

Meet.me

Dashboard Module Specifications

CONTENT

OBJECTIVE	4
TEAM CONFIGURATION	4
Team Lead:	5
Session Manager:	5
Summary Logic:	5
Telemetry:	6
Persistence:	6
DEPENDENCY DIAGRAM	7
MODULE DIAGRAM	8
Server Boots Up:	8
Client Joins the Meeting:	8
Client Departure:	9
End Meeting:	10
Summary And Telemetry Analytics Retrieval During The Session:	11
Connection Error:	12
MODULE TESTING	13
SESSION MANAGEMENT SUBMODULE	16
Overview	16
Objectives	16
Design Analysis	17
Class Diagram	18
Interfaces	20
Further Plans	21
SUMMARY LOGIC SUBMODULE	22
Overview	22
Objective	22
Class Diagram	23
Interface	23
Design Analysis	24
Creation of Summarizer	24
Summarizer	24
Algorithm	25
Stretch Code	26
TELEMETRY SUBMODULE	27
Overview	27

Objective	27
Design Analysis	28
Class Diagram	28
Interfaces	29
Further Plans	29
PERSISTENCE SUBMODULE	30
Overview	30
Objective	30
Design Analysis	30
Class Diagram	31
Interfaces	32
Further Plans	32
FUTURE SCOPE	33

OBJECTIVE

- Provide a session manager who will manage all the relevant details about a session both on the server as well as client-side.
- Provide a summarizer that will summarise the meeting held with appropriate filtering.
- Provide a telemetry analyzer that will analyze the data transferred for the discussions held on the server and present it in a neat and efficient manner.
- Provide a persistence interface that will handle storing of files and images that are generated by the telemetry analyzer and summarizer.
- The dashboard keeps track of users in the session and all the relevant information such as usernames, streams, session passwords, etc.
- Maintain the session details such as users in the session, session password, user and IP mappings, etc.

TEAM CONFIGURATION

Team Lead:

- **Handled By:** Siddharth Shah
- **Role:**
 - To design the abstract design for the dashboard module.
 - Handle the deadlines for the dashboard module and accordingly design time schedules for other team members as well.
 - Integrate the module with other modules and handle internal team conflicts or design choices.
 - Regression testing and E2E testing for the module.

Session Manager:

- **Handled By:** Rajeev Goyal
- **Role:**
 - Runs as controller on the server as well as on the client-side
 - Initialize all managers in other modules with appropriate dependencies
 - Act as an interface between all the submodules in the Dashboard module.
 - Maintain the details of all the users in the session
 - Set the users list for the Networking module to broadcast

Summary Logic:

- **Handled By:** Sairoop Bodepudi
- **Role:**
 - Design an algorithm to summarise the contents discussed during the discussion
 - Efficient model the summarizer to run efficiently whenever client asks for summary.

Telemetry:

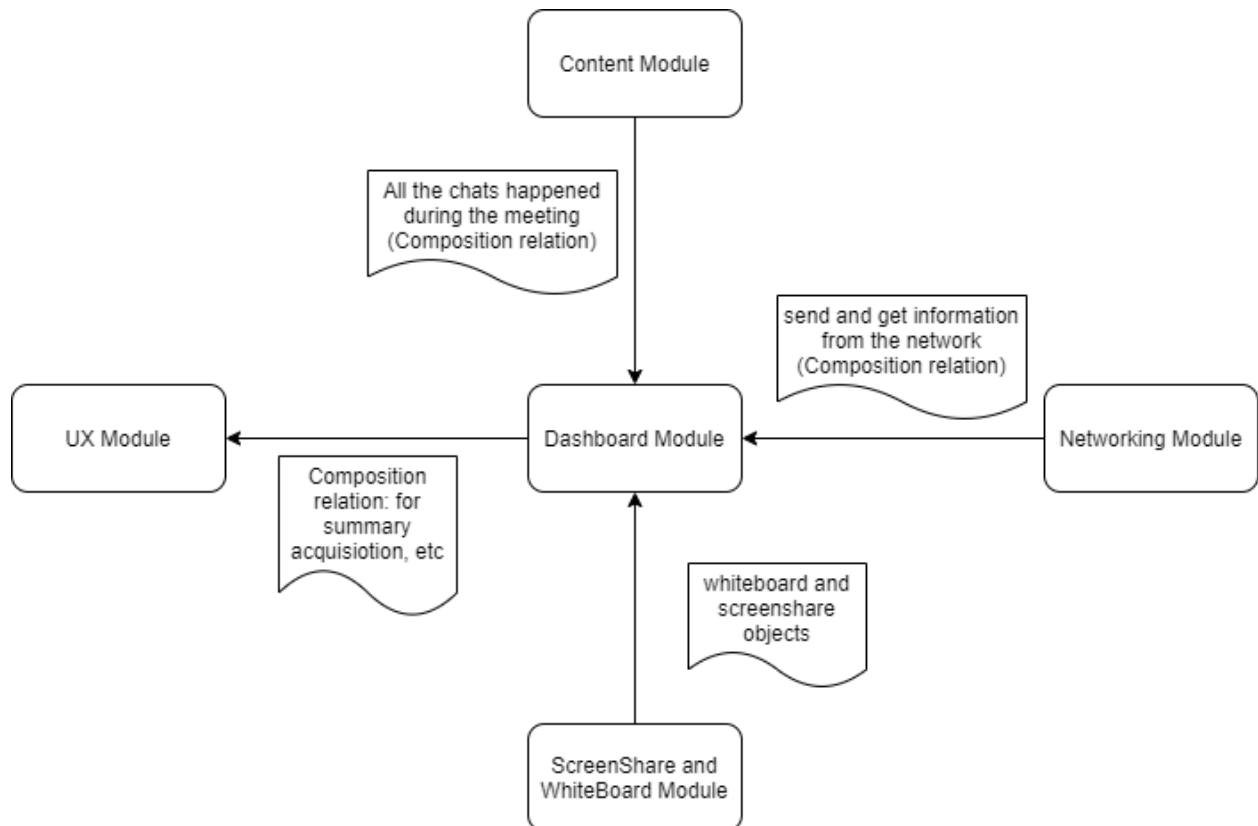
- **Handled By:** Harsh Parihar
- **Role:**
 - Run analysis on the data of all discussions done on the server
 - Run analysis on the data of a particular discussion
 - Analysis telemetrics include users/discussion, number of total discussions, chats/user for ongoing session, the time any user spends in the meeting.

Persistence:

- **Handled By:** Parmanand Kumar
- **Role:**
 - Store the summary and telemetry analysis at the end of the meeting efficiently and effectively.
 - Create graphs for the telemetry analytics and store them as in PNG format.

DEPENDENCY DIAGRAM

- The dashboard will depend on all the modules present in the project.
- Following is the dependency diagram which states the dependency relationships and the information passed during in the relation



- As the dashboard module runs as a controller module, it has to initialize almost all the modules in the software

MODULE DIAGRAM

The dashboard modules have different flow tracks based on different events, all the events and the data for the event are depicted in the following diagrams.

Server Boots Up:

When the server is started the session object is initialized on the server-side by the session manager and the session manager also regulates the flow of IP address and port to connect to the server.

