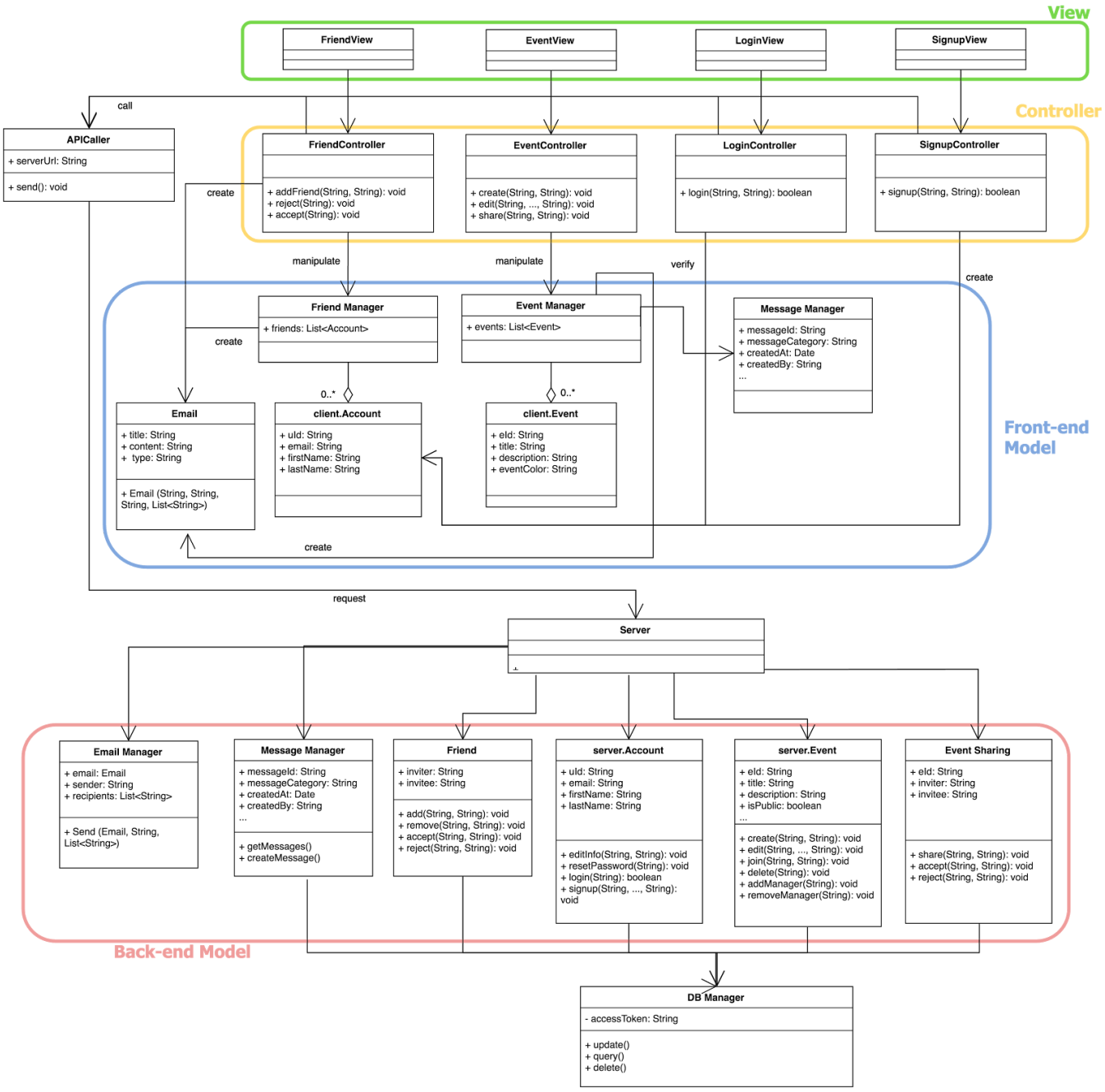
1.3.1 Create Event Sequence



1.3.2 Sign up Sequence



1.4 UML Class Diagram



1.5 Design Alternatives

Table Schema for Event Information

There are two types of events: public events and private events.

The current solution to Event Information table is to create one table of Events and one attribute in the table indicating whether this is public or not. One alternative to this design is to have two separate tables for public and private events respectively. The pros of this alternative is that it will help to reduce run time while searching for public events because you do not need to go over all events, especially when the table grows bigger and bigger. The cons of it is that if two tables are created, two separate events managers are needed and that is actually unnecessary and requires a lot of extra work.

