highscalability.com

Designing WhatsApp - High Scalability -

16-20 minutes

This is a guest post by Ankit Sirmorya. Ankit is working as a Machine Learning Lead/Sr. Machine Learning Engineer at Amazon and has led several machine-learning initiatives across the Amazon ecosystem. Ankit has been working on applying machine learning to solve ambiguous business problems and improve customer experience. For instance, he created a platform for experimenting with different hypotheses on Amazon product pages using reinforcement learning techniques. Currently, he is in the Alexa Shopping organization where he is developing machine-learning-based solutions to send personalized reorder hints to customers for improving their experience.

Problem Statement

Design an instant messenger platform such as WhatsApp or Signal which users can utilize tosend messages to each other. An essential aspect of the application is that the chat messageswon’t be permanently stored in the application.

FUN FACT: Some of the chat messengers such as FB Messenger stores the chat messages unless the users explicitly delete it. However, instant messengers such as WhatsApp don’t save the messages permanently on their server.

Gathering Requirements

The instant messenger application should meet the following requirements.

- It should be able to support one-on-one conversations between users.
- It should be able to show Sent/Delivered/Read confirmation to other users.
- It should be able to provide information about the last time a user was active.
- It should allow users to share images.
- It should be able to send push notifications to other users.

Capacity Planning

We need to build a highly scalable platform which can support traffic at the scale of WhatsApp.Additionally, while doing capacity planning, we need to ensure that we think through the worst-case scenarios of peak traffic. Some of the numbers which we can use for capacity estimations ofan application (like WhatsApp) are listed below.

- Number of users on the application every month: 1 Billion
- Number of active users per second at peak traffic: 650, 000
- Number of messages per second at peak traffic: 40 Million

The entire application will comprise of several microservices each performing a specific task.The number of servers required in the data plane handling the traffic of chat messages can beestimated using the following equation.

$\#servers\ in\ chat\ microservice$= (#chat messages per second$*$ Latency)/ #concurrent connections per server

Let’s assume that the number of concurrent connections per server is 100K, and the latency of sending a message is 20 milliseconds. In such a scenario, the estimated number of servers required in the chat servers’ fleet (using the equation mentioned above) will be 8 (i.e., 40 Million*20 ms/100K). In standard practice, it’s recommended to add a few more servers to account for handling failures of these servers. In a subsequent section, we will see the impact which these chat servers will have on the overall infrastructure cost

FUN FACT: In this talk, Rick Reed(software engineer @ WhatsApp) talks about optimizing their Erlang-based server applications and tuning the FreeBSD kernel to support millions of concurrent connections per server. This helped them to a great extent in keeping their server footprint as small as possible

High Level Design

The required features of this instant messenger application can be modeled using two micro- services: Chat service and Transient service. The Chat Service will be the one serving the traffic of online chat messages sent by active users. The service will check if the user to whom the message is sent is online or not. If the user is online, then the message will be forwarded to that user instantly. Otherwise, the message will be handled by the Transient Service. This service will be responsible for maintaining all the messages (text or image) sent to offline users. The data will be stored in the Transient Storage temporarily until the offline user comes back online. We will provide more details about the individual components in one of the later sections.