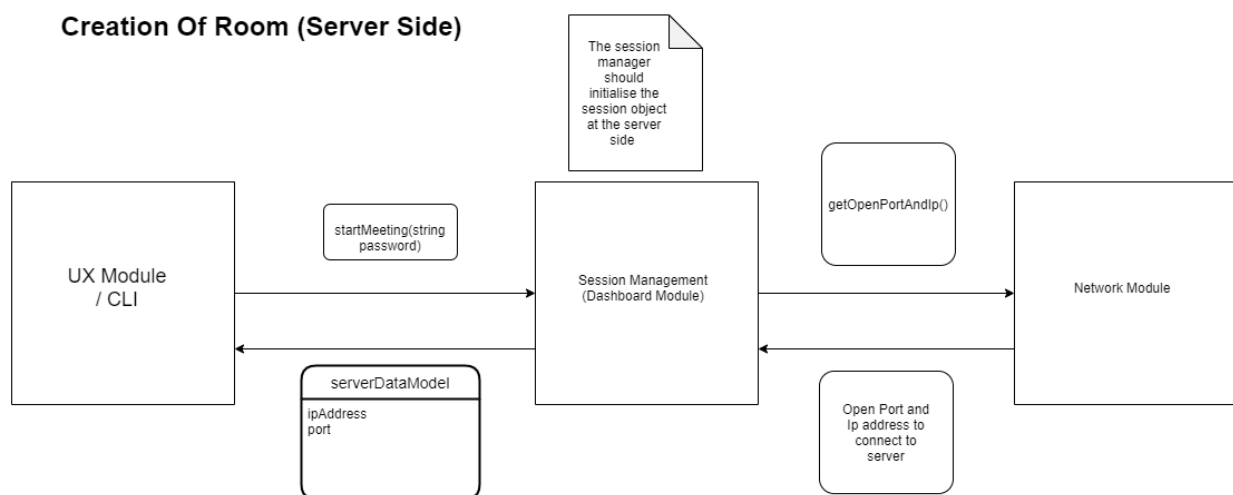


Figure 1. The data flow diagram whenever server boots up

Client Joins the Meeting:

When a client joins the meeting, the dashboard on the client-side receives the clientJoinModel from the UX which is transmitted to the Network Module on the client in order to send it to the server. The server-side dashboard module creates a user object based on the client details that are received and then adds this object to the session object maintained on the server-side. The telemetry submodule in the dashboard module must run its analytics on the changed data. This new user object is broadcasted to all the users who were present before the arrival of the new user and the server session object is sent to the new user. The received data at the client-side will be notified to the UX module from the dashboard module at the client-side.

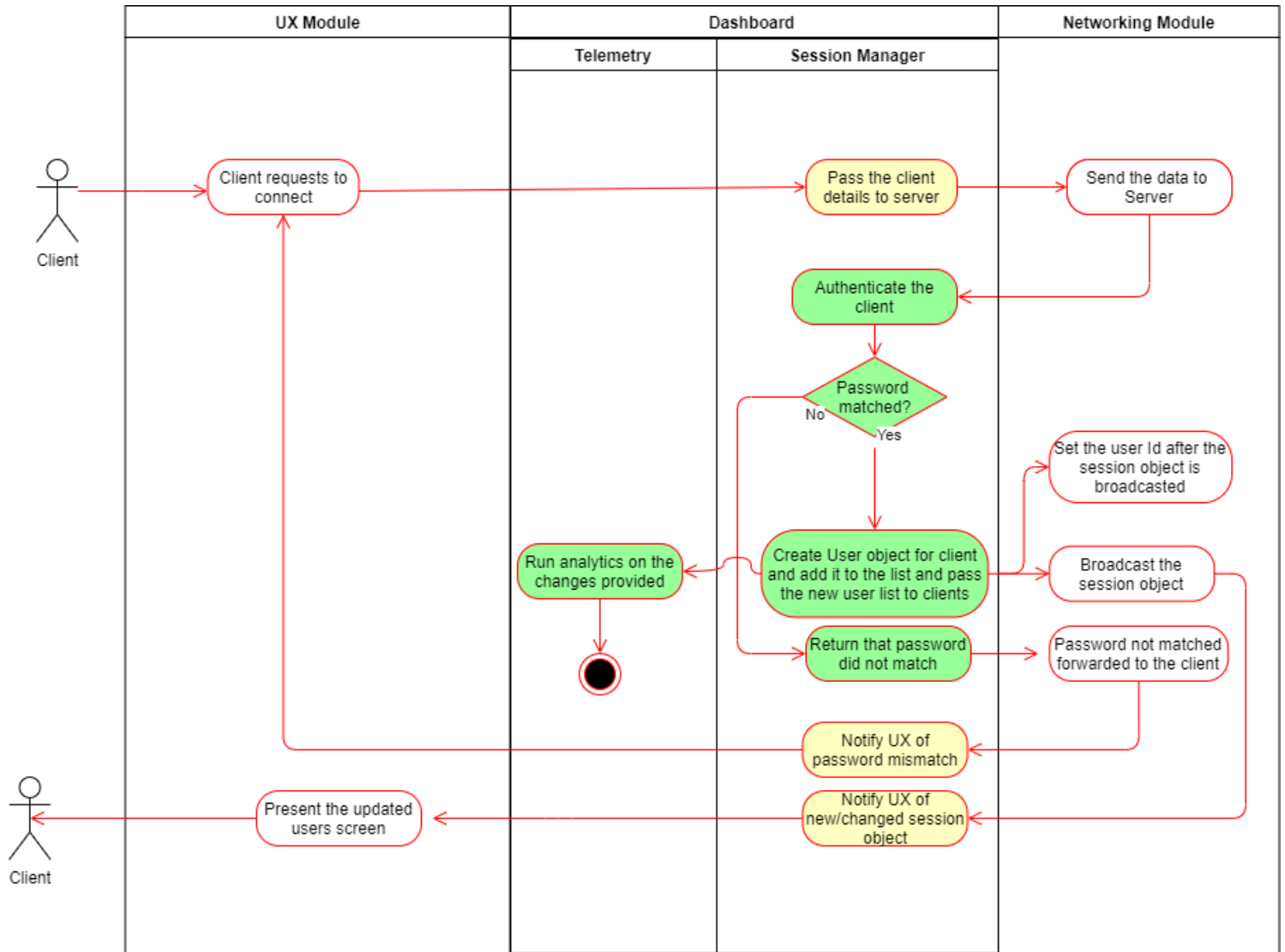


Figure 2: Activity diagram for client join event

Client Departure:

Whenever the client leaves the meeting, the UX will run the `clientLeft()` function from the session manager's interface for UX. This event will be passed on to the server via networking and then on the server side the dashboard module will remove the user from the session object maintained at the server-side and the same will be notified to the telemetry submodule via publisher-subscriber relationship. Then this new session object will be broadcasted to remaining clients and thus notified to the UX on other clients for the appropriate changes.

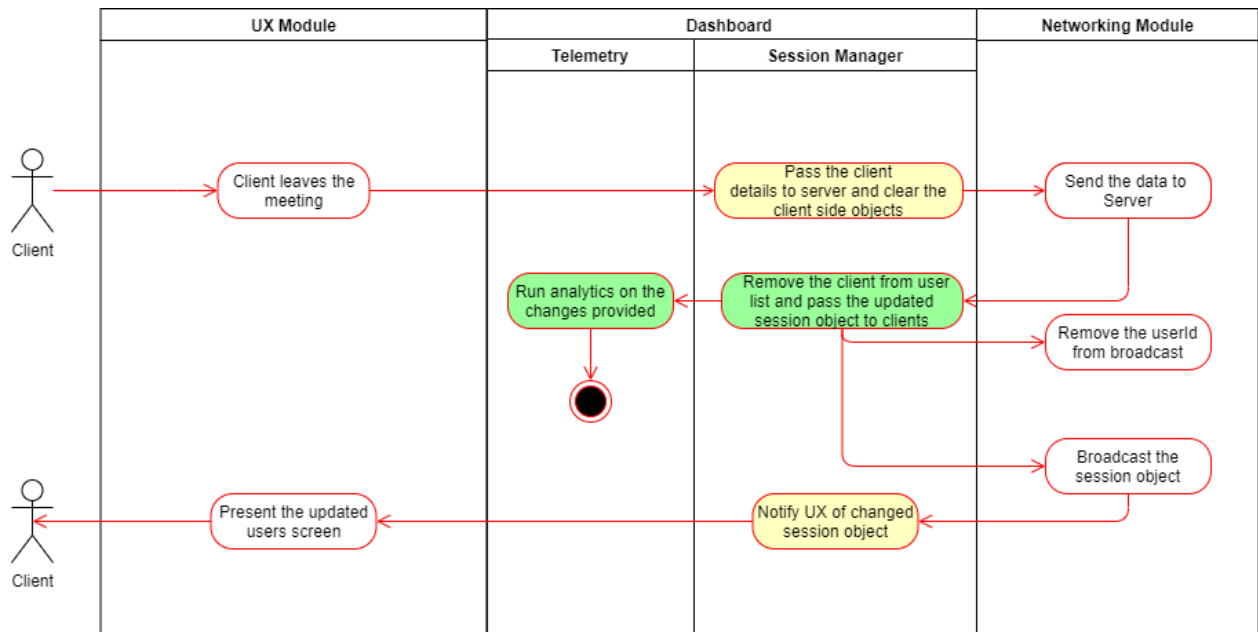


Figure 3: The activity diagram for the event when some client departs

End Meeting:

When the meeting ends, the session manager should clear up the session object at the client-side. The session manager at the server-side should gather all the chats from the Chat module and provide them to the summary logic submodule and the telemetry for a summary of the meeting and analyzing the data transferred during the meeting. This summary and analysis should be given to persistence in order to store as text and png files on the host/server.