Relationship Table Operation Design

The current solution to for saving friend invitation and acceptance is to put one entry for each. In other words, for each pair of friends A and B, there will be two entries added into the table: (A, B) and (B, A). One alternative is to store only one entry (A, B) for each pair of friends. The pros of this alternative is that the table will be much smaller (halved) especially when our user group becomes bigger and bigger. The cons of this design is that it will be extremely hard to fetch the friend list of a user. Because you have to search the table at least twice.

1.6 Assumptions

One main assumption is that our user group will not grow too big so that our database for user information and event information will stay in a reasonable size and not affect too much on run time.

Another assumption is related to user's permission on events. It is assumed that our database is stable and safe enough that will not be modified externally. As a result, it is only necessary to check the invitation table on database that some certain user was invited and has the permission to this event.

Part 2 Process

2.1 Risk Assessment

2.1.1 Risks to the cost of using APIs from a non-open source API (e.g. Google Maps) exceeds our expectation.

- Likelihood of occurring: High
- Impact if it occurs: Low
- Evidence upon which you base your estimates:

  In our ACalendar application, there is a searching feature which allows users to search on public events based on given filters. In order to enable a map view of events, we need to use Google Map APIs. Also we need to use AWS database. Google map has a limitation on the number of requests each day. While we have more and more users, it will be inevitable that we reach the limitation and need to pay for Google Maps API. Also as time pass by, our database will also grow and give rise to costs on AWS.
- Steps you are taking to reduce the likelihood or impact, and steps to permit better estimates:
  1. User multiple Google API keys so that we could have enough requests allowance during the development process.
  2. Optimize the algorithm to reduce the number of request generated.
- Plan for detecting the problem:

  Implement a detector on whether each API keys is out of free requests. If there is an error occurred due to the limitation on requests, automatically start to use the next available key.
- Mitigation plan should it occur:
  1. Apply for education discount of AWS.
  2. Add advertisement on this application and pay the fees with earnings from advertisement.

2.1.2 Risks that users joined the event which they do not have permissions to.

- Likelihood of occurring: Medium
- Impact if it occurs: High
- Evidence upon which you base your estimates:

  In our ACalendar application, one of the key ideas is that users could create events as private. For the private events, only users with permission can join. We do not want any unauthorized user somehow managed to take a detour and joined the event.
- Steps you are taking to reduce the likelihood or impact, and steps to permit better estimates:
  1. In the database we have a table storing users who was invited to this event.
  2. Every time the event is accessed by a user, we go to the database and check whether this user was invited.
  3. Reject the request from uninvited users.
- Plan for detecting the problem:
  1. Implement a verification tool with in the event module that constantly check each event to make sure that each person in this event is authorized by the owner.

2. Enable reporting features for event owners to report on unauthorized participants.

- Mitigation plan should it occur:
  3. Check authorization in multiple levels, not only from database level, but also in APIs.
  4. Have protections on database.

2.1.3 Risks with the hardware and software (the development platform) chosen to perform project development. e.g., can this hardware and software handle the workload required to complete the project?

- Likelihood of occurring: Low
- Impact if it occurs: High
- Evidence upon which you base your estimates, such as what information you have already gathered or what experiments you have done:

We will be using Android Studio as our development platform. Android Studio is the official tool for building Android application and our application is a fairly simple project. As a result, we believe that it will be capable of satisfying all requirements we have on ACalendar. Also, Android Studio platform is powered by IntelliJ IDEA, and all backend implementation will be coded in Java, which we all very familiar with. Therefore, I do not think Android Studio is very likely to broke during the implementation of our project.

However, if for some reason, we end up facing this challenge, it will be very costly to fix it since we have to abandon this IDE and switch to a new one. That will cost both time and workload.

- Steps you are taking to reduce the likelihood or impact, and steps to permit better estimates:
  1. Have a solid plan in advance.
  2. Improve coding skills of each team member.
  3. Learn about the building process of Android Studio thoroughly.
- Plan for Detecting the problem:
  Run performance test constantly.
- Mitigation plan should it occur:
  Research on substitution of Android Studio in advance.

