CS4032: DISTRIBUTED SYSTEMS
LAB 3: DESIGNING A CHAT SERVER

David Kelly
Ben Lynch
Cian McElhinney
Alex Mooney
Jasmine Talukder
Padraic Rowley
Caleb Prior

Introduction

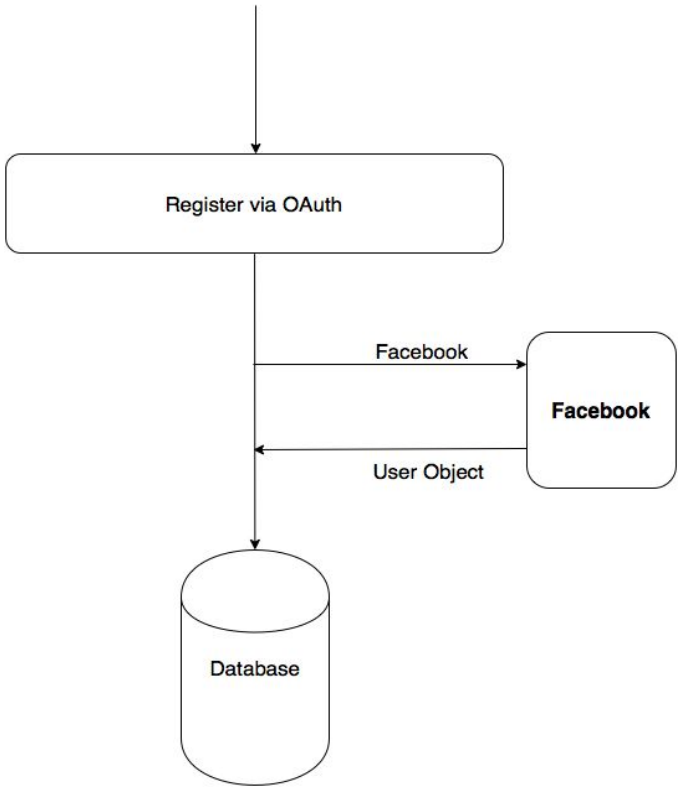
We looked at a number of different protocols and methods for construction our architecture.
In the end we chose a predominantly pulling RESTful method, similar to that used by many simpler messaging services or email clients like POP3 or IMAP.
To ensure a lower latency we chose to implement a long polling notification center.

When the user firsts connects they would be sent to sign in with their Google account. This will provide use with a unique method of identifying each user.
A main server will be used to serve REST requests.
The client will be able to make a request to the server asking for messages and their metadata relating to them. The client can make a request to the same server to add a new message to a conversation or create one.
All of these requests will be authenticated.
With this approach alone the client could continually poll this server to see if any new messages had arrived. This would be very inefficient and costly to the both the user and the server in terms of processing power and bandwidth.

To reduce these problems, a long polling approach will be used to alert the user if any new messages arrive for them, making our solution a pull/push hybrid.
This will use a long polling queue system similar to that of Amazon's SQS.
This will consist of the user will make a request to the server with a large timeout value set, for example 60 seconds, if during this time the user receives a message in any of their conversation the notification center will push the message to the client, next the client can then pull extra information, if needed, from the main server. Then the process is started again.
If the long polling times-out and the connection is broken then the client will make a new connection to the notification center and wait for any responses.

The main advantage to our choice is that it is easy to develop and debug. Have a pulling service as the backbone of our applications means that it will work on older devices or devices with worse connectivity as they don't always have to be online. Memory will be saved by not having to keep all sockets open, allowing for horizontal scaling. It will be simple to manage connected devices as they disconnect after short periods of time. Finally, the server stores chat history allowing for latecomers to view previous conversation messages.