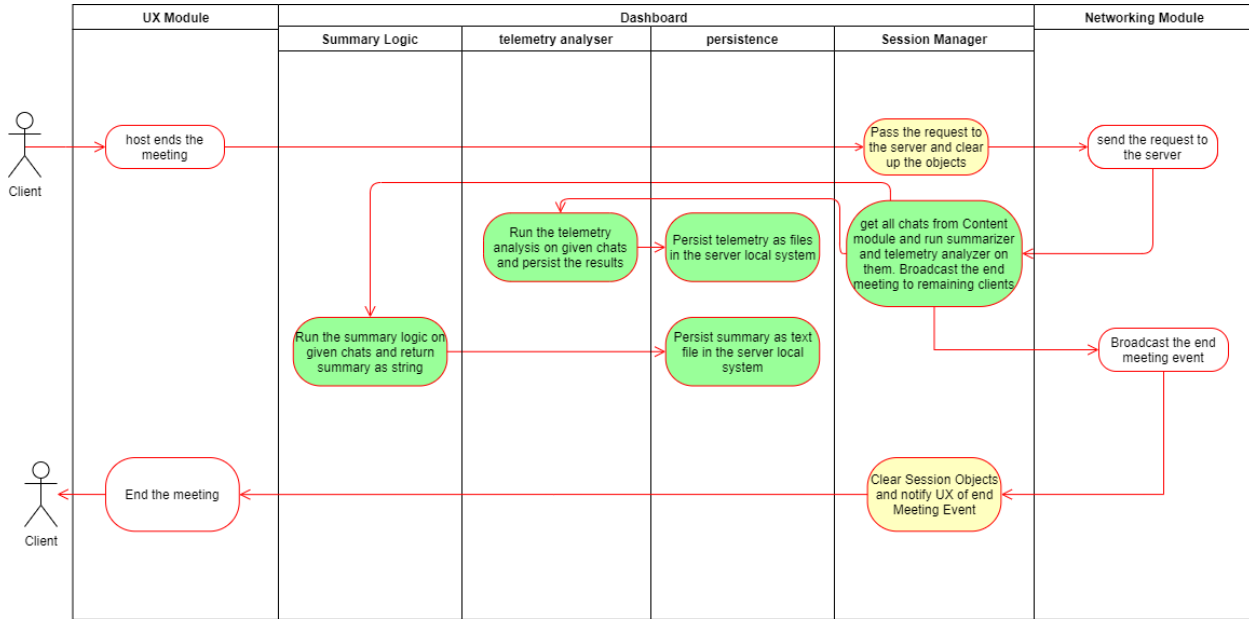


Figure 4: a) Activity Diagram for End Meeting Event.

Summary And Telemetry Analytics Retrieval During The Session:

When some client requests the summary for the ongoing discussion, the dashboard will send the request from the client-side to the server-side, where it will run the summary logic on all the chats in discussion and return the generated summary to the client who requested it.

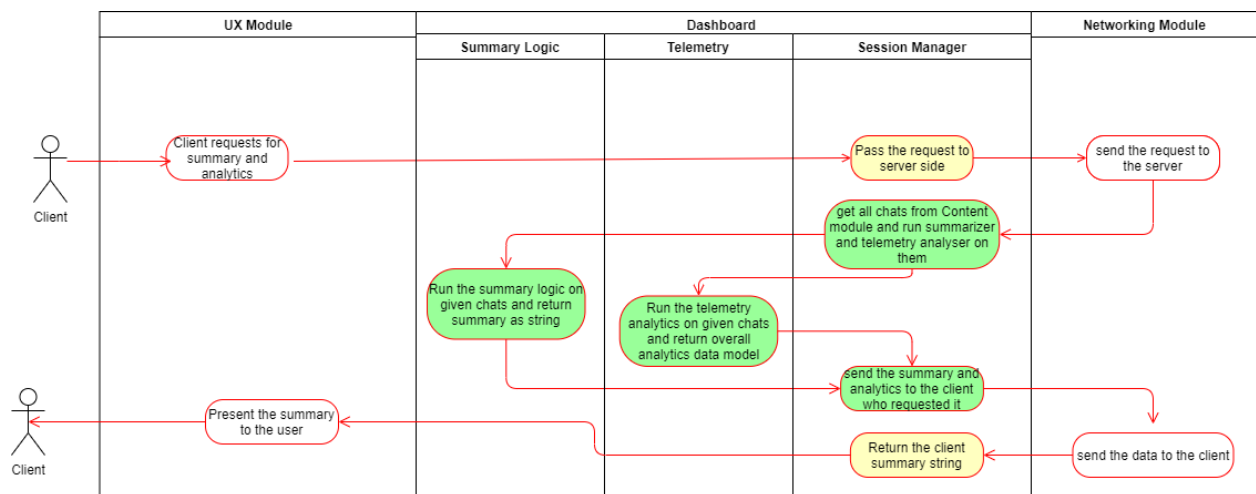


Figure 5: a) Data flow and b) Activity Flow diagrams when some client requests for summary

Connection Error:

When the server is unable to send a client a message, the networking module will try to connect with the client n times and even if then the connection is not established, the client is marked as if it left the meeting. A similar procedure is applied if client is unable to connect to the server, a connection to server is tried and if not acquired then an EndMeeting Event executed at the client side

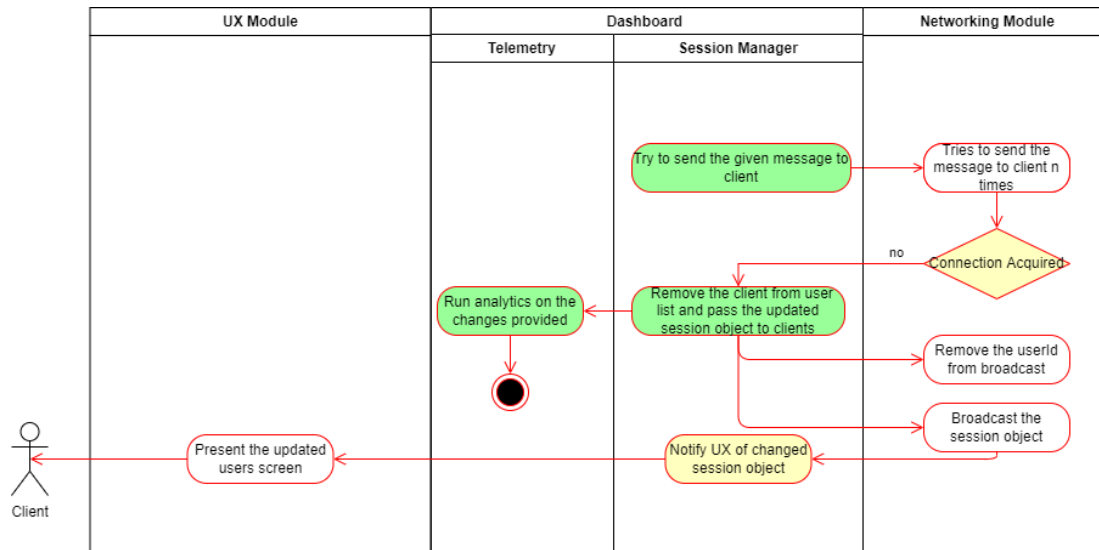


Figure 6: a) Activity Flow diagram when server is unable to connect to some client