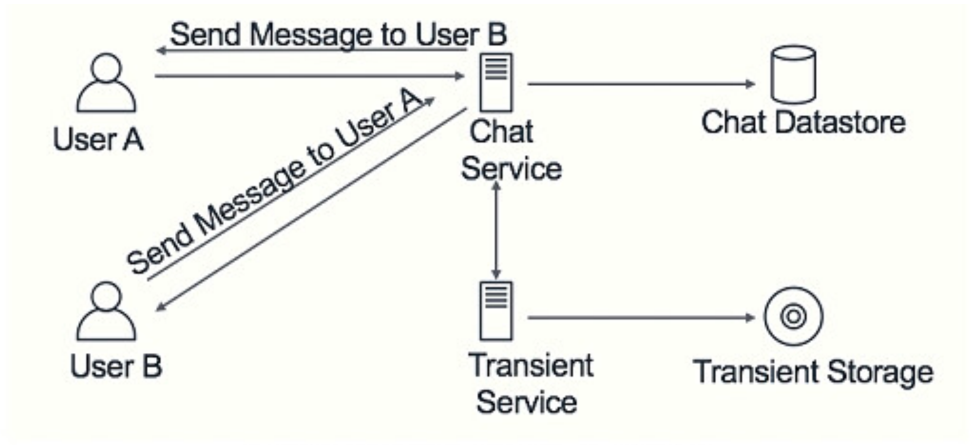


Fig 1: High Level Design - WhatsApp

FUN FACT: WhatsApp actually uses a much similar approach as discussed by the sameWhatsApp engineer(Rick Reed) in a [different talk](#).

API Design

We can expose a REST endpoint to interact with the Chat Service. The definition of the APIendpoint to send messages is mentioned below.

sendMessage(String fromUser, String toUser, ClientMetaData

clientMetaData, String message)

Request:

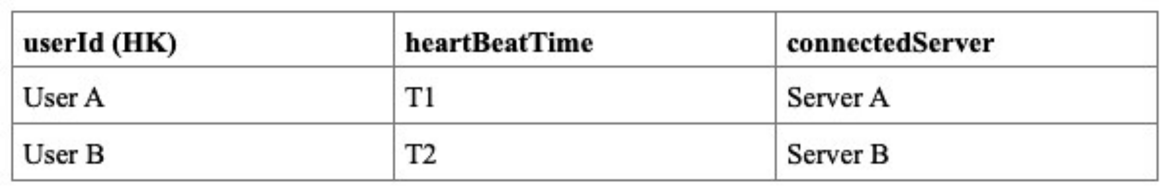
fromUser:The userId who is sending the request

toUser:The userId to whom the request is being sent

clientMetaData: The metadata to store client's information such as device details, locations etc.

message: The message being sent as part of the communication.

Data Model



userId (HK)	heartBeatTime	connectedServer
User A	T1	Server A
User B	T2	Server B

Table 1: Data Model – User Info

Component Design

In this section, we will talk about two different scenarios for sending messages **in a one-to-one communication**. After that, we will discuss the other features which we need to support, such as push notifications and user activity status. In the end, we will look into the different mechanisms for doing optimizations and handling failure scenarios.

One-to-One Communication

Here, we will talk about the two different scenarios associated with sending messages to another user. The first scenario involves sending a text message to an online user. In the second scenario, we have described the sequence of operations involved in sending an image to an offline user.