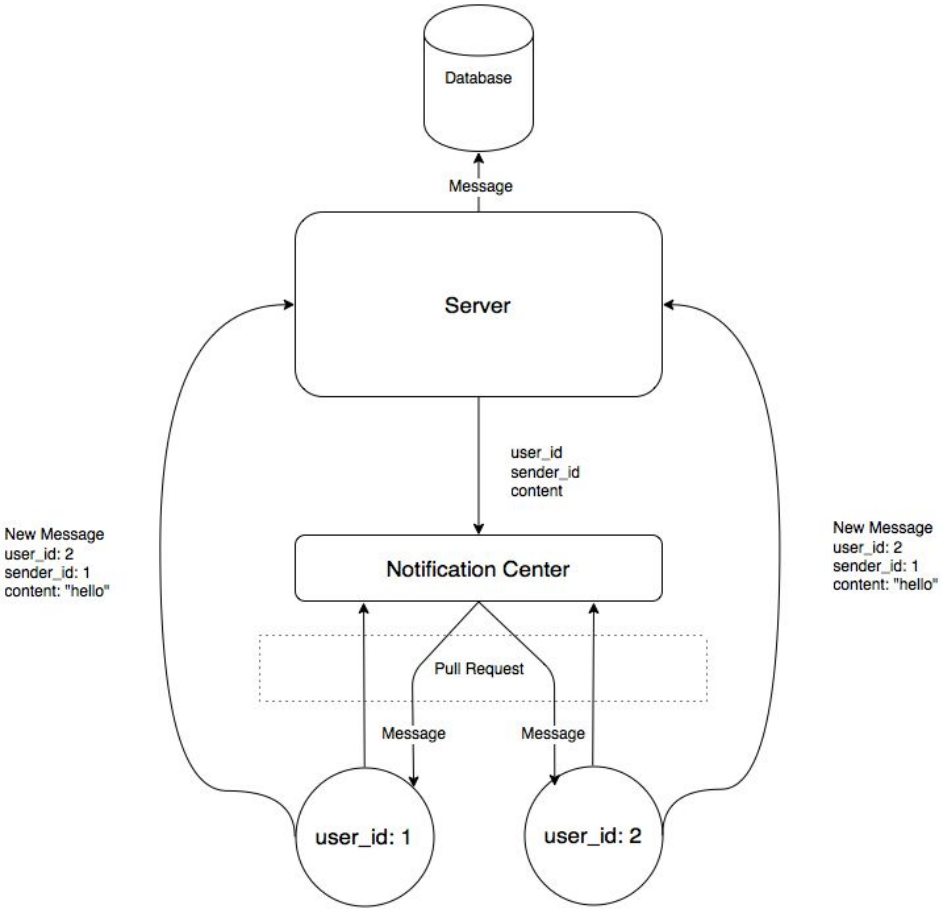
Authentication



Third party authentication
Using OAuth, register with the messaging app via a third party e.g. Google. After registering the application with Google, we will receive a client ID and a client secret. When the user first visits the site they are redirected to Google's authentication URL with our client ID passed as a GET variable.
The user will be prompted to authorize our application. This will redirect them to our server with a code as a GET parameter. Our server will then POST this code to Google along with our client secret in exchange for an access token. Providing everything is valid will will receive an access token for that user. This token can then be inserted as an Authorization header while making requests querying that user's data. This will return a User Object with id, username and email Returns.
User object is stored in database with record as so:

```
id: integer
username: string
email: string
```