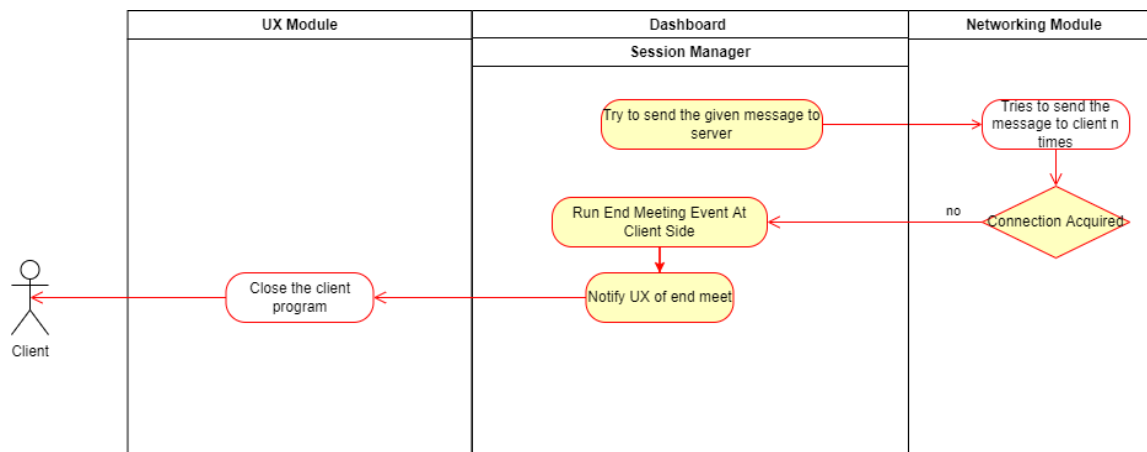


Figure 6: b) Activity Flow diagram when client is unable to connect to the server

MODULE TESTING

The module will be tested at each end point where the Dashboard module either interacts with other modules or Dashboard module stores something on the local system. To simulate how the dashboard would interact with other modules, a fake or test version of the modules which would be contacted by the Dashboard module are implemented. These fake versions would help in testing if the other modules were contacted as expected. These fake versions of other modules are provided in the constructor of SessionManager as SessionManager acts as a controller submodule for Dashboard Module and SessionManager is the only module which interacts with other modules on behalf of Dashboard Module. The parameterized constructor of session manager will only be used in the event of testing. A parameterized constructor was used because, Testing Module was using the Dashboard Module and to avoid cyclic dependency, an instance was passed in the constructors. Some features of the module had to be tested manually because of variable outputs. Following is the table which shows test cases:

Test Id	Event	Test	Expected Output	Status
1	Get IP And Port (Server Side)	Network Module returns invalid IP and port string	Return null to test UX module	Passed
2	Get IP And Port (Server Side)	Network returns valid IP and Port string	Return Meeting Credentials instance to test UX module	Passed
3	Client Arrival (Client Side)	Client provides valid IP and Port	Return true to Test UX	Passed
4	Client Arrival (Client Side)	Client provides invalid IP and Port	Return false to Test UX	Passed
5	Client Arrival (Server Side)	Network module informs dashboard module with the details of user sent from client dashboard to server dashboard	Update Network Module of new client Id and broadcast session object to all clients	Passed
6	Client Arrival (Server Side)	Network module informs dashboard module with the details of user sent from client dashboard to server dashboard	Broadcasts user object to all client where user object contains valid userId	Passed
7	Client Arrival (Client Side)	Network module informs dashboard module with the details of session sent from server dashboard to client dashboard	All the client UXes (old and new) should be notified of session object	Passed
8	Client Departure (Client Side)	When client leaves the meeting	Send an event of remove client with client details to network module to send it to server dashboard	Passed

9	Client Departure (Server Side)	Client Left event with the client details are provided to the server dashboard by Network module	Remove client id from network and session object and broadcast the session object via Network	Passed
10	End Meeting (Client Side)	When client ends the meeting	Send end meet event to network module for sending it to server dashboard	Passed
10	End Meeting (Client Side)	When Network module informs with object which indicates end meeting event	Update UX with meeting end event	Passed
11	End Meeting (Server Side)	When a end meet event is received from Network module	Saves the summary in local system	Passed
12	End Meeting (Server Side)	When a end meet event is received from Network module	Saves the telemetry analytics in local system	Passed
13	End Meeting (Server Side)	When a end meet event is received from Network module	Broadcast end meeting event to all clients via networking	Passed
14	Summary Retrieval (Client Side)	When UX wants to retrieve summary	Send an event of get summary with client details to network module to send it to server dashboard	Passed
15	Summary Retrieval (Server Side)	When a request of summary retrieval is received from Network and Chat Context is null (Chat Module Initialisation fails)	Broadcast summary as null to all clients with client details who asked for the summary	Passed
16	Summary Retrieval (Server Side)	When a request of summary retrieval is received from Network and no chats have happened until now	Broadcast summary as empty string to all clients with client details who asked for the summary	Passed
17	Summary Retrieval (Server Side)	When a request of summary retrieval is received from Network and small amount of chats have happened until now	Broadcast summary as non-empty string to all clients with client details who asked for the summary	Passed
18	Summary Retrieval (Server Side)	When a request of summary retrieval is received from Network and large amount of chats have happened until now	Broadcast summary as non-empty string to all clients with client details who asked for the summary	Passed
19	Summary Retrieval (Client Side)	When a broadcasted summary is received from network	Update UX of client who requested summary with modified summary	Passed
20	Analytics Retrieval (Client Side)	When UX wants to retrieve analytics	Send an event of get analytics with client details to network module to send it to server dashboard	Passed
21	Analytics Retrieval (Client Side)	When a broadcasted analytics is received from network	Update UX of client who requested analytics with modified analytics	Passed
22	Analytics Retrieval	When a request of analytics retrieval is received from Network	Broadcast analytics object to all clients with client details of	Passed

	(ServerSide)		the client who asked for the summary	
23	Module Initialisation (Client Side)	When the client dashboard starts	Initialise whiteboard, screen sharing and content modules	Passed

SESSION MANAGEMENT SUBMODULE

Overview

The UX layer sits at the top of the other module and users interact via this layer only. Below the UX layer is the Dashboard Module layer that handles UX's interaction with other modules. The Session Manager is an important submodule of the dashboard. It will create sessions at the beginning of the meet and manage these entities during an ongoing meeting. A server session consists of all the users currently in the meeting. The session manager is also responsible for asking the telemetry service to create/update statistics and the summariser to fetch the summary of the meeting.

Objectives

The Session manager is divided into two parts; server and client. Both of these should be able to complete the given tasks when the following events occurs:

The Server Session Manager

- **Host creates the meeting:** The Server Session Manager should give the UX the ports and IP address required to join the meeting.
- **User joins the meeting:** The Server Session Manager will create an ID of the user by incrementing the counter. The server session manager will then add the user to the current session and broadcast the new user to all the clients. Along with that, it will also notify the Telemetry module to generate the statistics accordingly.
- **User asks for getting the summary:** The Server Session Manager fetches the summary from the summarizer module. After fetching the summary is sent to the client side session manager of that particular user.
- **User asks for getting the analytics:** The Server Session Manager fetches the analytics from the Telemetry module. After fetching, the analytics is sent to the client side session manager of that particular user.
- **User leaves the meeting:** The Server Session Manager removes the user from that particular session. The Telemetry module is notified of this change.

- **Last user leaves the meeting:** The Server Session Manager checks if only one user is present and then asks the summary and telemetry modules to save the summary and analytics. The server UX is then notified of the meet end event.

The Client Session Manager

- **User joins the meeting:** The Client Session Manager will first verify the credentials of the incoming client and send the username of the client to the server side upon successful verification of the credentials. It will then receive the update session data from the server session manager and update its own session data. The Client UX module will then be notified.
- **User asks for getting the summary:** The Client Session Manager will first send the data of the user to the server side. After getting the summary from the server side, the Client UX is then notified of the summary creation via invoking an event.
- **User asks for getting the analytics:** The Client Session Manager will first send the data of the user to the server side. After getting the analytics from the server side, the Client UX is then notified of the analytics via invoking an event.
- **User leaves the meeting:** The Client Session Manager informs the server session manager about the client's request to leave. The server session manager updates its own session data and allows the client to leave. After getting the response from the server, the session data and user object of the client is set to null.
- **Last user leaves the meeting:** The Client Session Manager receives the information that the meeting has ended. It then notifies the Client UX about the end of the meet via invoking an event.

Design Analysis

The UX depends on the Session Management module which in turn depends on the Networking Module. The Session Manager uses the Content module to get the chats and send users to it. Similarly, the Screen-share and Whiteboard module is also used by the SM to send/set users. To achieve this design in the implementation, the following can be done:

- The Session Manager(SM) will maintain a session object on the server side which will contain the list of all the users present in the meeting.
- The SM will provide the UX with a factory using which the UX can create Session manager objects (both client and the server side) that will live till the end of the meeting.
- The session management module will consist of an interface which will be used by the UX module with the help of functions defined.
- To notify the UX about any changes, the session management module will also consist of listeners for different kinds of subscriptions.
- The Networking module will provide a factory to the SM so that it can create a networking object. The networking module will provide an interface to the Session Management module that will be used by the latter.
- The Session management module will subscribe to the listener(s) provided by the Networking module.
- The Content module will also create an interface which will be used by the SM to set users or get the chat. Same goes for the Screen-share & whiteboard module.
- The Summarizer submodule and the Telemetry submodule will also provide interfaces to the SM to fetch or save the summary and analytics respectively. These objects will be created using the respective factories of these submodules. The SM will also provide the Telemetry with a listener so that any changes in the analytics are notified.

