AMENDED Exhibit 802

PLAINTIFFS' OMNIBUS OPPOSITION TO DEFENDANTS' MOTIONS FOR SUMMARY JUDGMENT

WIKI VERSION (more up to date): https://wiki.sc-corp.net/display/COMENG/The+Book+of+Messaging

The Book of Messaging

Author: [redacted]

What is Messaging at Snap?

Messaging includes all things that power the “left” side (second tab) of Snapchat–the Feed and Conversation views–and the unique experiences within those views. It doesn’t matter if you’re sending a Snap or a Text, viewing messages directly from the Feed or in the full Conversation view, receiving the message in real-time, via notification, or as a data pull–Messaging systems decide the why/when/what/how of decisions like these (and many others).

Here’s a complete list of systems, concepts, and application screens that belong to Messaging:

- Message model
- Conversation model
- Snap model
- Group model
- Sending and Receiving all Message Types
- Message Type Definitions (including Text, Snap, and all the others)
- Push Notifications and In-App Message Notifications
- Real-time Message Delivery
- 1:1 Conversations
- Group Conversations
- User Presence
- Feed View
- Conversation View

Conversations

Conversations are the space where users–which we call participants–send and receive messages. Users view messages in conversations on the Conversation View, which lists messages in chronological order, the bottom-most message being the most recent. Paginating up moves the view in reverse chronological order.

Conversations have both a history of messages (as long as they have not yet expired) and associated conversation metadata.

Group Conversations

Group conversations are probably what you think of if you create a mental model of a conversation. A group conversation has a list of participants, a title, a retention policy, and a long list of other properties. Participants can send messages, and the entire group receives them. Groups operate in 24 hour retention mode.

1:1 Conversations

In Snapchat, 1:1 conversations are distinct from group conversations. A 1:1 conversation is an implicit conversation between any two Snapchat users–because a friend link exists between two users, there also exists a 1:1 conversation. 1:1 conversations have implicit titles (the title is the other friend’s display name) and

can have special retention rules. 1:1 conversations operate in “delete after viewing” retention mode by default, but can be switched to 24 hour retention mode.

You can have a group conversation with just 2 participants in it, and while at the data model level the two are almost entirely the same, the group conversation is conceptually different because it has different behaviors in some cases.

Messages

Messages are the bread and butter of Messaging. Today in Snapchat, everything you send and receive is modeled as a message. Messages have many different types, such as Text, Snap, Media, Story Share, and a long list of many others. Each message is transported the same way regardless of whether it is text, has media, or some other interactive properties.

Snaps

Snaps are the special sauce of Snapchat–they are the primary way for users to communicate in the moment with media and creative tools. They disappear by default, an essential behavior that requires a lot of bookkeeping to get right.

Within Messaging systems, they are more or less just another type of message, but they have special interactions and retention behaviors. Snaps disappear after viewing and they can only stay on screen for their 10 second length (unless the Snap is set to no limit), regardless of the retention mode of the conversation they appear in. Users are allowed one replay of Snaps; we always notify the other participants when a Snap has been replayed. The same goes for screenshotting; if a recipient of a Snap takes a screenshot, the other participants will always be notified of the screenshot.

Snaps also have a significant impact on Feed Interaction.

Feed

The Feed is the list of all conversations a user participates in, both Group and 1:1. Users view the list of their conversations on the Feed View in most-recently-active order. Each Conversation is represented as a Feed Cell, with the most recently active cell at the top of the page. Paginating down moves the view in reverse chronological order, according to the last activity time.

The Feed is an important point of user engagement, due to the behavior of Feed Cells. Feed Cells have interaction shortcuts that allow users to engage with new content. Unread Snaps take precedence; when the user taps on a Feed Cell, it launches the media player directly. This is one of the highest engagement interactions within Messaging.

Message Retention and Ephemeral Messages

Ephemeral messages are key to Snapchat’s experience, and retention policies are how we describe and enforce when messages expire and are subsequently deleted. We go to great lengths to meet our privacy guarantees around data retention–both in our services and on user devices–and ensure message data and media is deleted when users expect it to be. When messages expire, they are immediately deleted from user devices, but may take longer to be deleted from service databases (due to async cleanup jobs).

More details can be found in Message Retention.

Notifications

Messaging includes Notifications, both in-app and push notifications. When a user receives a message, we also show a notification for the message. If they are in the app, they will receive an in-app notification. If they are not in the app, they will receive a push notification from the operating system.