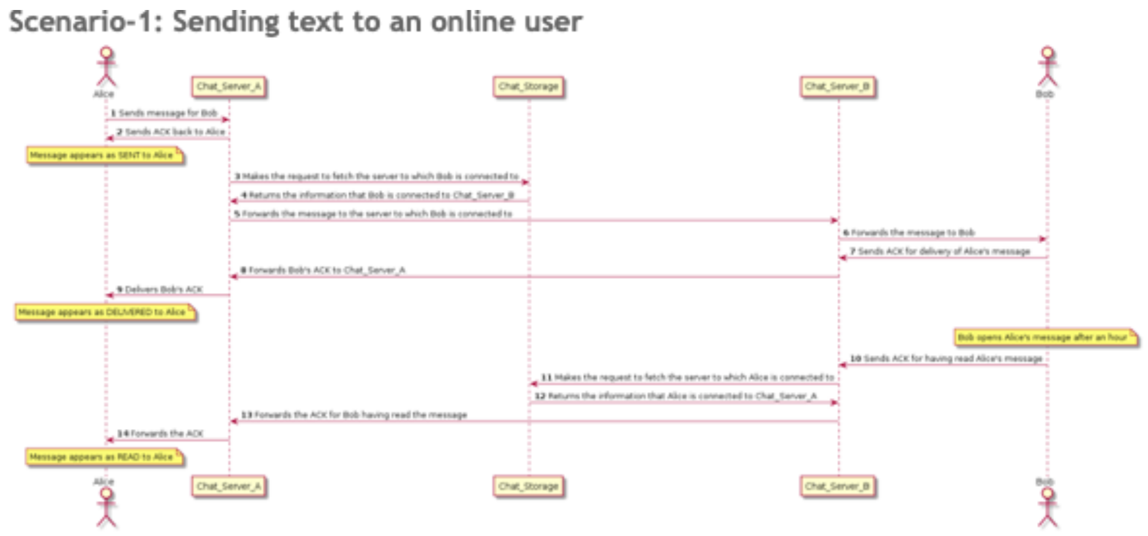

Scenario-1: Sending text to an online user
Alice
Chat_Server_A
Chat_Storage
Chat_Server_B
Bob
1 Sends message for Bob
2 Sends ACK back to Alice
Message appears as SENT to Alice
3 Makes the request to fetch the server to which Bob is connected to
4 Returns the information that Bob is connected to Chat_Server_B
5 Forwards the message to the server to which Bob is connected to
6 Forwards the message to Bob
7 Sends ACK for delivery of Alice's message
8 Forwards Bob's ACK to Chat_Server_A
9 Delivers Bob's ACK
Message appears as DELIVERED to Alice
Bob opens Alice's message after an hour
10 Sends ACK for having read Alice's message
11 Makes the request to fetch the server to which Alice is connected to
12 Returns the information that Alice is connected to Chat_Server_A
13 Forwards the ACK for Bob having read the message
14 Forwards the ACK
Message appears as READ to Alice

Fig 2: One-to-One text communication between online users

The details about each of the steps in the sequence diagram for sending a text message to an online user is mentioned below.

- Step 1: Alice sends a message to Bob which gets directed to the chat server with which Alice is connected.
- Step 2: Alice gets an acknowledgement from the chat server it's connected(i.e. Chat_Server_A) and the message is marked as SENT at Alice's end.
- Step 3: Chat_Server_A makes a request to the data-storage to fetch information about the chat server to which Bob is connected.
- Step 4: Chat_Storage returns the information that Bob is connected to Chat_Server_B.
- Step 5: Chat_Server_A forwards the message to Chat_Server_B.
- Step 6: The message gets delivered to Bob using a push mechanism.

- Step 7: Bob sends ACK back to Chat_Server_B.
- Step 8: The ACK is forwarded to Chat_Server_A to which Alice is connected.
- Step 9: The ACK gets delivered to Alice and is marked as DELIVERED.
- Step 10: Another ACK is sent to Chat_Server_B when Bob reads the message (let’s say after 15 minutes).
- Step 11: Chat_Server_B requests to fetch the server to which Alice is connected.
- Step 12: Chat_Storage returns the information that Alice is connected to Chat_Server_A.
- Step 13: Chat_Server_B forwards the read ACK to Chat_Server_A.
- Step 14: The ACK is forwarded to Alice marking the request as READ.

