Chapter 23

Messaging

1. Introduction to Messaging

- **Scenario:** Consider an endpoint for users to upload videos, which then need to be encoded into different formats/resolutions (a potentially long-running task).
- **Problem:** If the API gateway directly calls an encoding service and waits, it can be slow. A naive "fire-and-forget" approach risks losing the request if the encoding service instance crashes.
- **Solution:** Introduce a **message channel** (or message broker) between the API gateway (producer) and the encoding service (consumer).
- **Definition:** Messaging is a form of *indirect communication*.
  - A **producer** writes a message to a channel/message broker.
  - The broker delivers the message to a **consumer** on the other end.
  - The message channel acts as a *temporary buffer* for the receiver.
- **Asynchronous Nature:** Unlike direct request-response, messaging is *inherently asynchronous*, as sending a message doesn't require the receiving service to be online at that exact moment.
- **Message Structure:**
  - **Header:** Contains metadata, such as a unique message ID.
  - **Body:** Contains the actual content of the message.
- **Message Types:**
  - **Command:** Specifies an operation to be invoked by the receiver.
  - **Event:** Signals to the receiver that something of interest has happened

to the sender.

- **Service Adapters:** Services use adapters to interact with message channels.
  - *Inbound Adapters:* Receive messages from channels.
  - *Outbound Adapters:* Send messages to channels. *(As illustrated in Figure 23.1)*

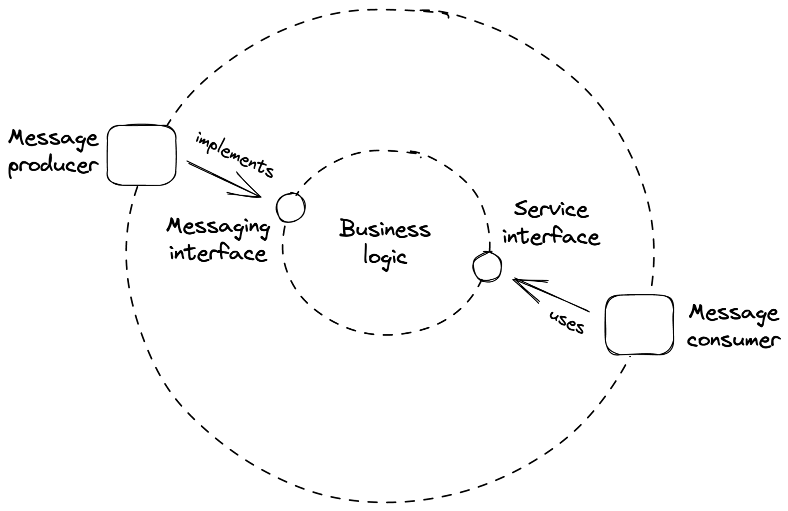

Figure 1: The message consumer (an inbound adapter) is part of the API surface of the service.

- **Example Workflow (Video Encoding):**
  1. The API gateway uploads the video to a file store (e.g., S3).
  2. It then writes a message to the channel, including a link to the uploaded file.
  3. The API gateway responds to the client with a `202 Accepted` status, signaling that the request has been accepted for processing but hasn't completed yet.
  4. Eventually, an instance of the encoding service will read the message from the channel and process the video.
  5. Crucially, the request (message) is *deleted from the channel only when it's successfully processed.* This ensures that if the encoding service fails mid-process, the message will eventually be picked up again and retried

by another instance or the same instance after recovery.

2. Benefits of Decoupling with a Message Channel

- **Availability:** The producer can send requests to the consumer even if the consumer is *temporarily unavailable* or offline.
- **Scalability:** Requests can be *load-balanced* across a pool of consumer instances, making it easy to scale out the consuming side.
- **Load Smoothing:** Because the consumer can read from the channel *at its own pace*, the channel helps to smooth out load spikes, preventing the consumer from getting overloaded.
- **Batch Processing:** Enables processing multiple messages within a single batch or unit of work.
  - Most messaging brokers support this by allowing clients to fetch *up to N messages* with a single read request.
  - **Trade-off:** Batching degrades the processing latency of *individual* messages.
  - **Benefit:** It *dramatically improves the application's overall throughput.* When the application can afford the extra latency for individual messages, batching is highly beneficial.

3. Channel Types (Based on Message Delivery)

- A message channel generally allows any number of producer and consumer instances to write to and read from it.
- Channels are classified based on how they deliver messages:
  - **Point-to-Point Channel:** The message is delivered to *exactly one* consumer instance from the pool of available consumers.

  - **Publish-Subscribe (Pub/Sub) Channel:** *Each consumer instance* subscribed to the channel receives a *copy* of every message.

4. Complexities Introduced by Messaging

- **Operational Overhead:** The message broker is *yet another service* that needs to be maintained, monitored, and operated.
- **Increased Latency:** There is an *additional hop* between the producer and consumer, which necessarily increases communication latency. This latency can be further exacerbated if the channel has a *large backlog* of messages waiting to be processed.
- **Trade-offs:** As always in system design, introducing messaging involves trade-offs.