Arroyo and the Times Before

Legacy Messaging

Legacy Chat Infrastructure provides lots of details about how Messaging used to work. Back in the day, we didn't have one messaging system - we effectively had three. First came Snaps (images and videos), which was the entirety of the app in the beginning and was implemented naively. Later, chat between friends was added, which was an entirely different system. This system was later evolved to support group chats, but groups were sort of bolted onto the original system. All of this was wrapped up in the legacy FSN monolith, and many of the design decisions were based on working around Google Datastore's limit of 1 write per second per key, which was too slow for a chat system. This combination of accretive implementations and weird limitations resulted in a system that was very expensive, slow, hard to operate, and difficult to extend and improve. Simple functionalities were tough to add because it'd need to be added to at least two systems - e.g. adding something to "Snaps" meant adding it to the 1:1 Snaps system and the Group Chat system.

- Legacy Chat Infrastructure

Arroyo Project and Launch

Chat Renovation Initiative (Arroyo) was a proposal made by engineers outside of the Messaging organization who wanted to completely revamp the Messaging system. This proposed a new, unified data model for all conversations (group and 1:1) where Snaps were just another message type. It prescribed a new, simpler RPC-based messaging service (replacing a combination of RPC and message-broker systems), breaking the realtime message notifications into its own generic system (Duplex), and rebuilding the client as a cross-platform C++ library to consolidate our complex client business logic. In addition, Arroyo was aimed at helping build out and be the star customer for a slew of new Snap platforms; it was a catalyst behind new systems like Service Mesh + API Gateway, the use of AWS instead of GCP, BOLT (media storage platform), ATLAS (user and friend data), Delta Force (structured storage), PNS (Push Notifications Service), gRPC and Protobuf, Golang, Kubernetes, and more.

The Arroyo project, which sought to build out this new system and migrate to it, took several years, and was delivered over a series of incremental milestones, starting with moving the Friend Feed (but keeping messages in FSN), then Group conversations, and finally 1:1 conversations. Along the way, we also had to help design and, in some cases, build out our dependencies listed above, and assemble a C++ client platform to mold our library upon.

Backwards Compatibility Layer and the Great Migration

To transition from the old system to the new, we built a backwards compatibility layer (BWC) in FSN. This layer is able to translate back and forth between our old and new data model, and enable the legacy APIs still used by older clients, to operate seamlessly on conversation and message data – whether it was still in the old system or if it had already been migrated to the new Messaging services. After validating that this conversion

and backwards compatibility layer was not losing any data via shadow testing and "diff" metrics, we had the confidence to start automatically moving data over to the new system one conversation at a time. Modern clients were aware of this transition and could independently choose, on a per-conversation basis, whether to use the new APIs or the old ones. After all data was migrated (over a period of many months), a client version was released which *only* used the new APIs; all legacy code was then deleted from the client. This migration was pulled off, in each phase, without any user-visible incidents – no users were aware that they had moved to another system. The backwards compatibility layer lives on in a simplified form (it no longer needs to support the legacy storage!) in FSN and the rest of our legacy chat infrastructure, and will continue to operate and require maintenance until the last client that is not capable of using the new APIs is deprecated (version 11.17).

B2

Life of a Snap

Let's walk through the things that happen when someone sends a Snap today, before getting into more details about the moving parts. This should help paint a picture of the many steps we take to send and receive a Snap.

1. Snap is created locally
2. Media is uploaded
3. Snap is sent to MCS (through the API Gateway!)
4. MCS claims the media in BOLT
5. MCS saves the message in DynamoDB, returns success
6. MCS writes to each participant's feed
7. MCS sends a Push notification in the background
8. MCS sends a Duplex notification in the background
9. Recipient gets the push notification, and the duplex notification if they're in the app
10. App triggers a conversation sync, which gets the new messages
11. Media is downloaded from BOLT
12. User taps on the message, after which we send a release/read
13. MCS calls BOLT to release the media
14. MCS marks the message itself released
15. Eventually it gets deleted by a mapper or something

Message Lifecycle

Sending and Receiving

Messages exist in one of two primary states. They are either pending (in the process of being sent) or committed (part of the conversation history). Committed messages follow all the rules of Message Retention outlined below.

- Deep Dive on Message Lifecycle

Message Retention

There are three primary retention policies that we use in Snapchat. The first two are conversation-centric and are policies set at the conversation level.

Page 4 Comments

Q1 Is it a good idea to present both Life of a Snap and the Messaging Life Cycle as part of regular company wide new hire onboarding ?

[redacted] *3/1/2022 04:09 PM*

B2 I think it would be good to present Life of a Snap as part of new hire onboarding, yeah.

[redacted] *3/8/2022 03:26 PM*

- **Time-based**. Expire and delete messages after every participant has viewed the message, and after the specified amount of time (typically 24 hours). This policy is set for all messages in a conversation.
- **Delete after view**. Expire and delete messages immediately after participants have viewed the message. This policy is set for all messages in a conversation.