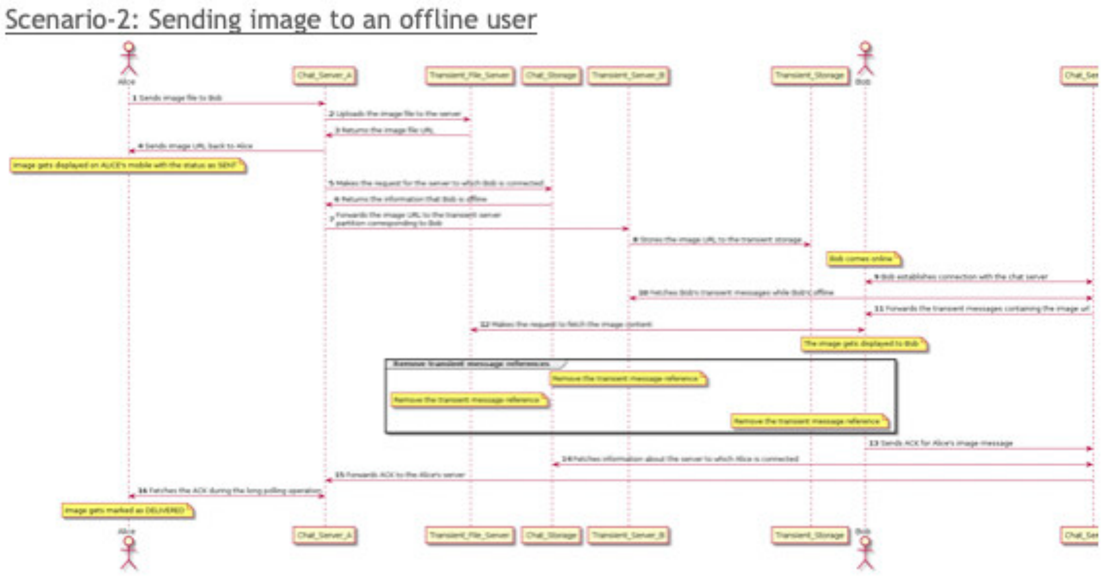

Scenario-2: Sending image to an offline user
Alice
Chat_Server_A
Transient_File_Server
Chat_Storage
Transient_Server_B
Transient_Storage
Bob
1 Sends image file to Bob
2 Uploads the image file to the server
3 Returns the image file URL
4 Sends image URL back to Alice
Image gets displayed on ALICE's mobile with the status as SENT
5 Makes the request for the server to which Bob is connected
6 Returns the information that Bob is offline
7 Forwards the image URL to the transient server partition corresponding to Bob
8 Stores the image URL to the transient storage
Bob comes online
9 Bob establishes connection with the chat server
10 Fetches Bob's transient messages while Bob's offline
11 Forwards the transient messages containing the image url
12 Makes the request to fetch the image content
The image gets displayed to Bob
Remove transient message references
Remove the transient message reference
13 Sends ACK for Alice's image message
14 Fetches information about the server to which Alice is connected
15 Forwards ACK to the Alice's server
16 Fetches the ACK during the long polling operation
Image gets marked as DELIVERED

Fig 3: Sequence Diagram: Sending an image to an offline user

The details about each of the steps in the sequence diagram for sending an image to an offline user is mentioned below.

- Step 1: Alice sends an image to Bob which gets forwarded to Chat_Server_A, the server with which Alice is connected.
- Step 2: Chat_Server_A uploads the image to a File Server where the file stored in a directory structure
- Step 3: File Server returns the image url of the uploaded file to Chat_Server_A.
- Step 4: The image url is returned to Alice which is used to render the image on Alice's device. The image is marked as SENT on Alice's end.
- Step 5: Chat_Server_A makes request for the server to which Bob is connected.
- Step 6: Chat_Storage returns information that Bob is offline.
- Step 7: Chat_Server_A forwards the message containing the image-url to the transient server
- Step 8: The transient sever stores the message containing the image-url in transient storage.
- Step 9: Bob comes online and performs heart-beat with Chat_Server_B.
- Step 10: Chat_Server_B fetches transient messages for Bob from the transient server.
- Step 11: Chat_Server_B forwards the transient messages to Bob.
- Step 12: Bob fetches the image from the file server. At this point, the image gets delivered to Bob's device and all the references to the transient messages are removed from the system.

- Step 13: Bob’s device sends ACK for Alice’s image to Chat_Server_B
- Step 14: Fetches information about the server to which Alice is connected; i.e. Chat_Server_A
- Step 15: Forwards the ACK to Chat_Server_A
- Step 16: The ACK gets delivered to Alice marking the message as DELIVERED.

Transient Data Storage

We can implement a queue-based mechanism to store and retrieve the transient messages using a FIFO based policy. We can use existing cloud-based technologies for this purpose, such as Amazon SQS or Windows Azure Queue Service. We can use these queues to store transient messages sent to offline users. All the references to these transient messages are removed from the system once the messages are delivered to the offline user.