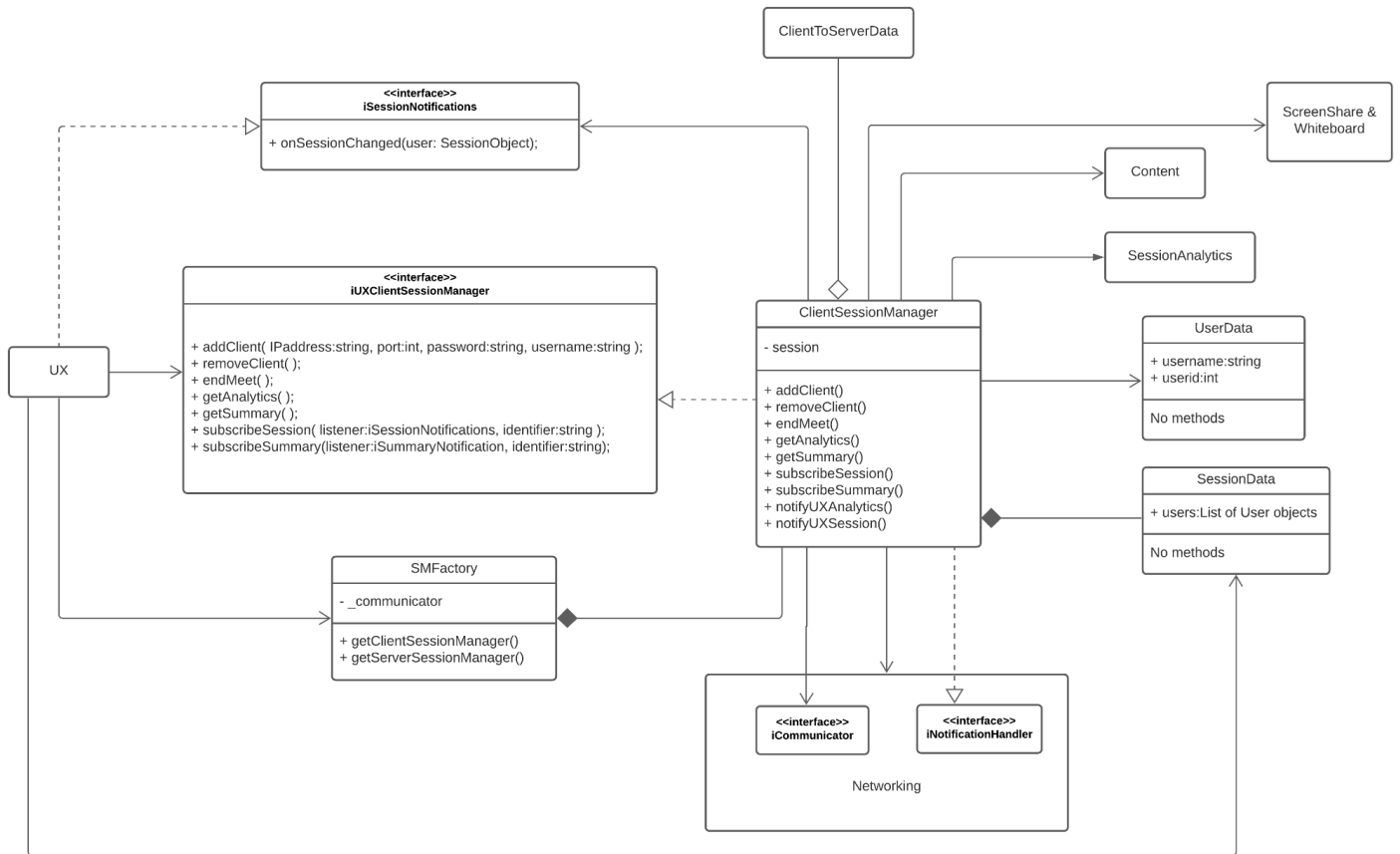
Communication between the Server and Client Sides

To send and receive data from each other, the Server Session Manager depends on the ClientToServerData object to send information about the type of event and the client who requested it. Similarly the server sends the data such as summaryData, sessionData, etc. via the ServerToClientData object.

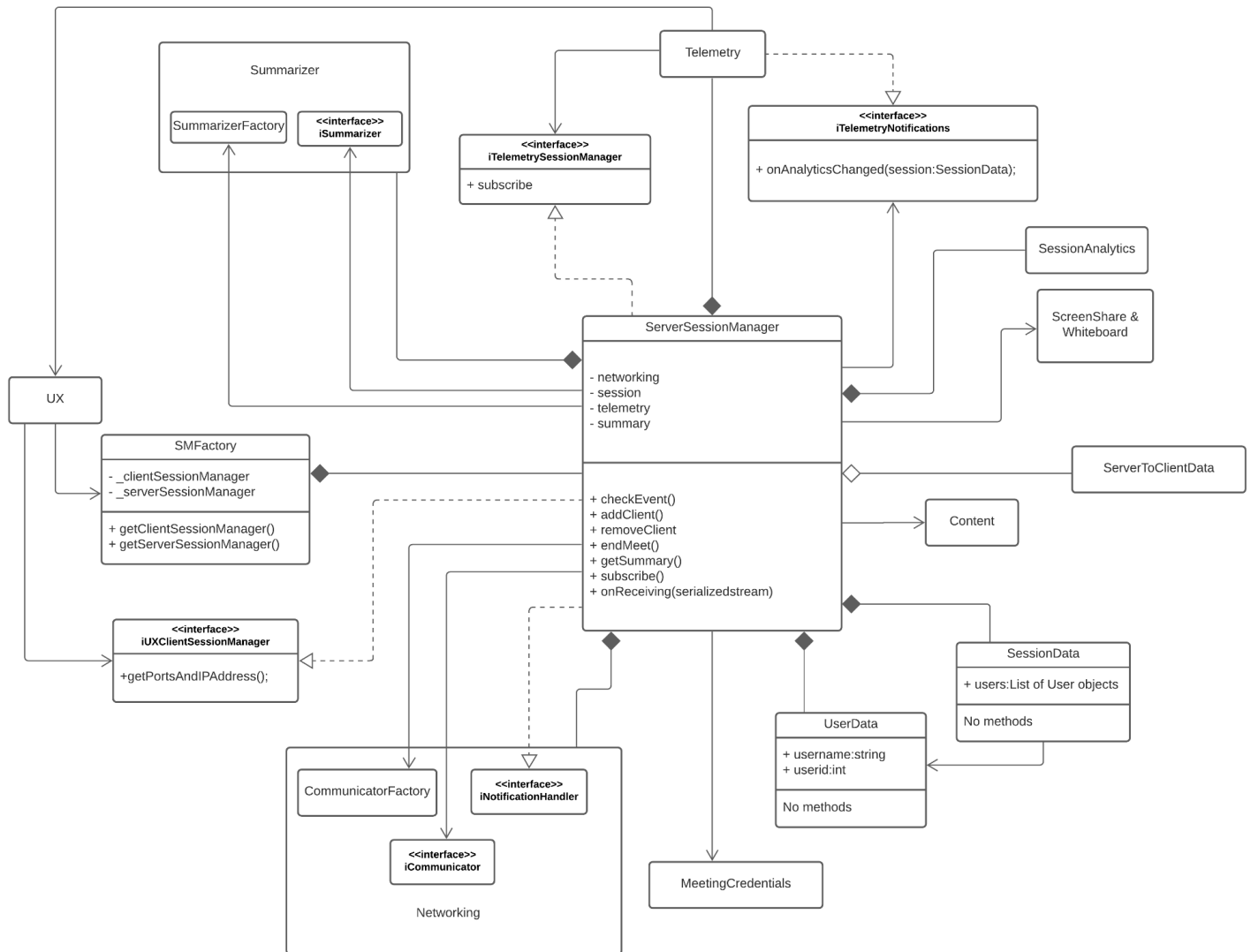
Class Diagram

For simplicity, the functions arguments are not specified in the Client Session Manager as they were already defined in the interface.