The last is message-centric.

- **Snaps**. Snaps have their own rules that supersede the policy on the conversation they occur in. Snaps expire after viewing and only stay on the screen for their 10 second length (unless set to no limit).

Conversations also have unread retention rules that govern how messages that have never been viewed before expire.

- 1:1 conversations have a 31 day unread retention period.
- Group conversations can have a 7 day retention period.

There are some other rules worth mentioning that impact message retention.

- Messages can be saved (and unsaved), bypassing retention rules altogether. See Saving Messages.
- Messages with media are not considered to be viewed until the media has both been loaded and launched in the player.
- Messages can be explicitly deleted, triggering an immediate expiration of the message.

More details on how both the client and server deal with retention can be found in the following documents.

- Deep Dive on Server Message Retention
- Overview of Client Message Retention

Saving Messages

Participants in a conversation can explicitly save any message, bypassing all rules of retention above. Saving a message instructs Snapchat systems to retain the message forever. If any user in a conversation saves a message, it is retained for all participants in the conversation. Retention rules will only kick in again if all users who have previously saved a message explicitly unsave the message.

Data Model

Conversation

Conversations are (conceptually) a collection of messages and a set of conversation metadata. In practice, the collection of messages is modeled as the set of messages sharing the same conversation ID, either packed into a list or at rest in a database.

Conversation metadata includes properties that describe the conversation (e.g. ID, creator, version creation time, participant list), affect behavior (e.g. type, version, retention policy), and affect display (e.g. title). The conversation version is an important piece of metadata that plays a role in Conversation Versioning.

- ConversationMetadata Definition

Messages

Messages are a set of identifying information, a set of policies governing behavior, a ContentEnvelope, and a set of message metadata. Messages are globally and uniquely identified by a (message ID, destination) tuple. Other identifying information includes the user ID of the sender.

- ContentMessage Definition

Destinations

Each message has a delivery destination, which is a type + ID combination that identifies where the message should go. Backend systems use this information to route the message to its destination. In most cases, this is just a conversation ID, but we also support other Stories as a delivery destination. In the future, we expect we might add other destinations, such as phone numbers.

- DeliveryDestination Definition

Message Metadata

Message metadata includes properties that track things like which participants have reacted to the message, which participants have seen the message, which participants have saved the message, which participants have screenshotted the message, and which participants have replayed the message (in the case of Snaps). Other properties can govern how the message should behave. For example, once every participant has viewed a message, it will expire (and be deleted) if the retention policy is set to “delete after viewing”.

- MessageMetadata Definition

Envelope

Snapchat uses what we call a Postal Service model, where for each message, you put the “From:” and “To:” on an envelope and we take care of delivering it. We never see what is in the envelope–neither backend systems nor the core C++ client have knowledge of what is inside. This allows us to treat Text, Snaps, and all other message types essentially the same, and makes it really easy to add new message types in the future. In practice, this means the payload of the envelope is a blob. This model is convenient for things like content encryption– since we don’t care what is inside, we can encode it in any way we want.

- ContentEnvelope Definition

Message Types and Message Content

Message content is effectively the view model for each message we display. Inside each envelope is the message content, an instance of one of many message types. When a client sends a message, it creates a content package, wraps it into an envelope, and packs it inside of a message. When a client receives a message, it follows this process in reverse, effectively unpacking the message content.

Remember, neither backend systems nor the core C++ client have knowledge of the content–only the presentation layers of the client stack need to deal with (both inbound and outbound) message content.

- Message Content / List of Message Types

Feed

Feed Cells are the primary data structure on the Feed and contain all the information required to order, display, and interact with the Feed.

- ConversationEntry definition

Feed Order

The Feed Cell order is determined by the last activity time. In general, activity occurs when a new message arrives in a conversation.

Feed Display

The Feed Cell display is affected by the most recent event that occurred in a conversation–it updates much more frequently than the last activity time that forces a conversation to “move up” in the feed. Pretty much any event in a conversation–reactions, screenshotting, or even simply viewing a message–updates both the Feed Cell display and the display timestamp.

The rules that govern the Feed Cell display are very complex; more information on them can be found here.

Feed Interaction

When a user taps on a Feed Cell, many different interactions can occur. Which interaction occurs is determined primarily by the type of the message that is the most recently received unviewed message, with several caveats. For most message types, tapping on a Feed Cell will take the user into the Conversation View to view the most recent messages. For Snaps, however, tapping on a Feed Cell will immediately start playback in the media player until all unviewed Snaps in the conversation have been viewed.

The rules that govern the Feed Cell interactions are complex; more information on them can be found here.