Push Notifications

The are two approaches to deliver messages to users by using push technology : client pull or server push. If we go down the route of client pull, we can either decide between long vs. short polling. On the other hand, there are two ways to implement the server push approach: WebSocket and Server-Sent Events(SSE). Websockets has been the de-facto communication protocol for chat applications. We have provided more details about it in the section below.

Using the polling technique, the client asks the server for new data regularly. The trade-off decision to choose the polling technique can be taken using the data-points mentioned below.

- **Short Polling (e.g., AJAX-based timer)**
- **Pros**: more straightforward and not very server consuming if the time between requests is long
- **Cons**: not ideal for scenarios when we need to be notified of server events with minimal delay
- **Long Polling (e.g., Comet based on XHR)**
- **Pros**: notifications of server events happen with no delay
- **Cons**: more complex and consume more server resources

The approach to push server messages to clients is mainly of two types. The first one is WebSocket which is a communication protocol. It provides duplex communication channels over a single TCP connection. It's ideal for scenarios such as chat applications due to its two-directional communication. The other one is called Server-sent events (SSE) which allows a server to send "new data" to the client asynchronously once the initial client-server connection has been established. SSEs are more suitable in a publisher-subscriber model such as real-time streaming stock prices; twitter feeds updates and browser notifications.

User Activity Status

The last time when a user was active is a standard functionality which can be found on instant messengers. We have shown the data-model to store the related information in Table 1 above.

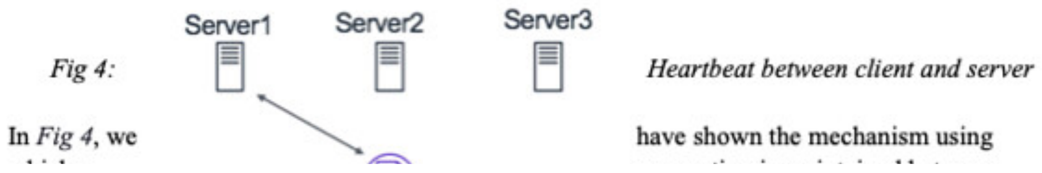
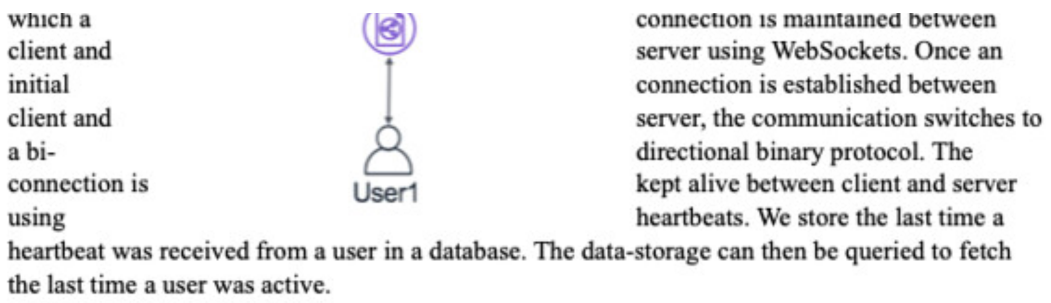


Fig 4: *Heartbeat between client and server*

In *Fig 4*, we have shown the mechanism using



which a connection is maintained between client and server using WebSockets. Once an initial connection is established between client and server, the communication switches to a bi-directional binary protocol. The connection is kept alive between client and server using heartbeats. We store the last time a heartbeat was received from a user in a database. The data-storage can then be queried to fetch the last time a user was active.

Optimizations

We can use the parameters listed below to suggest optimizations in the system.

