CS5617 Software Engineering

Content Module Specification

- By Sahil J. Chaudhari (111801054)
Team leader Content Module

Table of Content:

Team:

1. Sahil J. Chaudhari, 111801054, Content module team leader
2. Sameer Dhiman, 111801038, Team member
3. Vishesh Munjal, 111801055, Team member
4. Yuvraj Raghuvanshi, 111801048, Team member

Overview:

Content module is an important part of the project that takes care of sending and receiving chats and files from one client to another. This module will be used by the User Experience(UX) module to handle chat and file traffic with various features such as starring of chat, emojis, sharing links, message threading etc., and by the Dashboard module to create a summary. Content module will use the network module's class to send and receive chats and files in the form of serialized XML string where serializer and deserializer is provided by the network module from one client to another client.

The client of the content module will rest in the client side of software to send and receive messages and files but in order to store and fetch files in the system and to store chat tables, there will be a class of content module in server side as well. Server side content manager is also responsible for generating new message IDs, thread IDs, sensing sender and receiver and sending messages to respective receivers.

At the start of the session, As soon as the first user enters, the UX module will create a first client side content manager which will create a server side content manager to initialize chat and file tables.

In case of new user login into session, Session manager will call setUser method provided by client content manager which will assigns unique ID to content manager that was created by UX module and then Session manager will call SendAllMessagerToUser with user ID that is provided by server side content manager and server will sense it and will create list of thread datatype to send all data stored in chat and file tables in database of server to desired client side content manager of ID of userID and then client side content manager will inform UX team about data and they will show it to User.

Objectives:

Phase 1:

- ➔ **Sending and receiving of plain text messages in broadcast manner**
- ➔ **Starring of messages, message context (Reply in threads)**
- ➔ **Sending and receiving of files**

Phase 2:

- ➔ **Chat features such as emoji, chat threading, updating messages**

- ➔ **Link sharing, Highlighting messages, private messaging**
- ➔ **File preview**

Note: Phase 2 objectives are highly dependent on other modules availability as they have to modify their classes and designs to support it. E.g for private texting we require a one to one interface from the networking team, to provide emojis, chat threading and other feature UX modules need to be confident.

Roles:

i) Content Client side Chat (Vishesh Munjal)

There will be a client side manager for each user, UX will call sendchat, mark star and download, getThreadID, etc functions that are available to other modules. Then the content manager will redirect requests to the chat client which will create a message object with a particular event attached to it such as mark star, update along with message body, sender info and type as chat and then serialize it in XML string and will send it to server side manager via network module.

On the other hand, after receiving a message from a network where the content manager has subscribed, the chat client will process the object and will inform the subscribers about the incoming message so they can fetch it. This chat client will be built by Vishesh, calling network side subscribers and listening also will be handled by him.

- ➔ **Receiving messages from UX, converting them into sending message data blocks and serializing them using a network client.**
- ➔ **Handling of starred event, message updates, Hyperlinks**
- ➔ **Subscribe to network module, Deserialize messages received into receive message data blocks and inform subscriber about message**

ii) Content Client side File (Yuvraj Raghuvanshi):

As there will be a content client manager which will be providing interface to other modules along with client factory to establish interfaces will be taken care of by yuvraj. If a module sends a file as a message then the file client will process it and serialize it into xml string with send information and event information such as starred and send it to server side via network.

- ➔ **Implementation of ContentClientFactory and ContentClient to communicate with other modules**
- ➔ **Implementation of file client to handle files, handles download functionality.**
- ➔ **Subscribe to the network module, deserializes messages, stores files in memory on client side and informs subscribers.**

iii) Content Server Management (Sameer Dhiman):

As all of the message objects are redirected to the server to process them and update chat and file tables. Server manager of content will subscribe to the network to listen to message objects sent by client side and deserialize it. If the message is of file type then the

server will store it into the database and if the client side sends a download request then sends it to a particular user. If the message is of chat type then if it is a new message then manager will insert it into the table otherwise it will update the table in the database. Once finished, it will create a message object and broadcast it or in case of private message send it to a particular client side manager with ID via the network module after serializing the object into XML string.

Server also handler sendAllData request made by session manager that gives ID of user then server will create packets of list of thread datatype of chat and file tables and send it that ID on client side using network. Also handles getAllMessages requests which send data to the dashboard module.

