Message Brokers: History, Popularity and High-level overview

Background:

A message broker is an intermediary software program that translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver.

Protocols: Like AMQP (Advanced Message Queuing Protocol), MQTT (Message Queuing Telemetry Transport), and STOMP (Simple/Streaming Text Oriented Messaging Protocol). We'll relate this in the history section of this document.

Analogy:
Think of it as a digital post office for software applications. Where the message broker acts as the middle man to deliver messages from producers to consumers.

Before continuing further into this doc, we have to know about some message broker jargons:

1. Producers: Applications that create and send messages to the message broker. They are the originators of the information.
2. Consumers: Applications or services (in a microservice architecture) that receive and process messages from the message broker. They are the destination for the information
3. Queue/Topic: A queue is a line of messages waiting to be processed, typically on a one-to-one delivery model (one message to one consumer). A topic, on the other hand, is used for a one-to-many delivery model, where a message is published once and delivered to multiple subscribers (consumers).
4. Message: The actual data being sent from the producer to the consumer. A message usually has two parts: a header (with metadata like message ID, timestamp, etc.) and a body (the payload).

It is highly recommended to have a look at this blog by the RedMonk's analyst Kate Holterhoff: [Why Message Queues Endure: A History]

Kate's narrative, analogy and conversations with prominent figures in message space provides a more clear picture of why message brokers have evolved to what they are today, with their implication in history.

More on Message Queuing:

The early usage of queuing and brokers is rooted in telecommunication.

The very first account I find is Agner Krarup Erlang, a Danish engineer working for the Copenhagen Telephone Company in the early 1900s, is considered the father of queuing theory. He developed mathematical models to determine the optimal number of telephone lines and operators needed to handle a certain volume of calls without excessive waiting times for customers. His work was so influential that the unit of measurement for telecommunication traffic is named the "erlang" in his honor.

He was a prominent statistician that had following mathematical contributions that lead to the queuing algorithms and methods that we use today. Wiki

- 1909 – "The Theory of Probabilities and Telephone Conversations", which proves that the Poisson distribution applies to random telephone traffic.[7][8][9]
- 1917 – "Solution of some Problems in the Theory of Probabilities of Significance in Automatic Telephone Exchanges", which contains his classic formulae for call loss and waiting time.[10][11]
- 1920 – "Telephone waiting times", which is Erlang's principal work on waiting times, assuming constant holding times.[12][13]

The primary drivers for the advancement of this technology were the banking and airline industries. These sectors had a critical need for guaranteed message delivery to ensure that financial and reservation transactions were processed reliably without data loss, duplication, or corruption. This led to the creation of enduring systems like IBM's Transaction Processing Facility (TPF), which is still used by major airlines today.

A quote worthy conversation with Clemens Vasters

“So I think the key thing to focus on is asynchronous messaging. … Obviously people have been wiring computers together and wiring applications together for a long time. But it was always in a synchronous way. So in other words, one application had to be ready to receive the message before it could start sending it. They had to receive the whole message and acknowledge it before either could carry on. And we used to compare that to the wings … of a jumbo jet when it was going down the runway to take off, the wings actually flap slightly … because if they were rigid they’d snap off, so they have to have that flex in them, and by putting a queue or a springy connection in between two applications you stop them being rigid and then one of them snapping off, and the whole thing breaking, you allow them a little bit of flex.”

Throughout the whole history of message queuing, its core usage were found to be the following:

1. Asynchronous Communication: Message queues allow different parts of a system to communicate without needing to be available at the same time. This creates a more flexible and resilient system, as a delay or failure in one part won't immediately bring down the entire system.

2. Decoupling Applications: By acting as an intermediary, a message queue "decouples" applications. This means one application can send a message and move on to other tasks without waiting for the recipient to process it. The receiving application can then retrieve the message when it's ready.

3. Enhanced System Resilience: The combination of asynchronous communication and decoupling makes the overall system more robust. It prevents a "domino effect" where the failure of one component causes a cascade of failures throughout the system.

4. Guaranteed Delivery: Many message queue systems offer "assured once and once only delivery." This is a critical feature for many applications, as it ensures that a message, once sent, will be delivered to its destination without duplication or loss.