- **Latency**: We can use distributed cache such as Redis to cache the information of user activity status and their recent chats in-memory. This may help in reducing the overall latency of the application and provides a better customer experience. Even some of the database solutions do provide in-memory caching solutions such as Amazon DynamoDB Accelerator.
- **Infrastructure Cost**: It's apparent from the system that significant contribution to the infrastructure cost will be from chat servers. The cost incurred from the chat servers can mount quickly if its server foot-print isn't kept in check. One way of doing that is to increase the number of connections per host. It will significantly reduce the number of servers required for maintaining the service. We can accomplish this task by tuning the server configurations and choosing suitable technologies. For instance, engineers at WhatsApp were able to achieve millions of connections per host by optimizing their Erlang-based server applications and tuning the FreeBSD kernel
- **Availability**: We can maintain multiple copies of the transient messages so that even if the messages in one of the copies is lost

then it can still be retrieved from the other copy. This would imply maintaining replicas of those transient messages. The client would be responsible for fetching the messages from two queues and will merge them. We will discuss more in the next section.

Addressing Bottlenecks:

The major bottlenecks in the system which are more vulnerable to failures are the chat servers and the transient storage solutions. We have recommended some approaches to handle such failures in the section below.

- **Chat Server Failure**: The chat servers in the system will be holding connections with the users. There are two ways of handling the failure of chat servers. One approach can be to transfer those TCP connections to another server; however, implementation of such fail-overs isn't trivial. The second, which is comparatively more straightforward is to have the user clients initiate the connection automatically in case of connection loss. The information of the server to which the user is connected needs to be updated in the database is agnostic of the approach we take. For instance, in Fig 5, we have shown an illustration of the handling this failure scenario. We can see that User1 is connected to Server1, and when this server goes down, then the connection is re-established with another server (i.e. Server2), and this information is updated in the database.

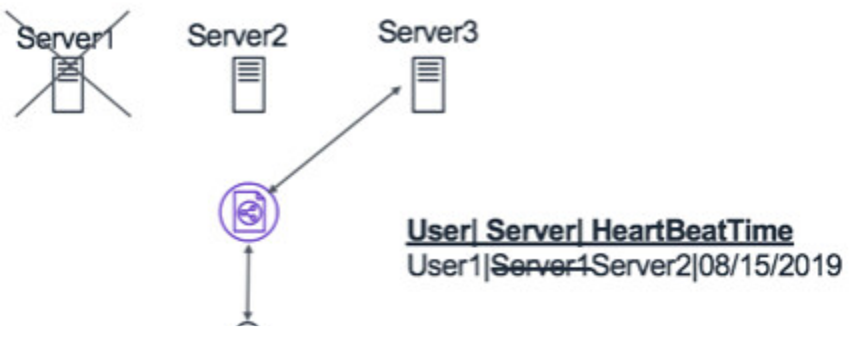

~~Server1~~
Server2
Server3
User| Server| HeartBeatTime
User1|~~Server1~~Server2|08/15/2019

Fig 5: Handling Chat Server Failure

- **Transient Storage Failure**: The transient storage is another component which is prone to failures and can cause loss of messages in-transit to offline users. We can make replicas of the transient storage of each user to prevent the loss of messages which were sent to them when they were offline. When a user comes back online then both the original and the replica instances of the user's transient storage are queried and merged. In Fig 6, we have illustrated a mechanism to handle transient storage failure which gets initiated when user comes back online.

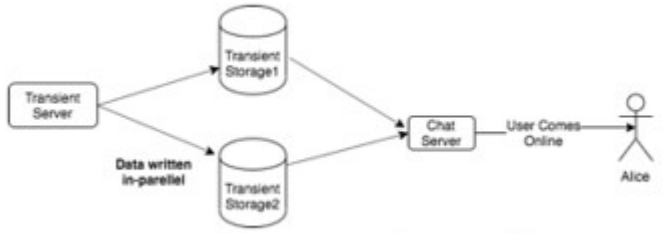


Fig 6: Handling Transient Storage Failures

Monitoring

We want to ensure that our service can meet the user demands with high availability and low latency. We can define service level agreements (SLAs) for these metrics and create moderate and severe monitors which can trigger an alarm when these SLAs are violated. For this application, we can define the following SLAs for the sendMessage API.