- ➔ **Insertion and updating of database**
- ➔ **Handling Subscribe, GetAllMessage, SendAllData request by other modules**
- ➔ **Handling downloading requests on server side**
- ➔ **Sending messages in broadcast and private manner according to requests.**

Design Pattern:

A. Singleton:

- ➔ Since there will be only one instance of client side content manager with unique ID and server side content manager, we have to restrict more than one instance of them, hence singleton design pattern is chosen.
- ➔ As our client side and server side factories are the one who are responsible for creating instances and providing interfaces, this will depend on singleton design patterns.
- ➔ As soon as our first client side interface is established via factory, server side content manager will be also initialized, due to the singleton pattern, it will be insured by that server.
- ➔ Advantages of Singleton pattern are:
  - ◆ Singleton pattern can implement interfaces
  - ◆ It helps to hide dependencies
  - ◆ It provide single point access to a particular instance, easy to maintain

B. Publisher-Subscriber:

- ➔ In a publisher-subscriber design pattern, the publisher publishes messages and there are one or more subscribers who listen to the published messages.
- ➔ It is another version of the observer design pattern.
- ➔ In our design, client that sending messages publishes message over network to server side of content and servers subscribes to network module to listen messages, it process messages and then broadcast or sends to private using network, on receiving side of client, it will subscribe to network to listen messages that are sent by server.

- ➔ Dashboard module will subscribe to server side of content and UX module will subscribe to client side of content. Hence there is publisher-subscriber design between UX, dashboard, content and network.
- ➔ Advantages of publisher subscriber pattern is that publisher and subscriber doesn't need to know each other whereas in observer design pattern observer must know its observable, plus if wrongly implemented, observer pattern can add complexity and affect performance while publisher subscriber pattern is less error prone and easy to maintain.

Activity Diagram:

Fig 1: Communication overview

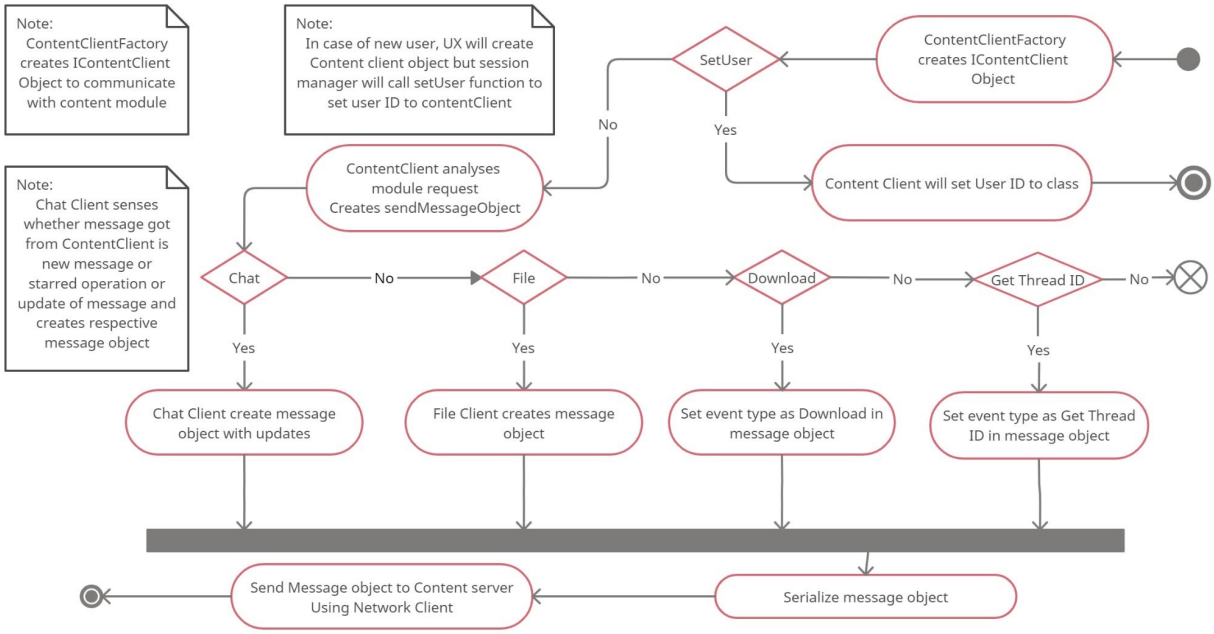


Fig 2: Content Client Sender side

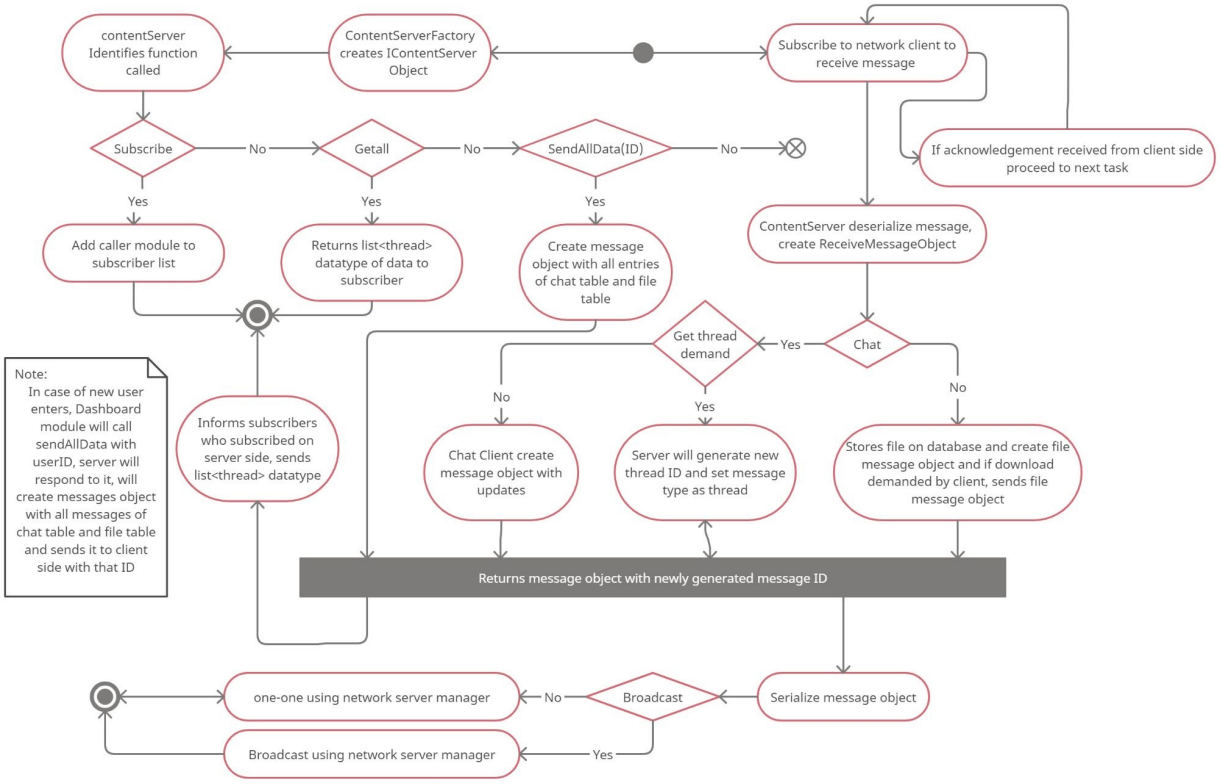


Fig 3: Content Server

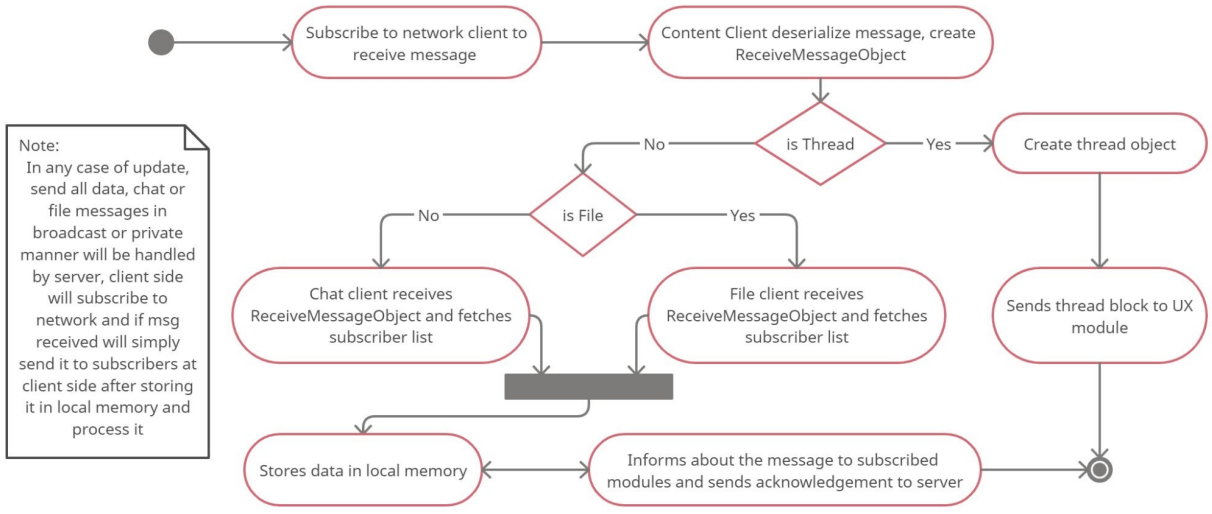


Fig 4: Content Client Receiver side

Data structures:

The following are the data structures that will be used for inter communication between content and other modules which will be used along with the functions called using interfaces provided on the next page.

```
enum MessageType
{
File,Chat
}

enum MessageEvent
{
    Update,
    NewMessage,
    Star,
    GetAll,
    Download
}

class SendMessageData
// to send a message, construct a SendMessageData object
{
    MessageType message, // enum/string indicating file or chat
    string message, // message if chat / filepath if message type is file
    int receiverIds[], // list of receiver ids
    int replyThreadId, // special value (-1) if message not part of any existing thread
                       // otherwise thread id of thread to which the message belongs
};

class ReceiveMessageData
// received messages will be ReceiveMessageData objects
{
    MessageEvent event, // will be one of update, new message or star
    MessageType message, // enum/string indicating file or chat
    string message, // message if message type is chat / filename and size if file
    int messageId,
    int senderId,
    int receiverIds[], // list of receiver ids
    int replyThreadId, // special value (-1) if message not part of any existing thread
                       // otherwise thread id of thread to which the message belongs
    DateTime sentTime, // time at which message was sent

    boolean starred // value indicating whether a message is to be included in the summary
};

class Thread
{
List<ReceiveMessageData> msgList,
int threadId,
int numOfMessages,
DateTime creationTime
}
```