2.1.4 Risks to the inevitable absence of team member(s).

- Likelihood of occurring: Medium
- Impact if it occurs: Low
- Evidence upon which you base your estimates, such as what information you have already gathered or what experiments you have done:

We have assigned each module of our project to different team members. And when it comes to the point we integrate all parts together, we may need to call on meetings to talk about compatibilities. However, we have six members and if one is absent inevitably, the other five could always take the responsibility of that member. Also we have build up shared docs and group chats for communications at any time.

- Steps you are taking to reduce the likelihood or impact, and steps to permit better estimates:
  1. Make sure each member is clear with the requirements
  2. Make sure each member has created detailed documentations
  3. Have meetings constantly and ensure compatibility requirements are clear.
- Plan for detecting the problem:

Keep in touch with team members. And each week have the product manager assign jobs to each team member and be sure that each team member is clear on his responsibility.

- Mitigation plan should it occur:

Make sure that there is someone to finish the job of the absent member. And make sure each person has created a detailed and complete documentation on all works he has done.

2.1.5 Risks that the project schedule may exceeds 6 weeks.

- Likelihood of occurring: Medium
- Impact if it occurs: High
- Evidence upon which you base your estimates, such as what information you have already gathered or what experiments you have done:

Since we have not started on development part, we could not tell in anyway how the workload would be. So there is great chance that we have set some requirements too complicated and is impossible to be finished by the deadline.

And if the problem does occur, we may end up with a incomplete application that is not functioning at all.

- Steps you are taking to reduce the likelihood or impact, and steps to permit better estimates:
  1. Have a solid plan/schedule in advance.
  2. Classify each feature as required and not required.
  3. Implement the required features first and not required features later if we have time.
- Plan for detecting the problem:

  Keep all the works up with schedule. If any part of the procedure seems too much to be finished in time, then we know the requirements has to be modified.
- Mitigation plan should it occur:

Cut unnecessary requirements to ensure the basic functionalities.

2.1.6 How this has changed since the SRS:

After we came up with the basic requirement specification, we have a general idea on what kind of job is need in this project. Also for each module in the project we have specified how it should be done technically. Therefore we have a more detailed idea on what risks we may be facing in the future.

2.2 Project Schedule & Team Structure

- Product Manager: Zhengyang Gao
- Each member of our group will be software developers (meaning that we will not have dedicated developers and testers). For each major module of ACalendar, we have agreed on the following schedule:
  - User accounts & login (Ruoyu Mo, finish by Oct.24)
    - Create account
    - Reset password (by verification code via email)
  - Event-related modules (event creation/edit/sharing, etc)
    - Calendar view of all events for a user (Qun Zhang, finish by Oct.24)
    - Event dashboard for each event (Tong Shen, finish by Oct.31)
      - ➢ Post proposals or updates(input box and display)
    - Event Pool (Ziyao Huang & Qinrong Lin, finish by Oct.27)

        - ➢ Searching an event
        - ➢ Marking an event on Google Maps
    - User Dashboard
        - Messages module (Zhengyang Gao, finish by Oct.27)
        - Emails module (Tong Shen, finish before Oct. 24)
    - Friends-related modules (Ruoyu Mo, finish by Nov.5)

The four outer modules listed above should not depend on each other too much, and therefore we can put different group members on different tasks. The assigned member(s) of each module are listed above.
Within each module, cohesion between each sub-module starts to surface, which is why we did not further granulate the task assignments.
While implementing the backend of our app, we will also be implementing a rough UI so that we can test our code as we go. After finishing implementing the backend, we will focus on beautifying/improving the frontend UI. We expect this to take about 2 weeks (finish by Nov.15th).
After finishing the majority of our app, we will set up an extensive bug bash. We expect this to take less than a week (finish by Nov.20th).
If everything goes smoothly, we will have about two weeks left in the end, which we can use to implement some of our stretch goals.
We plan to meet and sync group members' progress two to three times a week. Usually we meet on Tuesday afternoon, Wednesday afternoon, and Thursday afternoon.

2.3 Test Plan

The listing below identifies those items (use cases, functional requirements, non-functional requirements) that have been identified as targets for testing. This list represents what will be tested. Details on each test will be determined later as Test Cases are identified and Test Procedures developed.

2.3.1 Test Requirements

- Data and Database Testing
    - Verify access to every Database Table
    - Verify simultaneous record read accesses.
    - Verify lockout during ACalendar (database data) updates.
    - Verify correct retrieval of update of database data
- Performance Testing
    - Verify response time to access external Google Map system.
    - Verify response time to access external Email subsystem.
    - Verify response time for login.
    - Verify system response when ACalendar Database at 90% capacity.
    - Verify response time for submittal of registration.