5. Managing Workloads: Message queues can effectively buffer and manage fluctuating workloads. During periods of high traffic, messages can be held in the queue and processed as resources become available, preventing the system from being overwhelmed.

6. Separating Complex Processes: They enable the separation of complex and time-consuming tasks. These tasks can be offloaded to dedicated services that process them at their own pace, improving the overall performance and responsiveness of the main application.

History:

Early Development (Internal Applications):

~ 1965: IBM announces QTAM (Queued Telecommunications Access Method) for communication in a queue.

- This was partially funded by the NASA Apollo program. Somewhere I found that it was used for interstellar communication.

~ 1970: IBM releases Transactional Processing Facility (TPF) for guaranteed message delivery and to account for lost & duplicate records.

Commercial middleware:

1993: IBM MQSeries
Designed to help systems move away from synchronous applications.
It featured "assured once and once only" message delivery.

1997: TIBCO Rendezvous (TR)
Widely used on Wall Street trading floors.
It could broadcast data to many listeners simultaneously.

Late 1990s: Financial data standards like SWIFT and FIX (Financial Information Exchange) were established.

Open-Source Era:

2003: Advanced Message Queuing Protocol (AMQP)
An open-source protocol developed to remove the dependency between the protocol and the message broker.

2007: RabbitMQ
An open-source software that was the first to fully implement the AMQP standard.
It became popular quickly due to being developer-friendly and easy to use.

Kafka & Data Streaming (Present Date):

2011: Apache Kafka

It was not designed as a traditional queue but as a "distributed commit log" to handle immense data throughput.

2014: Kubernetes & Microservices

With the rise of microservices, message brokers became more popular and are currently a key tool for decoupling components in modern system architecture

Packet level Components:

Before we begin a high-level overview of message brokers, let's clear out the components, especially what is inside the header and body of each packet that is involved in flow of information between broker, producer and consumer.

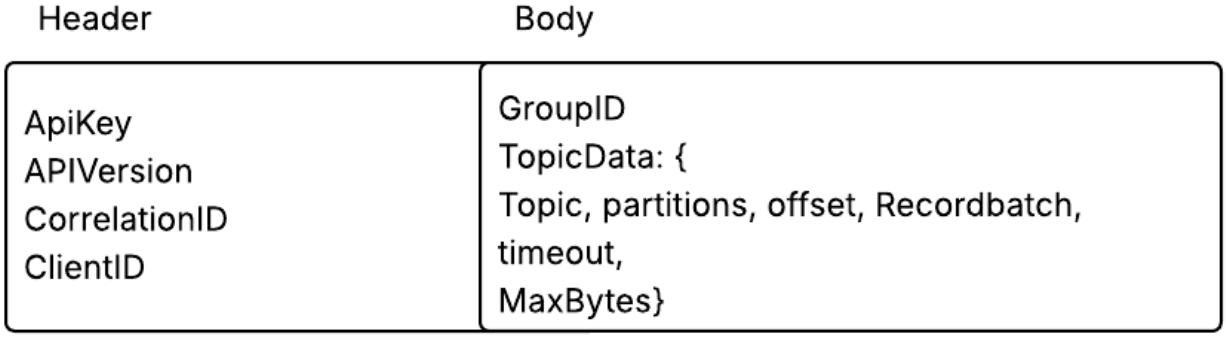

Fig: A typical packet in the message broker information flow.

As we go through the high-level flow of information, we will come across the fact that different processes (example producer to broker, broker to producer, broker to consumer and vice versa) packets are slightly different. In the above figure too, I have added MaxBytes that is used during broker to client communication but not others.

In the next page, we look at the packet contents in correspondence to their flow.

Flow 1 : *ProduceRequest* packet (from producer to broker)

Used by a producer application to send a batch of records to a Kafka broker. This is the fundamental "write" operation.

Packet header (Common to all requests)

Variable	Type	Example Value	Description
ApiKey	int16	0	A numeric code identifying the type of request. 0 always means ProduceRequest.
ApiVersion	int16	8	Specifies the version of the ProduceRequest schema being used. Allows for backward-compatible protocol changes.
CorrelationID	int32	57	A unique number generated by the client for each request. The broker's response will contain the same ID, allowing the client to match responses to their original requests asynchronously.
ClientID	string	"user-service-prod-1"	A user-defined string to identify the client application that sent the request. Invaluable for logging and debugging on the broker.
RequestSize	int32	1532	The total size of the request in bytes.