Interfaces:

Client Side (Overview of the IContentClient interface):

```
public class ContentClientFactory
{
	public static setUser(int userId),
	static getInstance() // returns an IContentClient implementation
}
```

setUser(userId): This function takes userID as input and sets it to IContentClient's ID. Will be used by the session manager from the dashboard module.

These are the functions provided by the client-side content module interface, IContentClient:

- CSend(SendMessageData)
- CDownload(int messageId, string savePath)
- CMarkStar(int messageId)
- CUpdateChat(int messageId, string newmessage)
- CSubscribe(IContentListener listener)
- Thread CGetThread(int threadId)

A. **CSend(SendMessageObject):** This function takes in a ***SendMessageData*** type object and sends the chat/file message depending on the data members of the object to the other clients.

B. **CDownload(int messageId, string savePath):** When the user clicks download on a file type message (which will essentially display just the filename and size, because the file isn't sent to a client until it is downloaded), the UX can call this function with the message id and the path to save the file in. The Content module will verify that message id is in fact for a file type message, fetch the file from the server and save it to the given file path.

C. **CMarkStar(MessageId):** This function is used to mark a message as starred, i.e., to prioritise it to be included in the summary generated by the dashboard module.

D. **CUpdateChat(MessageId, string):**This function allows editing an already sent *chat* message.

E. **CSubscribe(IContentListener):** This is the subscribe function based on the publisher subscriber design pattern. Any modules that want to listen to messages received on the client (UX, in this case) will provide an object that implements the ***IContentListener*** interface, which is defined as follows

IContentServerListener
{

```
        void OnMessage(ReceiveMessageData)
        void OnAllMessages(List<Thread>)
    }
```

F. **Thread CGetThread(int threadId):** Takes in the thread id and returns a `Thread` object whose id is the given thread id.

Server Side (Overview of the IContentServer interface):

```
public class ContentServerFactory
{
        static getInstance() // returns an IContentServer implementation
}
```

These are the functions provided by the server-side content module interface, IContentServer:

- SSubscribe(IContentListener listener)
- List<Thread> SGetAllMessages()
- List<Thread> SSendAllMessagesToClient(userId)

A. **SSubscribe(IContentListener listener):** This is the subscribe function based on the publisher subscriber design pattern. Any modules that want to listen to messages received on the server (Dashboard, in this case) will provide an object that implements the ***IContentListener*** interface, which is defined as follows

```
    IContentClientListener
    {
        void OnMessage(ReceiveMessageData)
    }
```

B. **List<Thread> SGetAllMessages():** This function can be used to receive all messages that have been sent until now. It returns an object of data type **list<Thread>**.

C. **List<Thread> SSendAllMessagesToClient(userId):** This function takes userID as input, and is used to send all messages that have been sent until now in broadcasted manner to provide userID's client side content manager which will send it to UX. It returns an object of data type **list<Thread>**.

Specification of Vishesh Munjal:

Topics Covered:
Client Side Content Module

- **The NetworkClientCommunicator:** This mainly deals with the Network's module subscriber and also tells a bit about the need of the class.
- **Chat Client:** This module deals with the implementation of the chat messages and explains the need and workflow along with the information about the methods in the class.

The NetworkClientCommunicator

This will implement the Network's module subscriber. It is intuitive from the name that this function would be subscribed to the networks module and in case of a receive of a message will act.
As per the design, the idea is that as an acknowledgement/working for a message once it is passed to the server it is resend to the client by rebroadcasting of the message. Cases like such creates a need for a receive message from the Network module in the client side as well.

ChatClient

This class will handle the chat message methods in particular. The workflow for a chat message is as follows: The message is given to the content module by UX through the means of subscribing(subscribing is handled in ContentClient) from there end. Then the message object for that message distinguishes between the appropriate message type i.e., if it's a file or chat and then the appropriate methods of ContentClient calls the ChatClient methods in case of chat type message. Understanding the workflow we get an idea for the need of the ChatClient.

ChatClient Methods:-

- Send(): This function first creates a respective message object specifically for chat to be sent to the server via networking to the server and sends it to the server.
- Receive(): This function will receive the ReceiveMessageObject for the purpose of sending it to the respective receivers i.e. to inform all the members of ContentClient::Subscribers (maybe UX or dashboard etc.). A message object is received usually from the server via a networking module which contains the event that is to be triggered for that particular object for example :- in case of a new message the event namely is a new message and if sent to UX with the same user ID will indicate an acknowledgement with message ID.
  This also updates the ContentClient::threadTable i.e. the hashmap structure to get fast access to thread index numbers and also updates the ContentClient::allMessage i.e. the list of threads. To clarify, here all messages are stored as threads only i.e. even if a single message which is not a part of a thread communication, it will be a thread of its

own. Now on receiving an object may have some event associated with it that may require making changes to these data members of ContentClient.