Client Side class diagram of the Session Manager:



Server Side class diagram of the Session Manager:



Interfaces

Client Side

```
/// <summary>
/// Interface to notify about changes in the client side
/// session data (SessionData Object).
/// </summary>
public interface IClientSessionNotifications
{
    /// <summary>
    /// Handles the changes in the SessionData object
    /// </summary>
    /// <param name="session"> The changed SessionData </param>
    void OnClientSessionChanged(SessionData session);
}

public interface IUXClientSessionManager
{
    /// <summary>
    /// Adds a user to the meeting.
    /// </summary>
    /// <param name="ipAddress"> IP Address of the meeting. </param>
    /// <param name="ports"> port number. </param>
    /// <param name="username"> Name of the user. </param>
    /// <returns> Boolean denoting the success or failure whether the user was added. </returns>
    bool AddClient(string ipAddress,
        int ports,
        string username);

    /// <summary>
    /// Removes the user from the meeting by deleting their
    /// data from the session.
    /// </summary>
    void RemoveClient();

    /// <summary>
    /// End the meeting for all, creating and storing the summary and analytics.
    /// </summary>
    void EndMeet();

    /// <summary>
    /// Get the summary of the chats that were sent from the start of the
    /// meet till the function was called.
    /// </summary>
    /// <returns> Summary of the chats as a string. </returns>
    void GetSummary();
}
```

```

    /// <summary>
    /// Gather analytics of the users and messages.
    /// </summary>
    void GetAnalytics();

    /// <summary>
    /// Fetches the user data object from the client session manager.
    /// </summary>
    /// <returns>A userData object for that paritcular client.</returns>
    UserData GetUser();

    // Event for notifying summary creation
    public event NotifySummaryCreated SummaryCreated;

    // Event for notifying the end of the meeting on the client side
    public event NotifyEndMeet MeetingEnded;

    // Event for notifying the creation of anlalytics to the client UX.
    public event NotifyAnalyticsCreated AnalyticsCreated;
}

```

Server Side

```

public interface ITelemetryNotifications
{
    /// <summary>
    /// Handles the change in the Global session (SessionData Object)
    /// </summary>
    /// <param name="newSession"> The changed session </param>
    void OnAnalyticsChanged(SessionData newSession);
}

```

```

public interface ITelemetrySessionManager
{
    /// <summary>
    /// Subscribes to changes in the session object
    /// </summary>
    /// <param name="listener"> The subscriber. </param>
    void Subscribe(ITelemetryNotifications listener);
}

```

```

public interface IUXServerSessionManager
{
    /// <summary>
    /// Returns the credentials required to
    /// Join or start the meeting
    /// </summary>
    /// <returns> A MeetingCredentials Object </returns>
    MeetingCredentials GetPortsAndIPAddress();

    /// <summary>
    /// Event to notify the UX Server about the end of the meeting.
    /// </summary>
    public event NotifyEndMeet MeetingEnded;
}

```

Unit Testing

The Session Manager module was tested for its overall functionality. The public functions of both server and client side were tested. To run the tests, the dependent modules were simulated by creating the fake objects for those modules. These fake objects subscribe to the session manager, call the function from the session manager and allow the session manager to use them. These fake objects were given using a parametric constructor that sets these fake objects directly without initialising them.

SUMMARY LOGIC SUBMODULE

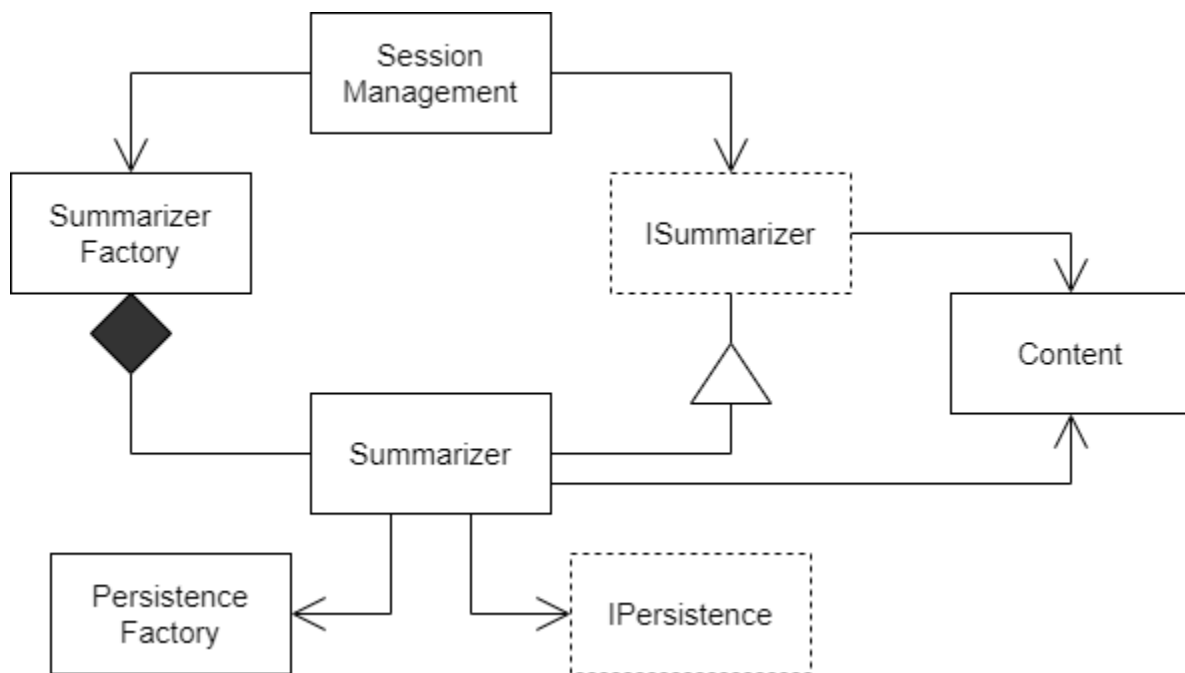
Overview

In our application, we wish to support summarizing the discussion of the session using chat to provide the user with a summary and gist of the discussion that has happened which can help in the revision of selected discussions. This module would be used by the Session management to generate summaries of different discussions.

Objective

- A single interface that can be used by the session management module:
 - This module should be simple and scalable with easy-to-use methods for session management.
 - There should be only one instance of the summarizer that carries out the predefined logic keeping track of the state and the prior while summarizing.
- The module should be running asynchronous in the server to generate a better summary:
 - The module should have methods and instances that run in the background server and not on the fly so that it would not bottleneck the performance of the application.
- The module should be able to generate summary statistics that can be influenced by the user preferences:
 - The summary generated should depend on the chat discussion in the room which would be provided after the end of the discussion and the logic can summarize it using the predefined logic.

Class Diagram



Interface

```
// Interface for Summary Logic Module

public interface ISummarizer {

    // Function to get the summary of the chat
    // and discussion to present in the Dashboard
    string GetSummary(Chat[] chats);

    // Function to save the summary of the entire
    // meeting after the completion via Persistence
    int SaveSummary(Chat[] chats);

}
```

The GetSummary function returns the string which is the summary of the chat without the presence of any metadata and sender information. This is a design choice taken since it is not required to store such information when generating the summary.

Design Analysis

Creation of Summarizer

- The creation of the summarizer is done by using the Summarizer factory which creates an instance of the summarizer which would thus persist throughout the session.
- This summarizer would be used by the Session management since the session manager would be responsible for obtaining the chat messages and other information and redirecting it to various modules.
- The interface of the summarizer is essential along with the summarizer factory since we would want the other modules to access only the different methods that would be required for summarizing and at the same time the so-called summarizer would be obtained by calling the GetSummarizer method of the summarizer factory.
- The abovementioned design choice would provide an abstraction of the internal summarizer class which would not be exposed to other modules as only its interface functions are needed and the other methods would not be required for the functionality.

Summarizer

- The Summarizer class would run the summary logic algorithm which can be data-driven or predetermined based on the requirements after the end of the discussion.
- This particular design would not bottleneck the performance of the entire application waiting for the algorithm to complete execution and it can be done at a later point in time. This also facilitates better summarization of the discussion since the algorithm would be exposed to the entire discussion/ chat while giving the summary a wider picture.