Private Messaging

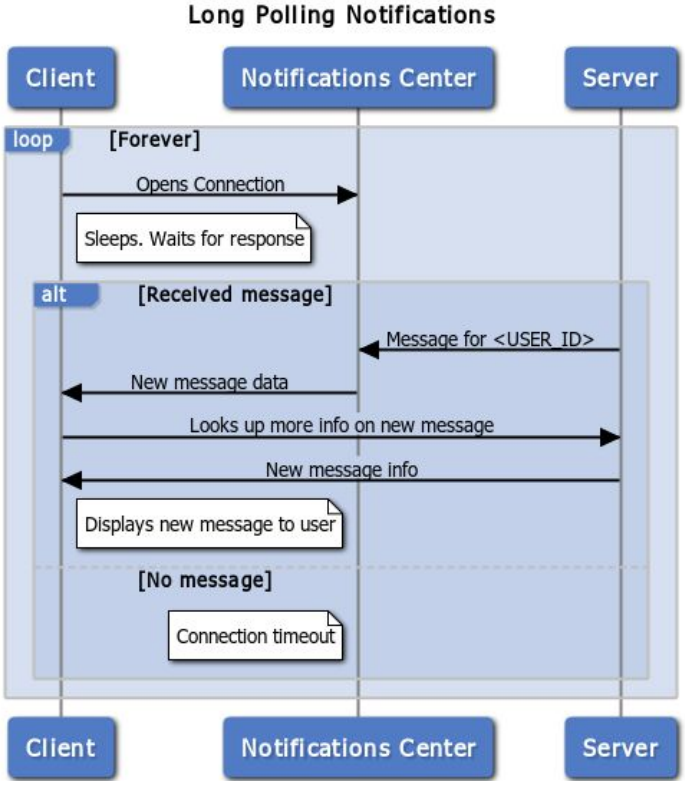


Private messaging:
Users will add people based on their username to their friends list. Private messaging occurs when a User sends a message to a single user_id as seen in the diagram. Users will long poll the notification center server for updates in messages with their user_id.

Long polling

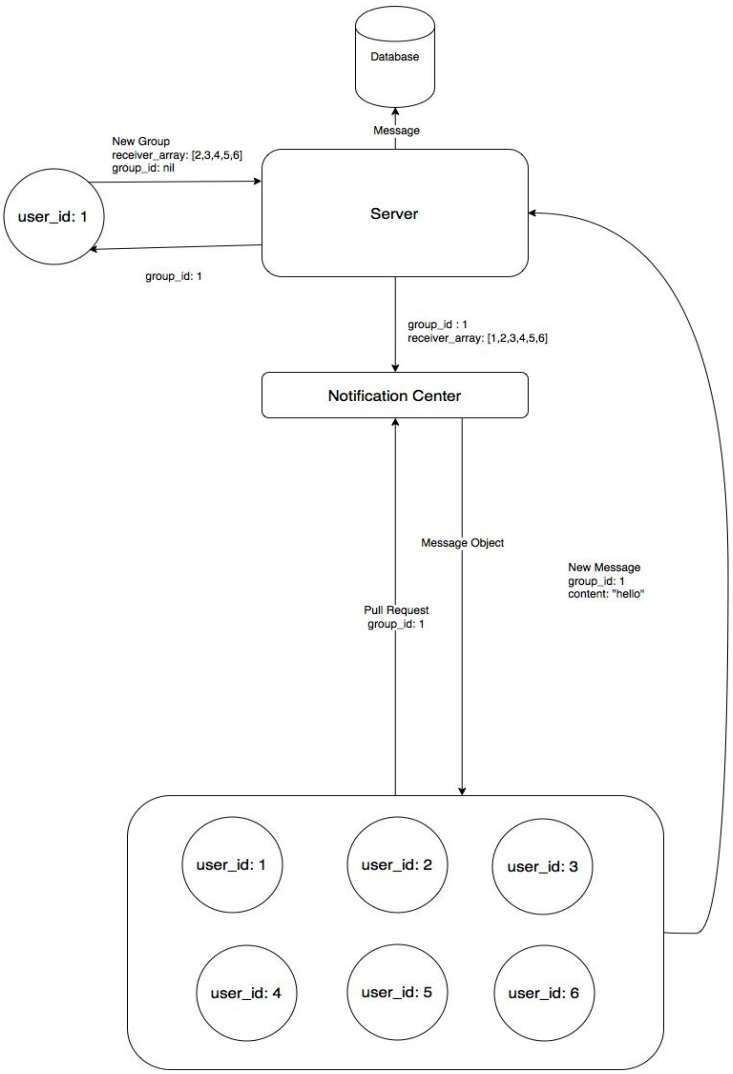
As discussed previously, a long polling solution was needed to increase the efficiency of our program and reduce latency. This involves the client opening a socket connection to the Notification Center where the client will have a large timeout on that request. If a message is received by the main the main server, it will send a TCP request to the Notification Center say that there's a new message for a given user. This will contain the target user and where this message can be found. The Notification Center will act as a queue and send that information to the client if they are connected. If the client is not connected, the next time it connects, it will be notified with the new message. The client will then open a TCP socket to the main server to get relevant information about the new message and then display it to the user. This process is then repeated regardless of messages being received.