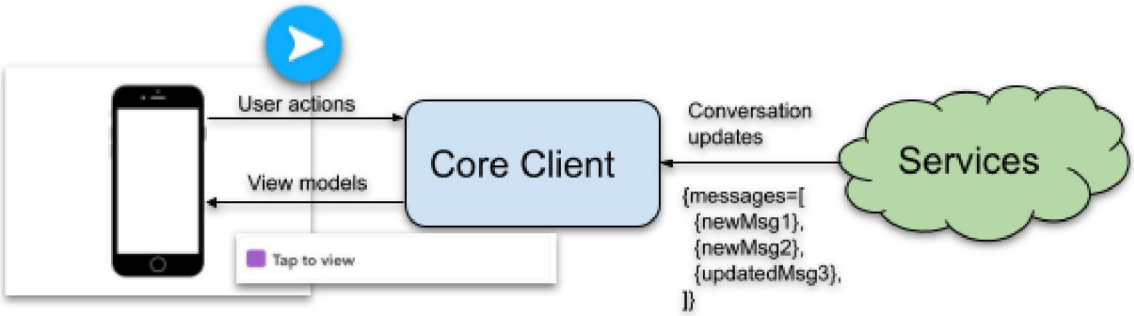
Client Architecture

Overview

The core client for messaging is built in cross-platform C++ and lives in the Snapchat/client monorepo. We’ve tried to encapsulate as much of the messaging business logic as possible in this library so that we write the code once and only once. This is especially beneficial for Desktop Web, where we compile most of our C++ code to WebAssembly and re-use our core client on the web platform in addition to Android and iOS.

Data Flow

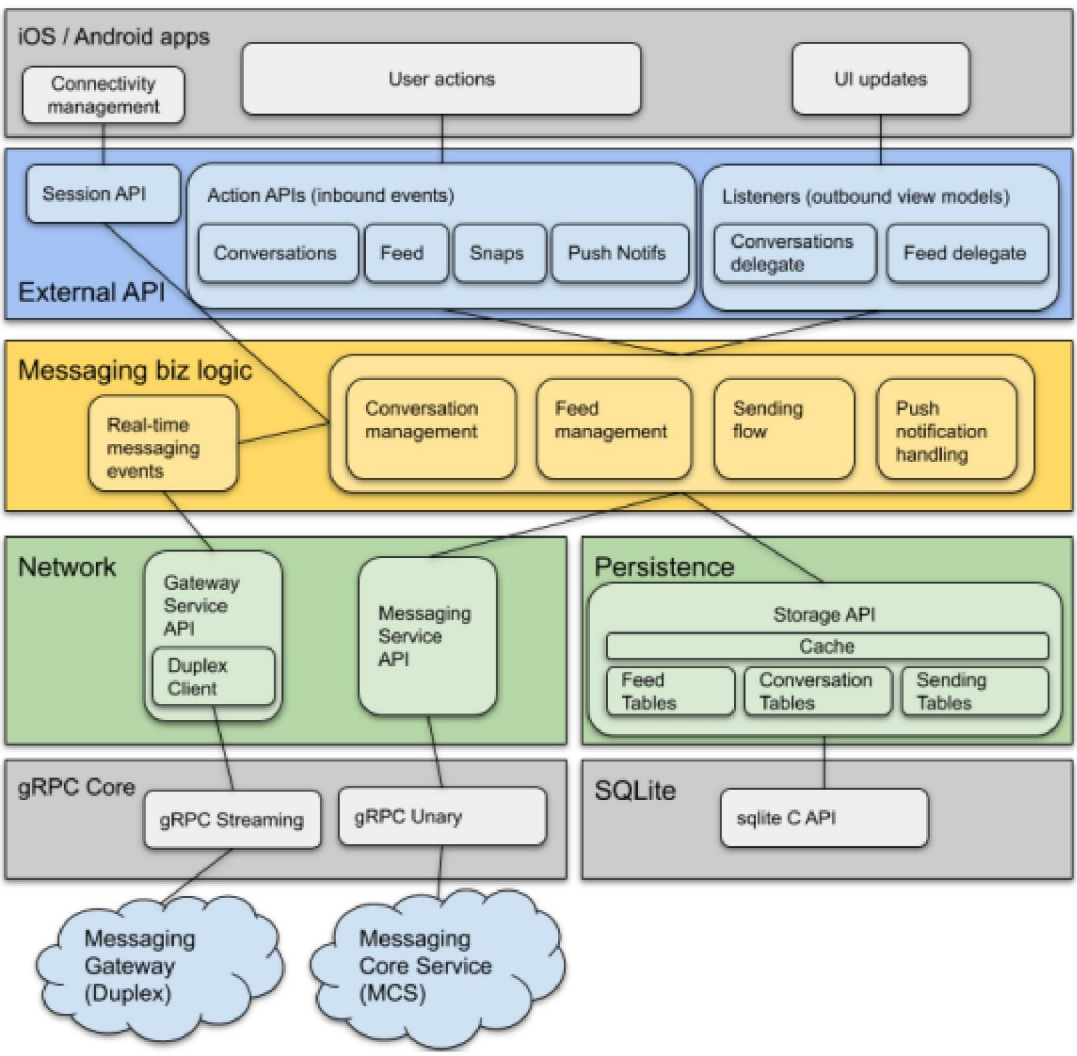
The client follows a one-way data flow. User actions come in, view models go out. User actions can be either the current device’s user interacting with the app, or new message events coming in over the network in response to user interactions (e.g. sending) on the devices of other conversation participants.