- The first method that would be available to the session manager would be the GetSummary which would be a method that would take a chat string input and provide a concise representation of the chat that would be displayed on the dashboard.
- The Summary logic algorithm would be running a probabilistic algorithm under the hood which wouldn't be requiring any dataset or prior information for the summarization of the chat.
- The generated summary would be used by the session manager to store in the database and retrieved for showing on the dashboard.
- This particular Summarizer runs only on the server side and not on the client side and this is to keep the summary processing only on the server side thereby reducing the need for processing on the client side and the client gets the summary through the UX and Session Manager.

Algorithm

The algorithm that has been designed for the chat summarizer is as follows:

- For each given chat message we find the semantic meaning and associate it with a score based on the count statistics.
- We first find all the tokens using regular expression matching and after breaking it into sentences we also find the tokens of each sentence which would be the building blocks of the sentences.
- We find the lemmas associated with each of the words using the Porter stemming algorithm, more information about which can be found [here](#).
- Upon finding all the lemma tokens we use this to obtain a count based vocabulary which will help determine the score which after being normalized is used by the sampler to generate samples.
- Note that the algorithm that has been designed is a randomized algorithm and the summary samples are obtained based on the probability scores associated with each of the sentences which is found using the normalized scores.

Stretch Code

Multiple directions of work in the future include:

- Incorporation of more information for the summary logic algorithm in addition to the chats for example screen share, whiteboard data, etc.
- Having sophisticated data-driven Machine Learning algorithms that can be used to summarize the collected data while keeping in mind the computational barriers and making the most efficient use of the same.
- Addition of more features including recommendation systems based on the collected data to improve user experience.

TELEMETRY SUBMODULE

Overview

The function of the telemetry module is to provide analytics and statistics related to session and after the sever ends, in an easy to observe format, like histogram or pie chart. This sort of gives the overall activeness of a particular session and the load over whole server.

Objective

To fetch the information from the session management using pub-sub model, i.e. my function will be called if any change occurs. An object of SessionAnalytics is saved with Persistence which contains:

1. A dictionary representing User count at each time stamp.
2. A dictionary representing chat count for each user.
3. Provide the list of users who were there in the session but for very less time(less than threshold time). That means, it checks that all the users who were present in the session for more than a threshold time would only be considered as present in the meeting, else they would be returned in a list.

This object is also forwarded to the UX module so that the statistics can be shown (it is sent via session management).

Also when the session ends, I am providing a general summary of the current session. For that, every time any session ends, I am updating the ServerDataToSave object. The ServerDataToSave object contains the sessionCount and a list of all previous sessions' summary, SessionSummary. SessionSummary contains:

1. userCount : Total number of “sincere” users present in the session.
2. chatCount : Total number of chats in the session.
3. score : It is a score assigned to the session which is nothing but product of userCount and chatCount.

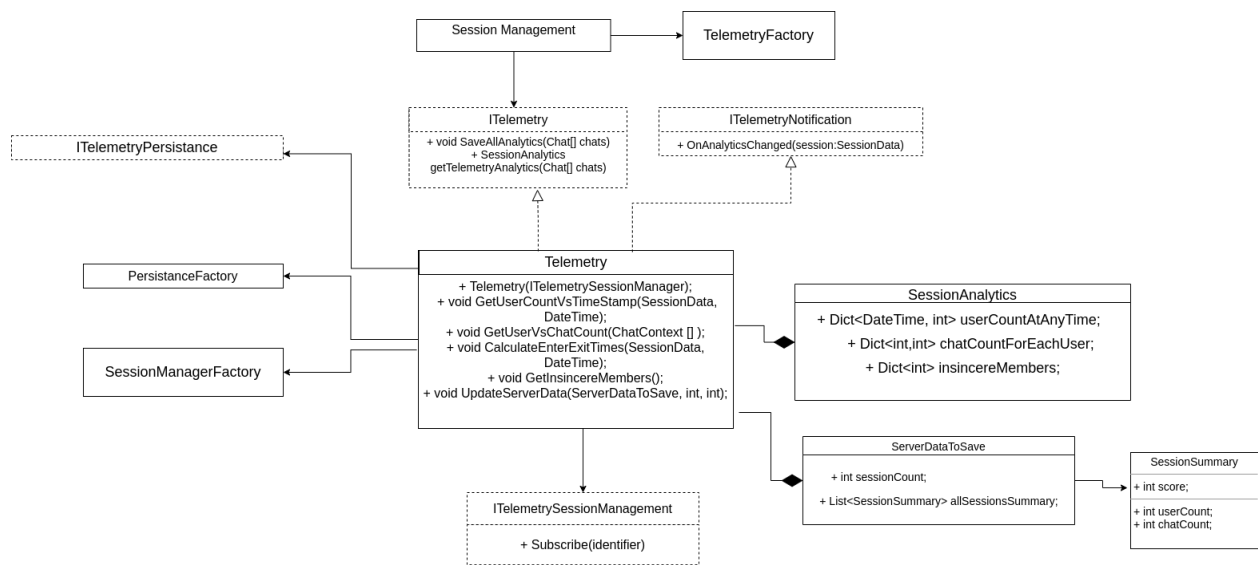
As, if userCount in any particular session is large, then we can say that session was more important, and if chatCount is large then we can say that session is

more interactive, hence score is a function of userCount and chatCount. More score implies more important session.

Design Analysis

- To get the data from session management, I would be subscribing the ITelemetrySessionManager interface, so that whenever any changes occur I would be provided with it. Hence Telemetry would be using the ITelemetrySessionManager interface.
- Telemetry class implements all the analytics functions, and for storing, it will be calling the methods of ITelemetryPersistence, hence, it will be using the ITelemetryPersistence interface.
 1. ITelemetryPersistence method, Save(SessionAnalytics obj) will take object of class SessionAnalytics and saves it.
 2. ITelemetryPersistence method, SaveServerData(ServerDataToSave obj) will take object of class SaverServerDataToSave and stores it.
- Whenever the session ends, the Session Management module will call the ITelemetry's SaveAllAnalytics() method, so that the analytics of the session can be saved with persistence(because only the session management knows when the session ends). Also, session management will call the getTelemetryAnalytics() method to take all the analytics of the session and forward it to UX module, so it will be using the ITelemetry interface.

Class Diagram



Factory

```

public static class TelemetryFactory
{
    static TelemetryFactory()
    {
        if(_telemetry == null)
        {
            _telemetry = new Telemetry();
        }
    }

    public static Telemetry GetTelemetryInstance()
    {
        return _telemetry;
    }

    private static Telemetry _telemetry;
}

```

Interfaces

```

public interface ITelemetry
{
    /// <summary>

```

```

    ///     simplifies all_messages into easily plotable data.
    /// </summary>
    /// <params name="allMessages"> Array of ChatContext object, which
contains info about threads </params>
    public void SaveAnalytics(ChatContext[] allMessages);

    /// <summary>
    ///     Returns the analysed data to be passed to the UI
    /// </summary>
    /// <params name="allMessages"> The chat data of all the threads
</params>
    /// <returns> SessionAnalytics object </returns>
    public SessionAnalytics GetTelemetryAnalytics(ChatContext[]
allMessages);
}

```

Further Plans

1. To store users priority wise according to messages they sent. This would be useful to find the active students in the sessions and inactive students.
2. To come up with more innovative ideas(like insincereMembers) of analysis which will add more value to the project, and hence can be more useful.

PERSISTENCE SUBMODULE

Overview

Persistence module is one of the important modules in the dashboard, main function of this module includes saving the Analytics, Histograms etc in the png format in each session and also summary of the chats to the local file system as.txt, when the meeting session ends. It is directly connected to other two modules namely Summary and Telemetry. These modules are supposed to call save and retrieve functions in order to persist the summaryChats and Analytics images.