long polling sequence diagram and pseudocode.



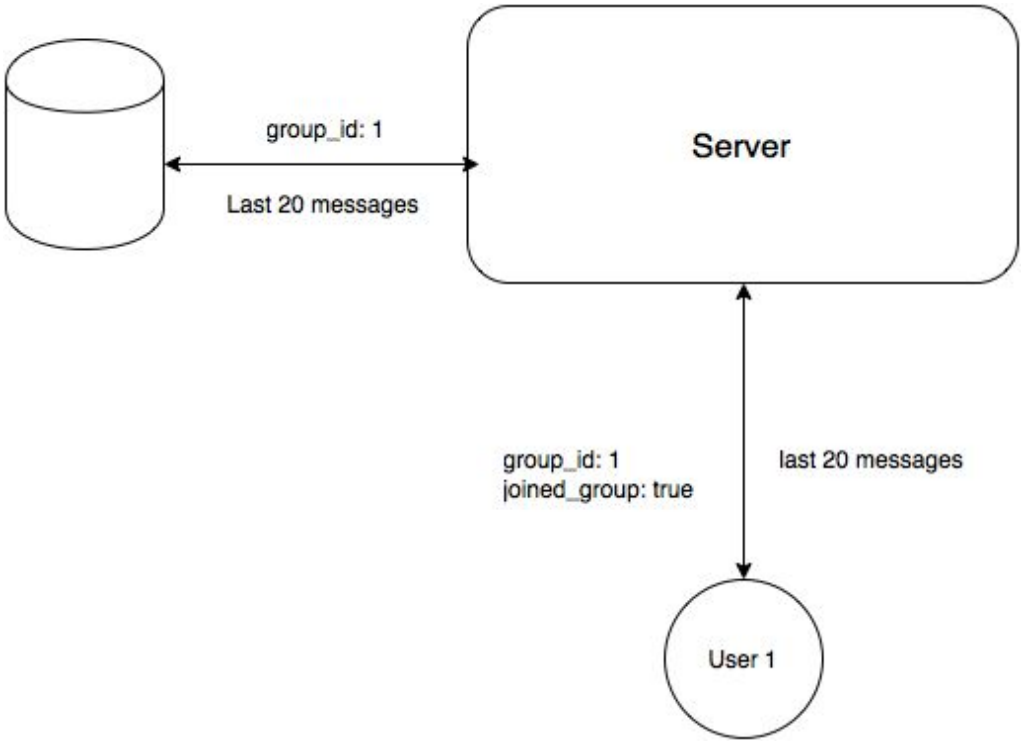

```
def listening_thread():
    while true:
        data = long_poll()
        if data:
            new_messages = get_meta(data)
            display(new_messages)
```

Group messaging



Group messaging:
A User will initiate a group chat by sending a message to an array of user_id's. The Server will return a group_id which references all the user_id's in the database. A notification of a group_id will be created in the notification center for all the user's. A user will then send messages to the group_id and the server will update the notification center accordingly.

Conversation History



Conversation History:
If a User joins a group at a point after conception and messaging, an existing user will send their id and the group_id to the Server which will add it. When the added User polls the notification center, they will make a request for the last 20 messages to have been sent. Then if they scroll to the top of the screen, we will fire a trigger to retrieve another 20 messages and append them to their chat.