Note: The creation of a message object here as well depends on the factor that the event of a message may be marked here in this class.

A depth look at the message objects/ data structure:

Note: This contains the data objects for the whole module and hence is not specific for only one speculation.

```
enum MessageType
{
File,
Chat
}

enum MessageEvent
{
    Update,
    NewMessage,
    Star,
    Download
}

class SendMessageData
// to send a message, construct a SendMessageData object
{
    MessageType message, // enum or string indicating file or chat
    string message, // message if chat, filepath if message type is file
    int receiverIds[], // list of receiver ids
    int replyThreadId, // special value (-1) if message not part of any existing thread
                       // otherwise thread id of thread to which the message belongs
};
```

```
class ReceiveMessageData
// received messages will be ReceiveMessageData objects

{
    MessageEvent event,  // will be one of update, new message or star
    MessageType message, // enum/string indicating file or chat
    string message, // message if message type is chat / filename and size if file
    int messageId,
    int senderId,
    int receiverIds[], // list of receiver ids
    int replyThreadId, // special value (-1) if message not part of any existing thread
                       // otherwise thread id of thread to which the message belongs
    DateTime sentTime, // time at which message was sent
```

```
    boolean starred // value indicating whether a message is to be included in the summary
};

class Thread
{
List<ReceiveMessageData> msgList,
int threadId,
int numOfMessages,
DateTime creationTime
}
```

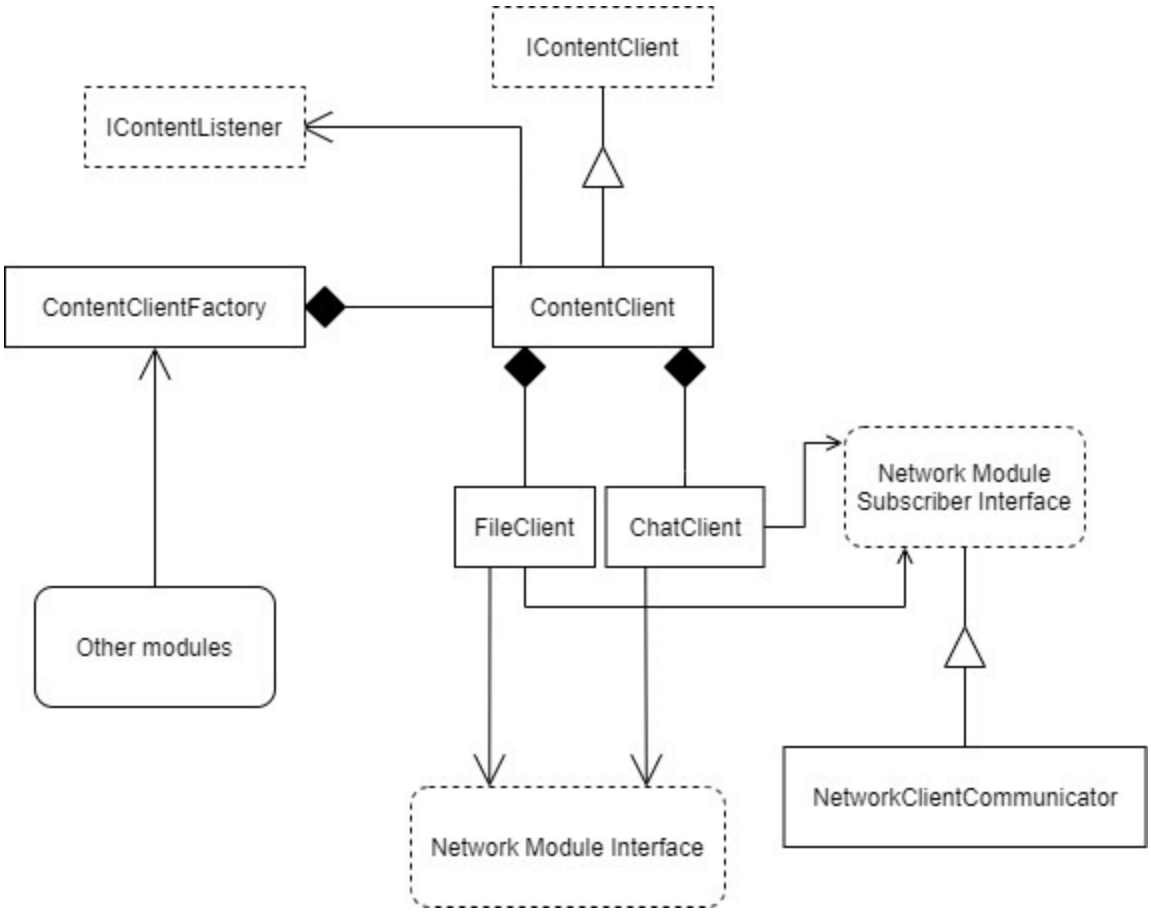
Message data structure to be used as an object universal tool for all internal working.

```
// used for communication between the content module on client and server
MessageData
{
    message event // update, new message, star, download file
    message type, // file or chat
    message string, // filename if file / chat message if chat
    message id, // -1 if sending to server, id if downloading or receiving from server
    file data, // empty if chat
    sender id,
    receiver ids,
    reply thread id,
    starred, // false default
};
```