Layers of the Client Stack

Generally speaking, the core C++ client is responsible for all business logic, persistence, and networking. Android and iOS layers only deal with user interactions and view model rendering–with some caveats.

When we put it all together with Android and iOS code, the layer diagram looks like this:



- Client Architecture Overview

Conversation Versioning

Each time any part of a conversation changes, the conversation version changes. This fine-grain versioning is a critical step that allows for efficient synchronization.

The basic rules of conversation versioning are:

- Conversations start at version zero.
- When creating a new message, its message ID is assigned to the current conversation version and the conversation version increments.
- When mutating a message or conversation metadata, the conversation version increments.

These simple rules allow the client to tell the server its current version of the conversation so the server can make assumptions about what data the client already knows about, dramatically reducing the amount of information that the client needs to stay up to date. This keeps the amount of data transferred over wire to a minimum.

- Deep Dive on Conversation Versioning

Synchronizing Conversations

Overview

Conversations stay up-to-date in several ways.

1. The client performs a DeltaSync operation, pulling a delta of conversation data either explicitly (e.g. pull to refresh) or implicitly.
2. A real-time message delivery or message update arrives while the app is active and connected (via Duplex).
3. A push notification arrives while the app is active, triggering an implicit sync.

If the app is active and connected via Duplex, roughly 80% of messages arrive in real-time without requiring a DeltaSync. This provides a real-time, interactive messaging experience to users without waiting for server round trips.

Delivering messages with both pull and push models is intentional and key to keeping conversations up-to-date quickly. For more details on this pattern, see Triple Sync Pattern.

DeltaSync

DeltaSync is key for the client to stay up-to-date. Even if real-time delivery systems are down, the client can always periodically DeltaSync to retrieve new conversation data.

The client always keeps track of the version for each conversation. If it thinks conversation XYZ is out-of-date, it issues a call to *DeltaSync(XYZ, currentVersion(XYZ))*. If the current version is 5, this effectively tells the server, “give me all the new data since version 5 for conversation XYZ”. The server will return any new conversation, or nothing if the conversation is already up-to-date on the client.

- Deep Dive on Conversation Synchronization

Real-Time Delivery

When message data arrives in real-time via Duplex, the client has to determine whether they can be committed to the local conversation state. Versioning allows for this with one simple algorithm. For conversation XYZ:

- If the version of an incoming message is currentVersion(XYZ) + 1, then the message is committed to the local conversation state.
- Otherwise, there is a gap in the conversation, and we trigger a DeltaSync for conversation XYZ.

This simple rule allows us to support a high rate of real-time delivery (when connected) and strong consistency with DeltaSync as a fallback.

- Deep Dive on Conversation Synchronization
- Conversation Gap Detection

Send Flow

The Send Flow is a client workflow that triggers on every conversation mutation. It allows us to make updates to a conversation in a durable and consistent way. It is used for message sending, message updates (save, read, etc), and conversation updates (watermark, clearing conversation, etc). The flow was built in a way for it to be easily extended for new features or flows.

The Send Flow is also responsible for things like tracking retries, failing fast if we know there is no network connection, and triggering media uploads for Snaps and other media message types.

- Deep Dive on Send Flow

Media Management

The client coordinates with external systems to upload and download media for messages that include images and video.

Uploading

Uploading media is a multi-step process triggered from the Send Flow. The core C++ client delegates this process out to a platform-specific implementation of an UploadDelegate. The process is roughly the following:

1. Transcode media
2. Encrypt media
3. Get a Bolt UploadLocation
4. Upload media
5. Claim media

Bolt provides us with a ContentReference that we use to reference the media during upload, claiming, and download. We package this ContentReference inside of the message being sent so that recipients can use the same ContentReference to download the media.