Packet Body:

Variable	Type	Example Value	Description
Acks	int16	-1	The durability guarantee. Defines how many brokers must acknowledge the write before it is considered successful. • 0: Fire-and-forget. Producer does not wait for any acknowledgment. • 1: Leader-only. Producer waits for the leader broker to write the record to its log. • -1 (or all): Full durability. Producer waits for the leader and all in-sync follower brokers to acknowledge the write.
Timeout	int32	30000	The time in milliseconds the producer will wait for a response from the broker. If this timeout is reached, the client will consider the request failed.
TopicData	array		An array containing the records to be sent, grouped by topic.
↳ Topic	string	"user-created"	The name of the topic to which records are being sent.
↳ PartitionData	array		An array containing the records for each partition within that topic.
↳ Partition	int32	2	The index of the target partition.
↳ RecordBatch (Compressed using supported packages)	bytes	(binary data)	The core payload. This is a compressed batch of one or more records. It contains the message keys, values, timestamps, and headers.

Flow 2 : *FetchRequest* Packet (from broker to producer)

Used by a consumer to retrieve records from a Kafka broker. This is the fundamental "read" operation.

Packet Header:
The packet header is almost similar to the produce request; the only difference is that the *CorrelationID* has different values. If you have read the above header table, the *CorrelationID* is used as an identifier for the reply-response communication. In essence, this is meant for the correct reply to the asked question, *CorrelationID* is like a tracking number.

Packet Body:
**New thing to look for here is* FetchOffset, we'll use this in the high level flow as well*

Variable	Type	Example Value	Description
ReplicaID	int32	-1	The ID of the replica sending the request. For consumers, this is always -1.
MaxWait	int32	500	The maximum time (in ms) the broker will wait for new data to arrive if there are no messages available at the requested offset. This enables efficient long-polling.
MinBytes	int32	1	The minimum amount of data (in bytes) the consumer wants to receive. The broker will wait until this much data is available or MaxWait time is exceeded.
MaxBytes	int32	52428800	The maximum total size of data the response packet can contain. This prevents the broker from sending a message that is too large for the consumer to handle.
Topics	array		An array specifying which topics and partitions to fetch from.
↳ Topic	string	"user-created"	The name of the topic to fetch from.
↳ Partitions	array		The specific partitions to fetch within that topic.
↳ Partition	int32	2	The index of the partition to read from.
↳ FetchOffset	int64	46	The starting offset to begin reading from. This is the consumer's "bookmark" and is the most critical field for ensuring no data is missed.

Flow 3: OffsetCommitRequest Packet (From Client to Broker)

Used by a consumer to save its progress (the last successfully processed offset) for a given partition.

Packet Header:
The packet header is almost similar to the produce request; the only difference is that the *CorrelationID* has different values. If you have read the above header table, the *CorrelationID* is used as an identifier for the reply-response communication. In essence, this is meant for the correct reply to the asked question, *CorrelationID* is like a tracking number.

Packet Body:

Variable	Type	Example Value	Description
GroupID	string	"email-service-group"	The unique identifier for the consumer group. The broker stores offsets on a per-group basis.
GenerationID	int32	5	A number managed by the broker to identify a specific "generation" of a consumer group. This prevents "zombie" consumers from committing offsets after a group rebalance.
MemberID	string	"consumer-1-..."	A unique ID assigned by the broker to this specific consumer instance within the group.
Topics	array		An array specifying the offsets to be committed for each topic and partition.
↳ Topic	string	"user-created"	The topic for which the offset is being committed.
↳ Partitions	array		The specific partitions to commit.
↳ Partition	int32	2	The index of the partition.
↳ Offset	int64	47	The offset to be committed. Crucially, this is the offset of the next message the consumer expects to read (last processed offset + 1).

High level packet flow (Producer - Broker - Consumer):

Now that we have the packet parameters checked out and understood, we will move on to the next portion for visualizing. Don't worry if the above table is not clear, we'll use every variable in our flow to properly visualize the message broker system. We'll explore in parts, here's our [whiteboard] for reference.