Class Diagram:

Client Side Content Module

Note: The class diagram contains all classes for the client side of the content module that is being handled by Yuvraj and me(Vishesh).



Specification of Yuvraj Raghuvanshi:

What's covered in this sub-spec:

Client-side content module:

- Message objects for sending and receiving using the ***IContentClient*** interface
- Implementation of the ***IContentClient*** interface: The ***ContentClient*** class
- File sharing: The ***FileClient*** class
- Singleton Factory that returns an IContentClient implementation: ***ContentClientFactory***.

Message data structures for sending and receiving messages

```
enum MessageType
{
File,
Chat
}

enum MessageEvent
{
    Update,
    NewMessage,
    Star,
    Download
}

class SendMessageData
// to send a message, construct a SendMessageData object
{
    MessageType message, // enum or string indicating file or chat
    string message, // message if chat, filepath if message type is file
    int receiverIds[], // list of receiver ids
    int replyThreadId, // special value (-1) if message not part of any existing thread
                       // otherwise thread id of thread to which the message belongs
};
```

```
class ReceiveMessageData
// received messages will be ReceiveMessageData objects

{
    MessageEvent event,  // will be one of update, new message or star
    MessageType message, // enum/string indicating file or chat
    string message, // message if message type is chat / filename and size if file
    int messageId,
    int senderId,
    int receiverIds[], // list of receiver ids
```

```
    int replyThreadId, // special value (-1) if message not part of any existing thread
                       // otherwise thread id of thread to which the message belongs
    DateTime sentTime, // time at which message was sent

    boolean starred // value indicating whether a message is to be included in the summary
};

class Thread
{
List<ReceiveMessageData> msgList,
int threadId,
int numOfMessages,
DateTime creationTime
}
```

Message data structure to be used internally by the content module

```
// used for communication between the content module on client and server
MessageData
{
    message event // update, new message, star, download file
    message type, // file or chat
    message string, // filename if file / chat message if chat
    message id, // -1 if sending to server, id if downloading or receiving from server
    file data, // empty if chat
    sender id,
    receiver ids,
    reply thread id,
    starred, // false default
};
```

Overview of the IContentClient interface

These are the functions provided by the client-side content module interface, IContentClient:

- CSend(SendMessageData): This function takes in a ***SendMessageData*** type object and sends the chat/file message depending on the data members of the object to the other clients.
- CMarkStar(MessageId): This function is used to mark a message as starred, i.e., to prioritise it to be included in the summary generated by the dashboard module
- CUpdateChat(MessageId, string): This function allows editing an already sent chat message.
- CSubscribe(IContentListener): This is the subscribe function based on the publisher subscriber design pattern. Any modules that want to listen to messages received on the client (UX, in this case) will provide an object that implements the ***IContentListener*** interface, which is defined as follows:
    - IContentListener
      {
      void OnMessage(ReceiveMessageData)
      void OnAllMessages(List<Thread>)
      }
- Thread CGetThread(int threadId): Takes in a thread id and returns a `Thread` object whose id is the given thread id.

The *ContentClient* class

The interaction with the content module running on the client application is defined via the ***IContentClient*** interface. The ***ContentClient*** class implements this interface. It is *composed* of two helper classes- ***FileClient*** and ***ChatClient*** to fulfil the contract implied by the interface. Details of the ***ContentClient*** class:

Important Data Members:

- int userId: The user ID of the client running.

- IContentClient Subscribers []: A list of all objects that have subscribed to the Content module according to the publisher-subscriber design pattern.
- static Queue<SendMessageData> sendQueue: Queue for messages that need to be sent

- static List<Thread> allMessages: List that stores all messages received and sent on a client

Methods:

- CSend(SendMessageData): This function takes in a ***SendMessageData*** object, and depending on the message type, creates a ***MessageData*** object with the message event pertaining to a new message, and uses the ***ChatClient*** or the ***FileClient*** class to send the created ***MessageData*** object

- CMarkStar(MessageId): This function looks up the message with the given message id, creates a ***MessageData*** object, sets the message event appropriately, and again, uses the ***ChatClient*** or the ***FileClient*** class to relay the information to the server to mark the message as starred.