1. **Availability SLA**: p99.999
2. **Latency SLA:** p99.99 of 5 milliseconds

The availability SLA implies that that the monitor will trigger an alarm if more than 1 out of 1000 requests fail. Likewise, the latency SLA means that an alert will be triggered if the server takes >5ms to respond for more than 1 out of 100 requests it receives.

Additionally, we can put failure alarms in different error scenarios. One such scenario can occur when the chat server isn’t able to fetch transient messages for a user from all the replicas of transient storage. This maps to Step#10 illustrated in Fig 3 above where Chat_Server_B requests the transient server to fetch the messages sent to Bob when Bob was offline. Let’s assume that we maintain two copies of Bob’s messages in transient storage for making the system more robust. However, transient server isn’t able to retrieve messages from both the copies due to an interim issue with transient storage. This is an error scenario which needs to be debugged and so it requires adequate monitoring alarms.

Extended Requirements

We can extend the system to support group chats using which we can get messages delivered to multiple users. We can create a data-model to store the group data-entity, which will be identified by GroupChatID and will be used to maintain a list of people who are part of that group. The system described above is extensible to support the scenarios for sending message to online and offline users. We can build a component which will be responsible for ensuring that the messages get delivered to all the users in the group depending upon their activity status.

Each user has a public key that is shared with all the other users with whom the user is communicating. For instance, two users Alice and Bob, are communicating with each other.

Alice has Bob's public key and vice versa; however, their private key isn't shared. When Alice sends a message to Bob, the message is encrypted using Bob's public key and sent over the network. The server directs the encrypted message to Bob, who uses the private key to decrypt the message. In this way, the server only has access to the encrypted message, and only Alice and Bob can read the actual messages they exchanged.

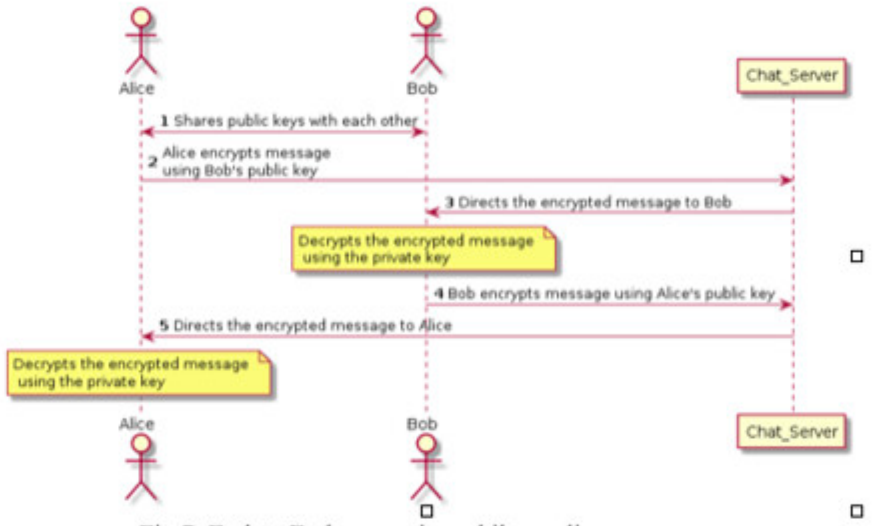

Alice
Bob
Chat_Server
1 Shares public keys with each other
2 Alice encrypts message using Bob's public key
3 Directs the encrypted message to Bob
Decrypts the encrypted message using the private key
4 Bob encrypts message using Alice's public key
5 Directs the encrypted message to Alice
Decrypts the encrypted message using the private key
Alice
Bob
Chat_Server

Fig 7: End-to-End encryption while sending messages

References

- https://developers.facebook.com/videos/f8-2016/a-look-at-whatsapp-engineering-for-success-at-scale/
- http://highscalability.com/blog/2014/2/26/the-whatsapp-architecture-facebook-bought-for-19-billion.html