1. Producer (Corresponds to flow 1):

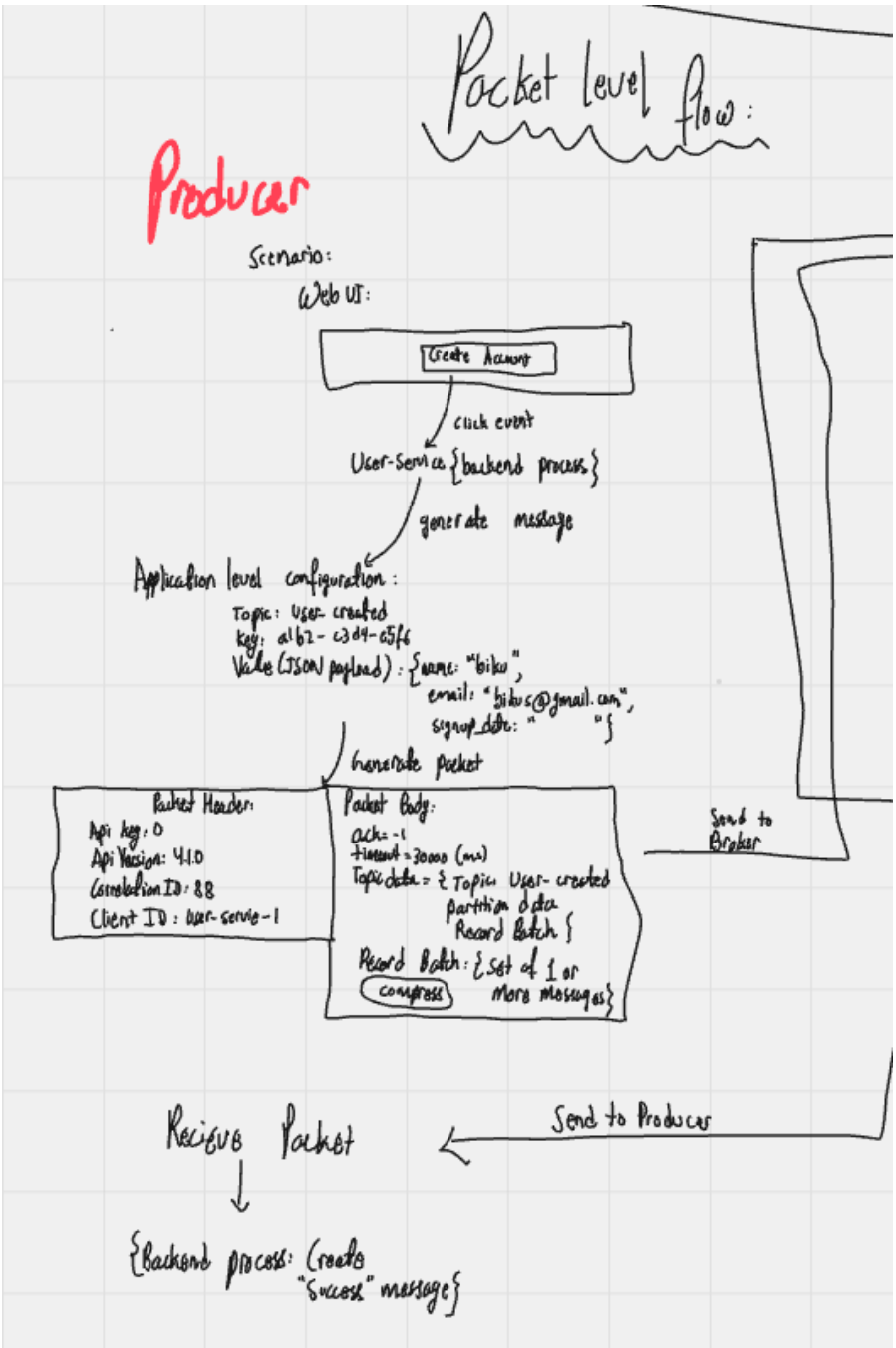

Figure: Packet generation flow, Producer.

Scenario: Our user application has the purpose of creating new users for our microservice application. We have different microservices set up such as Create User, Logs and Analytics that consume the messages and do their respective processings.