- CUpdateChat(MessageId, string): Similar to MarkStar, creates a ***MessageData*** object, sets the event to indicate update, and uses the ***ChatClient*** class (as we plan on only allowing chat messages to be edited.

- CSubscribe(IContentListener): The IContentListener object passed as an argument is appended to the Subscribers[] list data member, so the object is subscribed to all received messages.

- Thread CGetThread(threadId): Simply indexes into the list of threads (we may keep a hash table indexed by thread ID for fast access) and returns the required Thread object.

- Additional helper methods as needed that will be used internally by the content module, and will not be visible to any other module.

The FileClient class

The ***FileClient*** class contains methods to handle file sharing on the client by making use of the network module.
At the very least, it will contain functions for the following tasks:

- Sending files to the server from where it can be sent to the other clients as needed.
- Handling file notification messages from the server. What is meant by file notifications is the following- when a client sends a file, it isn't sent directly to all recipients. Instead, the recipients get notified that a file has been sent by the sender, and they can download it if they wish to. Visually, this notification is nothing but a message containing filename and size, along with a download button, that the UX will add to such messages.
- Downloading a file from the server. Once a user hits download on a file notification message, this function will be called by ***ContentClient*** to download the file from the server and store it in the path given to it.
- Several helper functions for the above tasks

The ContentClientFactory class

Following the factory design pattern, we provide a singleton factory- the ***ContentClientFactory*** class, which composes a class that implements the ***IContentClient*** interface.
This factory will provide two static methods:

- static void setUser(int userId): This needs to be called before obtaining an object to set the userId of the client. This function needs to be only called once to initialize the state of the static ***ContentClient*** class object.

- static IContentClient getInstance(): This function will return an object implementing the ***IContentClient*** interface. Since the factory follows the singleton design pattern. Only one object will be created and the same object will be returned each time this function is called.

Specification of Sameer Dhiman:

What's covered in this sub-spec:

Server-side content module:

- Implementation of the ***IContentServer*** interface: The ***ContentServer*** class
- File sharing: The ***FileServer*** class
- Chat: The ***ChatServer*** class
- Singleton Factory that returns an IContentServer implementation: ***ContentServerFactory***.

Message data structures for sending and receiving messages

```
enum MessageType
{
File,
Chat
}

enum MessageEvent
{
    Update,
    NewMessage,
    Star,
    Download
}

class ReceiveMessageData
// received messages (by the client) will be ReceiveMessageData objects

{
    MessageEvent event,  // will be one of update, new message or star
    MessageType message, // enum/string indicating file or chat
    string message, // message if message type is chat / filename and
size if file
    int messageId,
    int senderId,
    int receiverIds[], // list of receiver ids
    int replyThreadId, // special value (-1) if message not part of any
existing thread otherwise thread id of thread to which the message
belongs
    DateTime sentTime, // time at which message was sent
```

```
    boolean starred // value indicating whether a message is to be
included in the summary
};

class Thread
{
List<ReceiveMessageData> msgList,
int threadId,
int numOfMessages,
DateTime creationTime
}
```

Message data structure to be used internally by the content module

```
// used for communication between the content module on Server and
Client
MessageData
{
    message event // update, new message, star, download file
    message type, // file or chat
    message string, // filename if file / chat message if chat
    message id, // -1 if sending to server, id if downloading or
receiving from server
    file data, // empty if chat
    sender id,
    receiver ids,
    reply thread id,
    starred, // false default
};
```

Overview of the IContentServer interface

These are the functions provided by the Server-side content module interface, IContentServer:

- CSubscribe(IContentListener): This is the subscribe function based on the publisher subscriber design pattern. Any modules that want to listen to messages received on the Server will provide an object that implements the ***IContentListener*** interface, which is defined as follows:

- IContentListener
  {
  void OnMessage(ReceiveMessageData)
  void OnAllMessages(List<Thread>)
  }

- List<Thread> SGetAllMessages(): returns list of all threads sorted by creation time
- SSendAllMessagesToClient(userId): Will send all the messages till now to the new user that connects to the server.

The *ContentServer* class

The interaction with the content module running on the Server application is defined via the ***IContentServer*** interface. The ***ContentServer*** class implements this interface. It is *composed* of two helper classes- ***FileServer*** and ***ChatServer*** to fulfil the contract implied by the interface. Details of the ***ContentServer*** class:

Important Data Members:

- IContentServer Subscribers []: A list of all objects that have subscribed to the Content module according to the publisher-subscriber design pattern.
- static Queue<SendMessageData> sendQueue: Queue for messages that need to be sent
- static List<Thread> allMessages: List that stores all messages received and sent on a Server

Methods:

- CSubscribe(IContentListener): The IContentListener object passed as an argument is appended to the Subscribers[] list data member, so the object is subscribed to all received messages.
- List<Thread> SGetAllMessages(): returns list of all threads sorted by creation time
- SSendAllMessagesToClient(userId): Will send all the messages till now to the new user that connects to the server.
- Additional helper methods as needed that will be used internally by the content module, and will not be visible to any other module.