Claiming is the process of telling Bolt how we expect to use this media, when it should expire, and how it should be replicated in CDNs. Messaging claims happen in MCS, whenever a user successfully sends a media message, setting the media up for consumption on the recipient side.

- Bolt Concepts

Downloading

The other side of the upload coin is downloading media. ContentManager is a client library that coordinates with Bolt to claim and download media. We use the ContentReference described in Uploading to initiate a download with ContentManager. Whenever a message arrives on a recipient's device, we might opt to preload media so it is readily available if/when the recipient views corresponding messages. Otherwise, the media is immediately downloaded when the user views the message.

- Bolt Concepts
- ContentManager

Message Type Plugins

Many teams create new message types in the Conversation view to support other Snapchat features. This pattern is common enough that we created Message Type plugins, an easy way to make this process as self-service as possible. The idea is that Messaging is–for the most part–just a transport and a framework to work within. Partner teams own their own message type definition, the content format, and the rendering code for the message cell.

- Android Plugin Guide
- iOS Plugin Guide (soon to be deprecated)

Notifications

APIs to process incoming Push Notifications are platform-specific, so each mobile platform handles these differently and processing notifications is not generally part of the C++ client.

On iOS, we must run the notifications processor as an extension process, as Apple no longer allows us to wake up the main app with notifications. The Notification Extension compiles as a separate binary, runs as a separate process, and communicates with the main app using a shared file that the main app reads when the app comes into the foreground. In order to limit the application store size, we compile only a small subset of essential core C++ APIs into the Notification Extension.

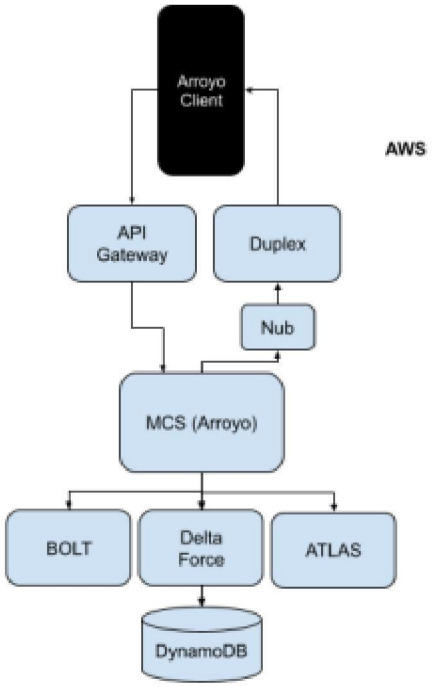
On Android, the story is much simpler. We can wake up the main app when we receive notifications, so the code to process notifications is co-located with the rest of the messaging client and has full access to core C++ APIs.

Currently, we do not include message content in notification payloads and only send a limited amount of message metadata, such as conversation ID, current conversation version, and message ID. This is enough information for the core messaging client to determine whether it already has the message corresponding to the notification or whether it should synchronize the corresponding conversation. In the future, we plan to explore even more real-time delivery by including actual message content in notification payloads.

- Notifications Architecture
- Deep Dive on iOS Notification Extension

Backend Architecture

Overview



The Messaging backend is made up of a small collection of services, with the addition of several platform dependencies. This collection of systems handles storage, transport, media management, friend link management, and sending/receiving messages.

Message Storage and Synchronization

In many messaging systems, messages are sent once, read by a bunch of users, and then never touched again. Message history is often immutable. These systems are read-heavy and the only write operation you really need is append (for new messages).

Snapchat is write-heavy. For every message sent, there are several other writes that occur–read receipts and watermarking always occur, messages must always be deleted, and optionally, things like saving and screenshotting can occur. Message entries must be modified in place and re-written often–they are mutable. This makes operating Snapchat more costly and more complex than other messaging systems.

To solve this, we use several techniques.

- We rely on an efficient delta synchronization model built on top of DeltaForce (and DynamoDB). Delta Force tracks changes on a per-row basis and allows us to request the current state of all rows changed since a certain (logical) timestamp.
- We shard data by conversation in DeltaForce. Each combination of users (pairwise 1:1 or arbitrary group) has its own dedicated conversation data that can be sharded independently. This lets us avoid a hot user partition problem, where popular power users who receive lots of messages (and associated write operations) might easily brownout the partition they are on.
- Every byte matters, both on the wire and in storage at rest. We use protobuf and a carefully manicured data schema to keep size down for write operations.