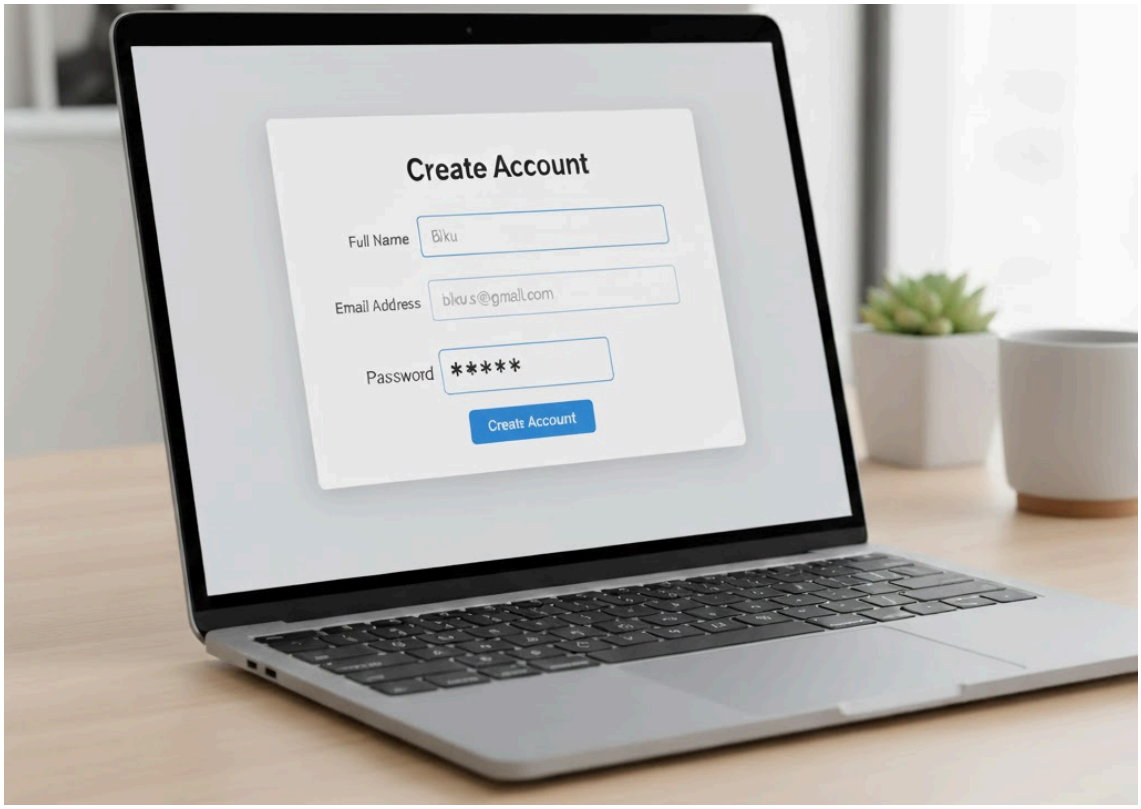

Fig: Simple Web UI for our use case.

The producer process is summarized in the points below:

1. User Action (Trigger):
   A user clicks "Create Account" on a web interface. This action triggers a backend process within the User-service.

2. Message Generation (Application Level):
   The User-service generates the core content of the message. This includes:

   Topic: user-created (The channel the message will be sent to).

   Key: A unique identifier (a1b2-c3d4-e5f6).

   Value (Payload): The actual data in JSON format, containing the new user's details like name and email.

3. Packet Assembly (Protocol Level):
   The application message is wrapped into a formal packet for transmission. This packet consists of:

   - Packet Header: Contains metadata for the broker, such as the API Key, API Version, Correlation ID (to track requests), and Client ID (user-service-1).

   - Packet Body: Contains the topic data, acknowledgment settings (Acks: -1), a timeout, and the actual message payload, which can be batched with other messages and compressed for efficiency.

4. Transmission and Acknowledgment:
   The Producer sends this fully assembled packet to the message Broker.

   The Broker processes the packet and sends an acknowledgment packet back to the Producer.

   The Producer's backend process receives this acknowledgment, confirming the message was successfully delivered to the broker, and can then create a "success" message for its internal logs or processes.