RESTful Service

The chat service is based on REST and uses HTTP as its underlying protocol. Resources on the server are represented using JSON. The server resources, available methods for each resource, URIs and descriptions are available in the below table.

Resource	Methods	URI	Description
User	GET, POST, PUT, OPTIONS	http://ChatServer/Users/{UserId}	Contains information about a user {UserId} is optional **Format:** json
Group	GET, POST, PUT, DELETE, OPTIONS	http://ChatServer/Groups/{GroupId}	Contains information about a group. A group can be joined by multiple users **Format:** json
Search	GET	http://ChatServer/Search?	Search for a user or a group **Format:** json **Query Parameters:** Name: String, Name of a user or a group Type**:** String, optional, User or Group. If not provided then search will result in both User and Group
Me	GET, OPTIONS	http://NotificationCenter/me	Contains information about a particular user's notifications **Format:** json

Message Examples

Client-server message examples

Send private message:
POST http://ChatServer/Users/{UserId}
<Request Body>: {“name”:”sender”, “message”: “msg”}

Pull private chat (pulls all private messages since last being active):
GET http://ChatServer/Users/{UserId}/conversation
<Request Body>: {“sender” : userId}

Response:
{“messages” : [message1, … , messageN]}

Create group:
PUT http://ChatServer/Group/{GroupId}
<Request Body>: {“members” : [member1 ,...., memberN]}

Pull group info:
GET http://ChatServer/Group/{GroupId}
{}

Response:
{“groupID” : groupId, “members” : [member1, … , memberN]}

Send group message:
POST http://ChatServer/Group/{GroupId}/conversation
<Request Body>: {“userId” : userId, “message” : msg}

Pull group chat:
GET http://ChatServer/Group/{GroupId}/conversation?Index=requiredIndex
{}

Response:
{“messages” : [{“senderId” : senderId, “message” : message}, ….}

Search for group:
GET http://ChatServer/Search?Name=GroupId&Type=Group
{}

Long polling:
GET http://NotificationCenter/me
{}

Message Examples

Notification Center message examples

The below examples are responses to GET requests sent via long-polling

GET http://NotificationCenter/me

Private message notification:

{“notification” : “new_private_message”, “from” : username}

Group message notification:

{“notification” : ”new_group_message, “group” : group_name}

Group created notification:

{“notification” : ”group_created, “group” : group_name}

Conclusion

We've looked at different existing applications before designing our chat server to get inspirations such as WhatsApp and Facebook Messenger. WhatsApp uses a customized version of XMPP while Facebook Messenger uses MQTT Protocol. The advantage of using XMPP is that servers can be isolated and Simple Authentication and Security Layer (SASL) and Transport Layer Security (TLS) encryption have been built into the core XMPP specifications. Saying that, XMPP doesn't support Quality of Service(QoS) nor end-to-end encryption. MQTT also has its advantages. MQTT are open standards and it's better suited to constrained environments than HTTP. It also offers sufficient reliability and it's lightweight on the client and on the wire. MQTT has it's drawbacks. In MQTT, each client must support TCP and will typically hold a connection open to the broker at all times. This can be a problem in environments where packet loss is high or computing resources are scarce

Having considered all protocols above and more, we decided to use the RESTful method. A lot of major web services on the internet now use REST. Examples of these are Twitter, Flickr and Yahoo's web services. REST is very lightweight, and relies upon the HTTP standard to do it's work. It produces human readable results and it is easy to build as it requires no toolkit. These are major reasons why we chose REST.

To summarize, users register using oAuth to log into the chat application using their Google account. A main server is used to serve REST requests and to ensure a lower latency, long polling is used to alert users of new messages.We have a pulling service as the backbone of our application. In client pull, a HTTP connection is never held open. Instead, the client is told when to open a new connection, and what data to get.