Messaging Core Service (MCS)

The main service for messaging operations is the **Messaging Core Service (MCS)**, which should've just been called Messaging Service. Within this service are three separate service interfaces, which are targeted at different kinds of clients:

- **MessagingCoreService gRPC service** - is used by our mobile and web clients, and provides user-facing APIs for reading and writing to conversations. It is exposed to the Internet via the API Gateway and uses stateless Snap Token user authentication.
- **MessagingInternalService (MIS) gRPC service** - is used by other Snapchat services to interact with messaging data for a limited set of use cases such as Download My Data, sending TeamSnapchats, or generating Friendship Flashback stories.
- **MessagingMigrationService (MMS) gRPC service** - was used for migrating data from the old FSN-based messaging platform and remains in place to power the backwards compatibility layer (BWC) between the old APIs and the new system (this will remain in place until pre-Arroyo clients are deprecated and FSN is shut down).

Alongside the MCS API service is **MCS Worker**, which is a different entry point into the same code, but focused on running asynchronous background jobs, responding to events from queues (such as account deletion), and running scheduled data mappers (such as our message expiration mapper).

MCS' main job is to act as an API for messaging data that is stored in **Delta Force**, our structured data storage system. MCS receives requests to send messages and stores them in Delta Force, sends notifications to all the participants via PCS and Duplex, and then updates the Feed for each of those participants to reflect the latest state of the conversation. Users can then invoke read APIs to load their Feed or to load messages from each individual conversation. All read APIs are paginated and support delta sync, so once the client has an initial version of the data, it can request incremental updates of only the data that has changed since the last sync.

Special care has gone into the design of MCS' API to make sure that operations are efficient. We strove to keep the wire size of requests and responses at a minimum (counting every byte!) and to perform the minimum number of DynamoDB operations per request. **DynamoDB** operations (via Delta Force) are optimized for Read and Write Capacity Units (RCU and WCU), because those are the units of billable work for DynamoDB, and thus not only minimize latency, but minimize cost. The size of the data stored in DynamoDB is also carefully optimized for cost reasons. For each API, but especially the message sending API, we are cautious in minimizing the number of service calls in order to improve latency and reliability, ideally only ever making a single Delta Force request before considering the request successful.

Requests are synchronous from the client's point of view - if a send request returns OK, the send was successful and the message was committed. However, MCS runs some work asynchronously, after returning a response to the client, if that work is not critical - this includes updating the feed or sending notifications. We leverage the Triple Sync pattern to recover from missed notifications. MCS' business logic largely revolves around permissions checking and validation, making sure that all operations are consistent with our message lifecycle design. We are very careful with MCS code, ensuring that we do not write anything that depends on the contents of a message, only the envelope - this separation prepares us for eventual end-to-end encryption of all messages.

In general, we do not favor batch APIs, with two exceptions. We built a batch conversation fetch API (which can sync multiple conversations at once) to work around per-request overhead imposed by our client networking system. The sending API is fundamentally a batch operation, where a single message can be sent to multiple destinations (conversations or Stories), which mirrors the product experience of selecting multiple

recipients in the app's Send To flow. While all sends to conversations are handled within MCS and stored in Messaging's Delta Force tables, Story destinations are simply forwarded on to the Story Posting Service, operated by the Discover team.

MCS depends on several other services beyond Delta Force. **Atlas** is Snap's API for user and friend data, and MCS reads data from Atlas for several purposes. Most importantly, we load user and friend information to determine messaging permissions - is user A able to talk to user B? This is a complicated policy which takes into account the relationship between the two users and their individual privacy settings, and in some cases their messaging history. This permissions model is a space where messaging will evolve in the future as we launch products like Non-Friend Messaging. We also load user information such as display name and language in order to fill in the contents of push notifications. In general, we try to minimize the critical path dependency on Atlas due to latency concerns and historical instability.

BOLT is Snap's media and CDN management platform, and it provides a uniform API over multiple storage systems (S3/CloudFront and GCS), as well as a DynamoDB-backed reference counting and an object lifetime management system. BOLT was designed with Messaging's requirements in mind and handles many of our product requirements, such as deleting expired media or media after the last conversation references it (e.g. if a Snap is sent to many conversations, there's only one copy of the media, and it isn't deleted until the last conversation releases it). However, MCS is the authority on message (and thus media) lifecycle, and so we explicitly manage that lifecycle in BOLT by claiming and un-claiming media. Claiming media for different message types is a generic operation that is based on information in the message envelope, and is handled as part of the process of committing a new message. It does not follow a full two-phase commit, but the order of writes to BOLT and Delta Force are set up such that if either fails, we preserve privacy without ever losing media.

Push Notification Service (PNS) is run by the Notifications team (which is a part of Messaging) and provides a uniform API for sending push notifications to devices, regardless of their type (iOS, Android, or Web). PNS manages device push registrations and allows consumers to send a push notification to a user rather than a device. PNS is not responsible for the contents of notifications, so MCS crafts these contents, including both their display properties and any metadata that is used by the client to sync conversations in response to notifications.