2. Broker:

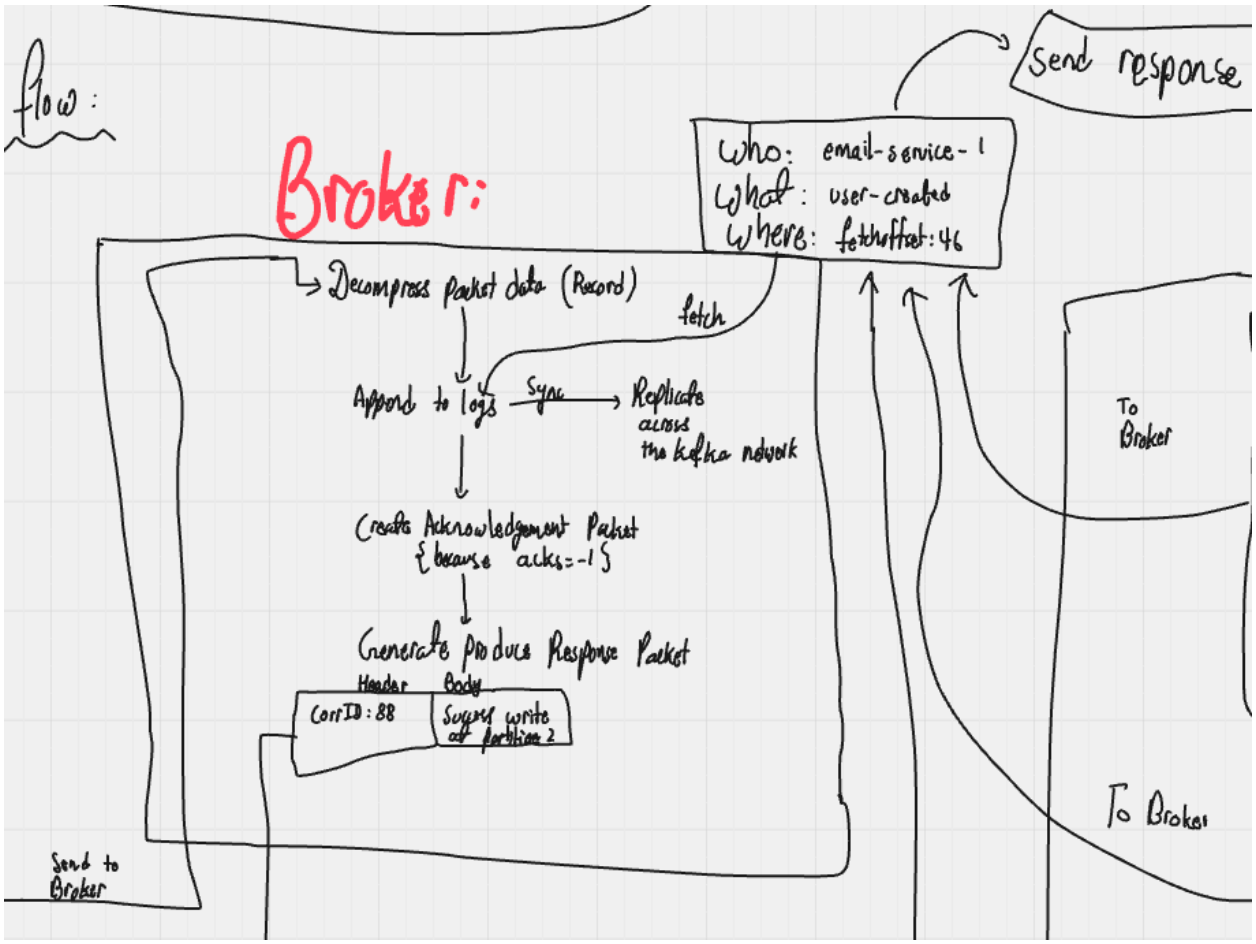

Fig: Broker log storing, response packet generation.

When the broker receives a packet from a producer, it performs the following critical actions:

1. Decompress & Validate: It first decompresses the packet data to access the message record(s). Especially look into the Record batch into topics

2. Append to Log: The broker appends the message to the appropriate log file based on its topic (user-created) and partition. For durability and fault tolerance, this data is also replicated across other brokers in the cluster.

3. Send Acknowledgment: It creates and sends an acknowledgment packet back to the producer. This packet uses the *CorrelationID* from the original request to confirm that the message has been successfully received and stored.