- System Testing
    - Verify system response when loaded with 100 logged on clients.
    - Verify system response when 30 simultaneous clients access to the ACalendar
    - System Testing

- Verify Login Use Case
- Verify Add Friend Use Case
- Verify Share Events Use Case
- Verify Edit Events Use Case
- Verify Create Event Use Case
- Verify Switch private/public Mode Use Case
- Verify Request Friend List Use Case

2.3.2 Test Strategy

- Data and Database Testing
  The databases and the database processes should be tested as separate systems. These systems should be tested without the applications (as the interface to the data). Additional research into the DBMS needs to be performed to identify the tools / techniques that may exist to support the testing identified below
  - Test Objective: Ensure Database access methods and processes function properly and without data corruption.
  - Technique:
    - Invoke each database access method and process, seeding each with valid and invalid data (or requests for data).
    - Inspect the database to ensure the data has been populated as intended, all database events occurred properly, or review the returned data to ensure that the correct data was retrieved (for the correct reasons)
  - Completion Criteria: All database access methods and processes function as designed and without any data corruption.
- System Testing
  Testing of the application should focus on any target requirements that can be traced directly to use case. The goals of these tests are to verify proper data acceptance, processing , and retrieval. This type of testing is based on upon black box techniques, which verifying the application by interacting which the application via the GUI and analyzing the output.
  - Test Objective: Ensure proper application navigation, data entry, processing, and retrieval.
  - Technique:
    - Execute each use case using valid and invalid data.
    - Expected results occur when valid data is used
    - Should be able to appropriate error or warning messages when invalid data is use.
  - Completion Criteria:
    - All planned tests have been executed
    - All identified defects have been addressed
- Performance Testing
  Performance testing measures response times, transaction rates, and other time sensitive requirements. The goal of Performance testing is to verify and validate the performance requirements have been achieved.
  - Test Objective: validate System Response time for designated transactions under normal anticipated volume and worse case volume
  - Technique: Modify data files to increase the number of iterations each transaction occurs.

        - Completion Criteria:
            - Single Transaction : successful completion of the test without any failure and within the expected time.
            - Multiple transactions:: successful completion of the test scripts without any failures and within acceptable time.
    - Track bug : our group will use github to track bugs and fix them.

2.4 Documentation Plan

Since our app is aimed at everyday users, we plan to provide users with a brief tutorial (either printed user guide or a little video demo) of how our app works and what its highlights are at the end of the project.

2.5 Coding Style Guidelines

Code quality is of significance. We will periodically gather together and review each other's code in person to ensure each team member follows an acceptable coding guideline.
Below are some links to some pre-existing coding style guidelines:

Java -- https://google.github.io/styleguide/javaguide.html
XML -- https://google.github.io/styleguide/xmlstyle.html

Design Changes and Rationale (Beta Release):

- For this beta release, when signing up for a new account, we will not send a verification email to the email address provided (we will send an email saying that this user has successfully created a new account). For the next release, we are still debating whether or not we should send an email to verify the email address provided by users. We will have come to a conclusion by this Monday.
- For the risk of "users joining an event which they do not have permission to join", we changed this impact level to "High".
- We articulated our testing plans more clear. Each time a developer pushes, all the tests will be run by Travis. For tests, we would not be talking to the database each time we test. Instead, we would use a mocking library (Mockito) to mock out the code that talks to the database and only tests the logic. This way, all unit tests can be run locally.
- For db interactions, we decided not to use the Java Hibernate Library and went with the set of APIs provided directly by AWS.
- We also made some changes to the timeline regarding the development of ACalendar. Specifically:
    - For this beta release, we didn't get to implementing the "Forgot password" functionality.
    - For this beta release, we didn't get to implementing the "Post proposals or updates" functionality. We are planning on doing this right after the beta release and hopefully we can get it done before the next release.
    - For this beta release, we didn't get to implementing the "Search public events" and the "Marking an event on Google Maps" functionalities. We are planning on implementing

both of these functionalities after the beta release and hopefully can get them done before the next release.

- For the notifications part, we decided not to use push notifications. Instead, for each event, we will maintain a list of all messages (notifications) posted to this event.
- For the friends module, we decided not to use the Facebook login APIs and instead developed our own login module. The reason we did it this way is because we thought this would be a good opportunity for us to learn about how login is done.

No other major changes to our SDS document.