Duplex (and Nub) are another type of notification service that are owned by the Messaging team. Duplex is a generic push notification system that makes use of a persistent connection from the Snapchat app whenever the app is open. This allows us to send richer notification payloads, with less latency and higher reliability vs. OS-level push notifications, but only when the app is open. This is the third leg of the Triple Sync strategy. Nub is Duplex's internal-facing API that allows services to send a payload to a user without having to know which Duplex hosts the user is connected to.

Besides these main services, there are some adjacent or peripheral services:

- Fidelius is the service that manages E2EE keys for user devices. This has a big impact on messaging, but the client interacts with Fidelius, MCS never talks to it. It is owned by the Privacy Engineering team.
- URL Preview Service is an independent service that provides link previews for URLs shared in chat. It requests web sites on the user's behalf to preserve their privacy (hide their IP) and parse out a title, icon, and more to render a preview. This service is called directly from the client. It is owned by the Messaging team.
- Best Friends Service (BFS) is notified on certain messaging events (such as sending a message to a friend) and maintains a sliding window history of their events in order to calculate a user's Best Friends -

the small list of users they interact with most. It is owned by the Messaging team. Here's the BFS Design Diagram.
- SnapPro and Story Reply services are owned by the Snap Pro team and are used to implement "public profiles" for Stories. MCS consumes information from them for its permissions model (public profiles can be sent certain messages without being friends) and to notify about Story Replies, a type of message sent to a public profile in reply to their Story. The Story Reply service uses this information to show Story replies in the Story management UI.
- PhotoDNA is a service owned by the Privacy team which scans all camera roll media shared in chat ("chat media") for CSAM imagery.
- Group Invite service is possibly still in FSN, owned by the Send To team, and responsible for generating invite links to Groups - however, there's discussions about discontinuing this feature.

Duplex

Duplex provides the ability to send and receive arbitrary, user-addressed messages between Snapchat clients and any backend systems. It is entirely decoupled from Messaging and acts as a general purpose, real-time streaming transport. It doesn't care what is being sent or which backend systems it acts as a transport for.

Duplex is simple and only provides the following operations:
- Send a message from a client to a backend system.
- Receive a message on a client from a backend system.
- Send a message from a backend system to (all of) a user's currently connected clients.
- Receive a message from a user's client on a backend system.

Duplex also has some constraints:
- It does not track any states, except for the currently connected set of users.
- It offers at-most-once delivery and does not deliver messages to offline users. Messages sent to offline users are dropped and forgotten.
- It offers best-effort delivery and does not buffer or queue messages for later.

Duplex effectively acts as a message router for real-time communication between Snapchat connected users and backend systems. It abstracts away all the complexity and heavy lifting associated with keeping open millions of client connections that are addressable by user ID and capable of sending/receiving data.

Duplex+Nub

The term "Duplex" is typically used to refer to the single communication system described above. In reality, it is actually two different systems working together: Duplex (proper) and Nub.

Duplex handles the maintenance of a giant set of open connections and defines a bidirectional gRPC API for communicating over those connections. It also defines a cross-platform API for sending and receiving messages on the client.

Nub handles the maintenance of a giant distributed hash table [of user ID → connection handle] and defines a gRPC API for sending messages from backend services to connected users.

- Duplex Service API
- Duplex Client API
- Nub Service API

Use in Messaging

Messaging uses Duplex for real-time delivery of messages from MCS to connected clients; this was the primary reason for creating Duplex. It is a one-way communication, where MCS broadcasts each message sent to all participants in a conversation. If participants are not online, messages are dropped.

Duplex is also used to send and receive user presence heartbeats within the Presence Core Service (PCS). This is a two-way communication. When a user is present in the conversation view, we send presence heartbeats for the current user to PCS and receive presence heartbeats for other (present) users from PCS. Heartbeats are used on the recipient client to render presence indicators at the bottom of the conversation view.

Push Notifications

To send a push notification, we use the Push Notification Service (PNS). It in turn connects with the Push Notification Data Registry (PNDR) to look up the routing information for the devices registered to this user which, in the single-login world[1], is essentially a combination of push token, device type (iOS | Android | Web) and build type.

- Notifications Architecture

Regionalization

TODO S3

[1] Multi-login will include a session ID to link tokens to.

Page 16 Comments

S3 @s[redacted]

[redacted] 3/7/2022 12:41 PM

Appendix

Resources

Messaging for dWeb
Arroyo Q&A Takeover - MM
Client Documentation Repository
Message Retention Overview
Notifications Architecture