There are some additional processes shown in the figure, I'll connect that in the next section.

3. Consumer:

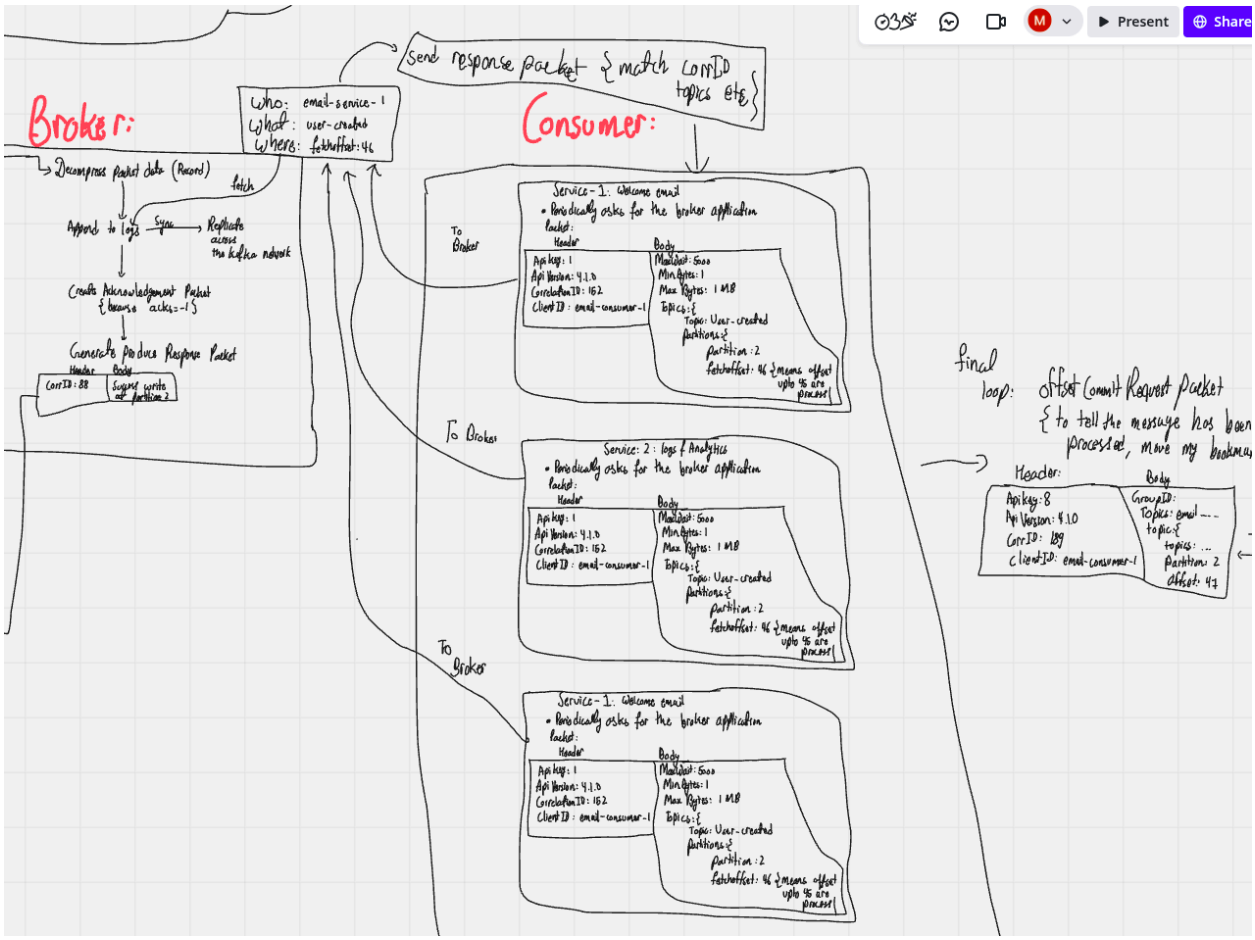

Fig: Services fetching messages, broker communication.