The FileServer class

The ***FileServer*** class contains methods to handle file sharing on the Server by making use of the network module.
At the very least, it will contain functions for the following tasks:

- Taking and storing files sent by a client on the server.
- Creating the message object from file info to send to other clients.

- Handling download requests from all clients.

The ChatServer class

The ***ChatServer*** class contains methods to handle chats on the server.
At the very least, it will contain functions for the following tasks:

- Receiving messages from clients and sending them to their respective receivers.
- Persisting the chats using ***ContentDatabase*** class.
- Handling updates to messages.

The ContentDatabase Class

This class handles all the operations relating to the database on the server. It will be our internal class that will be used to:

- Save a new message on the database.
- Update fields of previous message.
- Get all the messages.
- Look for a message using its id.

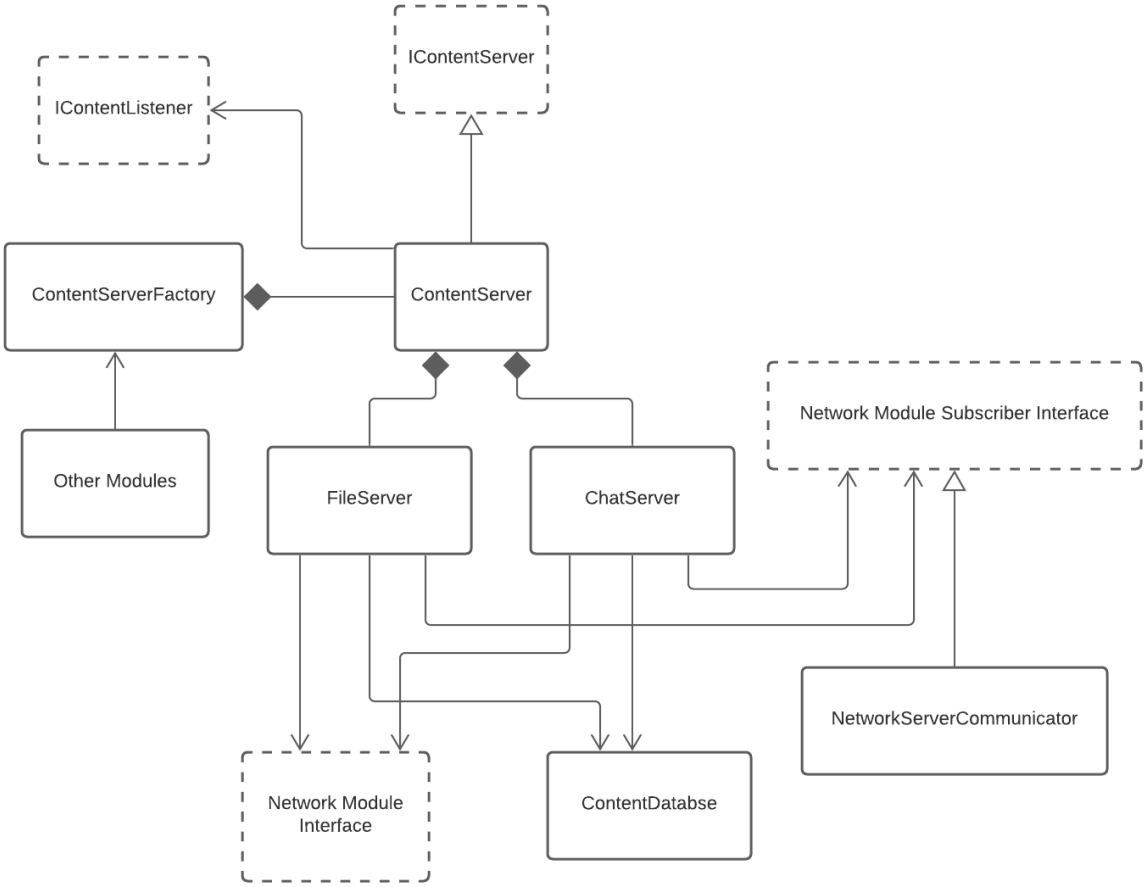
The ContentServerFactory class

Following the factory design pattern, we provide a singleton factory- the ***ContentServerFactory*** class, which composes a class that implements the ***IContentServer*** interface.
This factory will provide two static methods:

- static void setUser(int userId): This needs to be called before obtaining an object to set the userId of the Server. This function needs to be only called once to initialize the state of the static ***ContentServer*** class object.

- static IContentServer getInstance(): This function will return an object implementing the ***IContentServer*** interface. Since the factory follows the singleton design pattern. Only one object will be created and the same object will be returned each time this function is called.

Class Diagram


IContentServer
IContentListener
ContentServerFactory
ContentServer
Network Module Subscriber Interface
Other Modules
FileServer
ChatServer
NetworkServerCommunicator
Network Module Interface
ContentDatabse