Objective

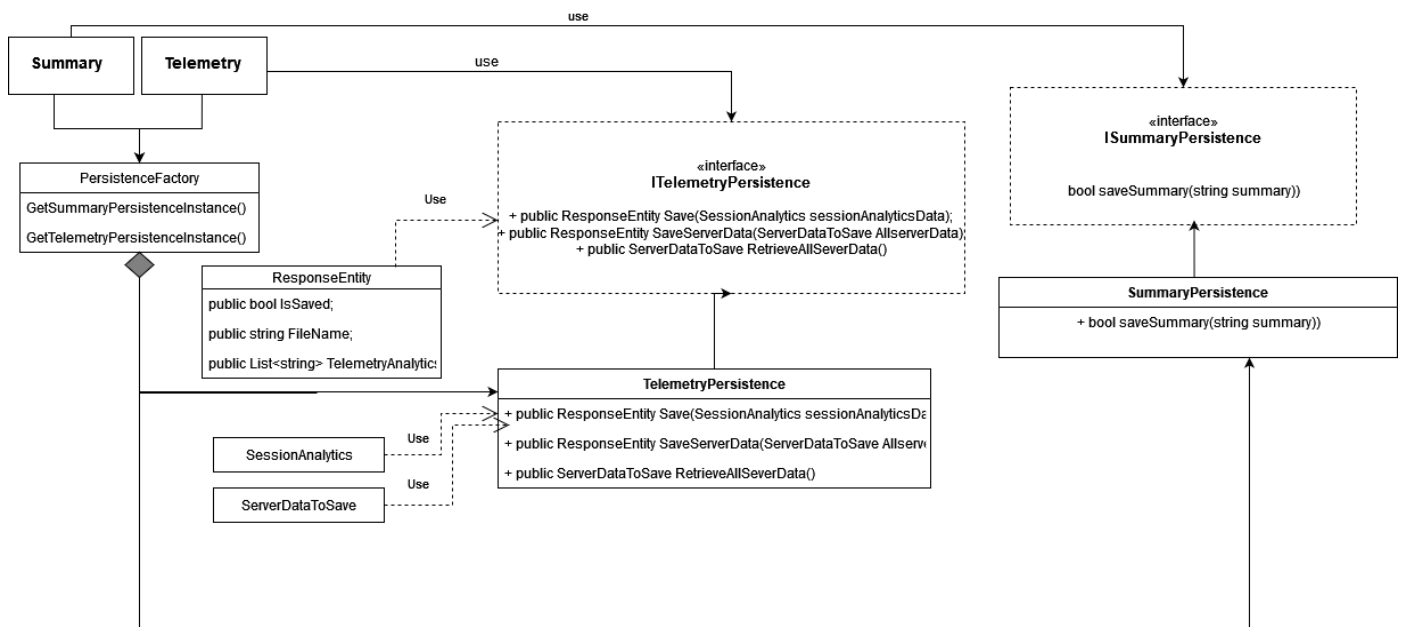
- Stores the Analytics images in the .png format session wise.
- Stores the Server Data in a meaningful way in a global XML document.
- Retrieves the above Server Data and sends it back to the Telemetry module at the end of the meeting to modify and store again.
- Stores the ChatSummary in the .txt format at the end of the meeting.

Design Analysis

- This module should expose IPersistence interfaces to the other two modules and it is good to have a Factory model to create an abstraction layer while creating objects.
- We need a PersistenceFactory, which should have GetTelemetry and GetSummary Persistence instance creator functions, which would be exposed to other modules as well.
- The Telemetry Persistence needs input in the form of Dictionaries to store UserCountVsTimeStamp, UserIdVsChatCount and InsincereMember as lists which create a histogram plot as png for former and a text file in order to store them.
- It must also get the data in the form of string(More preferably) from Summary Module and call Save function to save the Summary in a text file named in Current DateTime.txt format.

- There should be different save functions to store different files like in case of storing the summaryChat, we need different inputs from the other case in which it is responsible to store Histogram.
- To serve this purpose, it is better to create two different Interfaces like ISummaryPersistence and ITelemetryPersistence.
- Summary and Telemetry module should be using their corresponding interfaces.
- In case of telemetry, It is also required to store the ServerData, which needs to be kept updated as the meeting ends.
- It is easy to implement and maintain XML files for storing the ServerData as we can easily serialize and deserialize them to respective objects easily.
- For Insincere members, we can easily store them in a text file.
- Other Session Analytics data, only need to be saved in the form of png as they need not be retrieved but need to be persisted.

Class Diagram



Interfaces

```
namespace Dashboard.Server.Persistence
{
    ///SummaryPersistence Interface
    2 references
    public interface ISummaryPersistence
    {
        /// <summary>
        /// saves the summary of the session into a summary file
        /// </summary>
        /// <param name="message"> takes message string that need to be saved </param>
        /// <returns> return true if succesfully saved else return false </returns>
        2 references
        public bool SaveSummary(string message);
    }
}
```

```

using Dashboard.Server.Telemetry;

//Persistence
namespace Dashboard.Server.Persistence
{
    2 references
    public interface ITelemetryPersistence
    {
        /// <summary>
        /// save the UserCountVsTimeStamp, UserIdVsChatCount, InsincereMember data as png after each session.
        /// </summary>
        /// <param name="sessionAnalyticsData"> takes sessionAnalyticsData from Telemetry. </param>
        3 references | 1/1 passing
        public ResponseEntity Save(SessionAnalytics sessionAnalyticsData);

        /// <summary>
        /// append the ServerData into a file after each session end
        /// </summary>
        /// <param name="AllserverData"> takes ServerData from Telemetry to be saved into text file </param>
        /// <returns>Returns true if saved successfully else returns false</returns>
        5 references | 3/3 passing
        public ResponseEntity SaveServerData(ServerDataToSave AllserverData);

        /// <summary>
        /// retrives the ServerData after end of all of the sessions.
        /// </summary>
        /// <returns>returns List of SeverData</returns>
        5 references | 3/3 passing
        public ServerDataToSave RetrieveAllSeverData();
    }
}

```

```

/// <summary>
///     create instances of Summary or Telemetry Persistence Class respectively
/// </summary>
public static class PersistenceFactory
{
    ///<summary>
    ///     It is constructor for the Persistence factory and enables to create only
    ///     one instance in a single run.
    /// </summary>
    static PersistenceFactory()
    {
        if (_summaryPersistence == null)
        {
            _summaryPersistence = new SummaryPersistence();
        }

        if (_telemetryPersistence == null)
        {
            _telemetryPersistence = new TelemetryPersistence();
        }
    }

    /// <summary>Helps to create instance of ISummaryPersistence </summary>
    /// <returns> return ITelemetryPersistence Interface which enables to use functions of class inheriting the same </returns>
    public static SummaryPersistence GetSummaryPersistenceInstance()
    {
        return _summaryPersistence;
    }

    /// <summary>Helps to create instance of ITelemetryPersistence </summary>
    /// <returns> return ITelemetryPersistence Interface which enables to use functions of class inheriting the same </returns>
    public static TelemetryPersistence GetTelemetryPersistenceInstance()
    {
        return _telemetryPersistence;
    }

    private static SummaryPersistence _summaryPersistence;
    private static TelemetryPersistence _telemetryPersistence;
    public static ResponseEntity lastSaveResponse;
}

```

Persistence Module: Unit Testing

Test S.No	Test	Logic for Unit Testing	Status
1.	Tests the SaveSummary function of SummaryPersistence class, if it is able to correctly save the Summary as .txt file.	By Calling the Save function with random String as argument and checking if the saved file is present at the location and contains the same Text as expected.	Passed
2.	It tests the SaveSummary function of SummaryPersistence It must fail on null PathException, if we try to set the path as null	Creating summary_Persister instance, Setting the path to null and then calling the SaveSummary() with string as argument.	Passed

3.	It tests the SaveServerData function of TelemetryPersistence	Creating a sample ServerDataToSave instance and Saving the same by calling the function and then check if file is present at the location.	Passed
4.	It tests the RetriveAllSeverData function of TelemetryPersistence	Saving sample ServerDataToSave instance by calling SaveServerData() and check if file exists at the location. Also deserialized that xml document and creating the ServerDataToSave Object and then check if it is equal to the expected by calling isSave() util function	Passed
5.	It tests the RetrieveAllSeverData_FailsOnEmptyPathExceptions function of TelemetryPersistence It must fail to retrieve if path is set to empty as No xml file will be found there	Setting the telemetry_persister ServerDataPath to ""(empty string) and then catching the EmptyPathException	Passed
6.	It tests the SaveSummary function of Summary Persistence. It must fail on ArgumentException of Empty Path	Setting the summary_persister summary_path to ""(empty string) and then catching the ArgumentException	Passed
7.	It tests the Save function of TelemetryPersistence	Creating Sample Instance of All types Analytics data to create sample sessionAnalytics data and then calling Save() function, also checking if all three files are present there.	Passed

Further Plans

If time permits and feasible to do it, it will also store screen share data as well.

FUTURE SCOPE

- Add screen share images and whiteboard objects to the summary of the meeting.
- Add more telemetry analytics such as latency in the network, etc in the final meeting analysis