Independent consumer services, such as Service-1 (for sending welcome emails) and Service-2 (for analytics), operate in a pull-based model:

1. Periodic Requests: Each consumer service periodically sends a Fetch Request to the broker to ask for new messages.

2. Specifying the "Bookmark": This request is highly specific. The consumer tells the broker exactly which Topic and Partition it's interested in. Most importantly, it includes a *FetchOffset*. The offset acts like a bookmark, telling the broker the exact position of the last message the consumer successfully processed. This ensures the consumer doesn't get duplicate messages and can resume from where it left off if it restarts.

And the process (who, what, where) on the broker section,

3. Broker: Sending Data to Consumers
   Responding to Fetch Requests: Upon receiving a fetch request, the broker finds the requested topic and partition. It reads the messages starting from the FetchOffset provided by the consumer and sends them back in a Response Packet.

   And finally for system reliability,

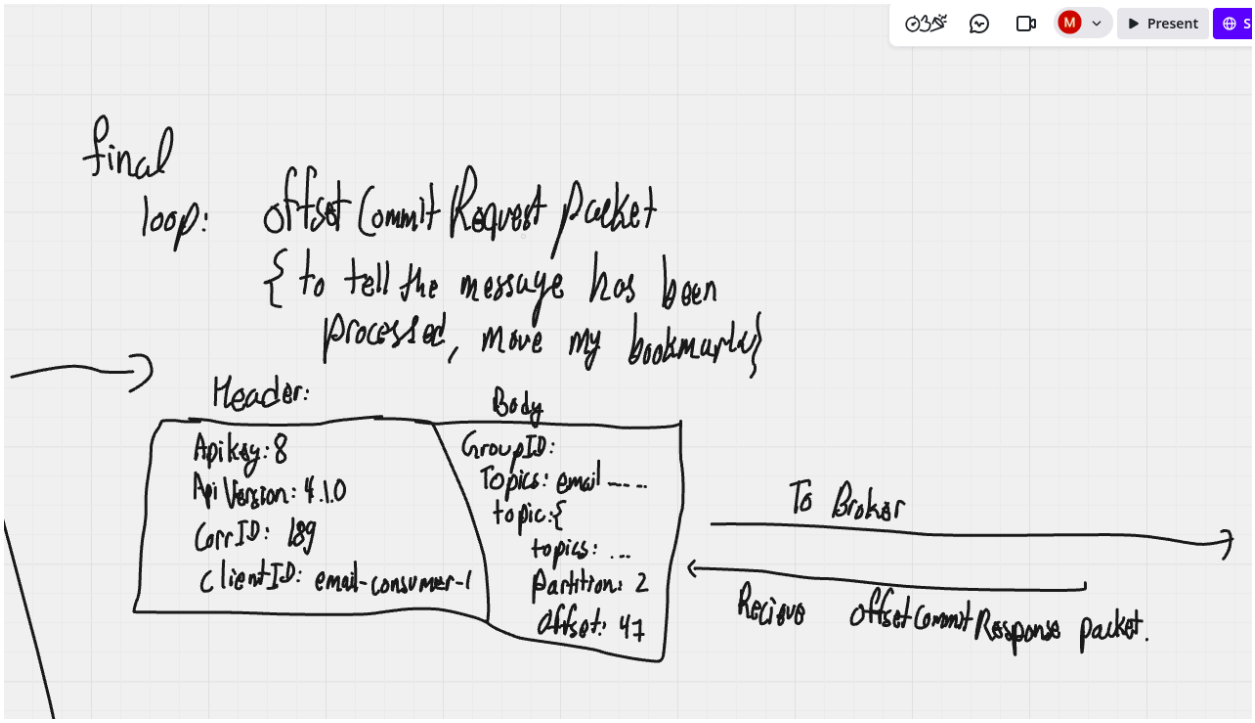

Fig: OffsetCommitRequest packet

1. Processing the Message: Once a consumer (e.g., Service-1) receives and successfully processes a message (e.g., sends the welcome email), it needs to update its "bookmark."

2. Sending an Offset Commit Request: The consumer sends an Offset Commit Request Packet back to the broker. This packet essentially tells the broker, "I have successfully processed all messages up to offset #42. My new bookmark is now at offset #43."

3. Updating the Bookmark: The broker records this new offset for that specific consumer group. The next time this consumer asks for data, the broker will know to start sending messages from offset #43 onwards, ensuring data is processed sequentially and reliably.

After this, the services will handle their respective tasks.