5. Common Messaging Communication Styles

- **One-Way Messaging:** *(As illustrated in Figure 23.2)*
  - The producer writes a message to a *point-to-point channel* with the expectation that a consumer will eventually read and process it.
  - This is the style used in the initial video encoding example.

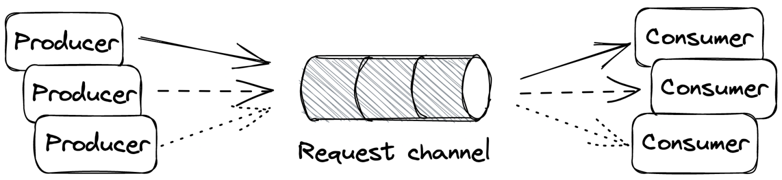

Figure 2: One-way messaging style

- **Request-Response Messaging:** *(As illustrated in Figure 23.3)*
  - Similar to the direct request-response style but with messages flowing through channels.

    - The consumer has a *point-to-point request channel* from which it reads messages.
    - *Every producer has its own dedicated response channel.*
    - **Flow:**
        1. A producer writes a message to the request channel, decorating it with a *request ID* and a *reference to its own response channel.*
        2. After a consumer reads and processes the message, it writes a reply to the *producer's specified response channel*, tagging the reply with the original request's ID. This allows the producer to identify which request the reply belongs to.

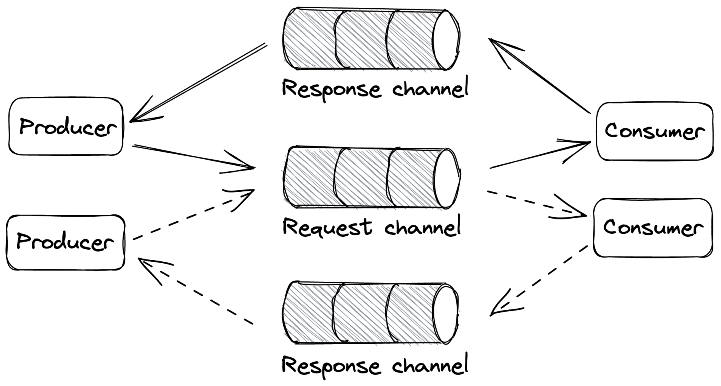

Figure 3: Request-response messaging style

- **Broadcast Messaging:** *(As illustrated in Figure 23.4)*
    - The producer writes a message to a *publish-subscribe channel* to broadcast it to *all consumer instances* subscribed to that channel.
    - This style is generally used to notify a group of processes that a specific *event has occurred* (e.g., as seen in the outbox pattern).

6. Message Broker Guarantees

- A message channel is implemented by a messaging service or **broker** (e.g., AWS SQS, Kafka), which buffers messages and decouples producers from

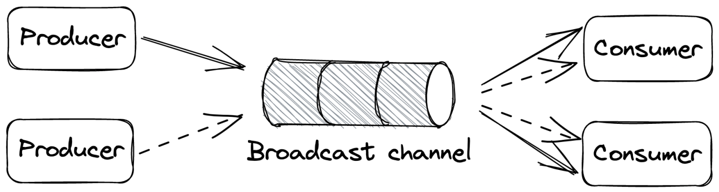

Figure 4: Broadcast messaging style

consumers.

- Different message brokers offer *different guarantees* depending on their implementation trade-offs.
- **Ordering:**
  - You might expect a channel to respect the insertion order of messages, but some implementations (e.g., SQS standard queues) *do not offer strong ordering guarantees.*
  - **Reason:** Brokers are necessarily distributed to scale horizontally. Guaranteeing order across multiple nodes is challenging as it requires coordination.
  - **Partitioned Channels (e.g., Kafka):**
    * Some brokers partition a channel into multiple *sub-channels* (partitions).
    * Messages are routed to sub-channels based on their *partition key.*
    * Since each partition is small enough to be handled by a single broker process, it's trivial to *enforce ordering of messages routed to it.*
    * However, to guarantee end-to-end message order, typically *only a single consumer process* is allowed to read from a given sub-channel.
  - **Caveats of Partitioned Channels:** (Similar to those discussed in Chapter 16 on Partitioning)
    * A partition could become "hot" (high volume of messages), causing

    the consumer reading from it to be unable to keep up.
    * Rebalancing might be needed (adding more partitions), which could degrade broker performance while messages are shuffled.
  - It should be clear why *not guaranteeing the order* of messages makes the implementation of a broker much simpler.

- **Other Guarantees and Trade-offs a Broker Makes:**
  - *Delivery guarantees* (e.g., at-most-once, at-least-once).
  - *Message durability* guarantees.
  - *Latency* characteristics.
  - *Messaging standards* supported (e.g., AMQP).
  - Support for *competing consumer* instances.
  - *Broker limits* (e.g., maximum supported size of messages).
- **Assumptions for Simplicity (in this chapter, similar to SQS/Azure Queue Storage):**
  - Channels are *point-to-point* and support many producer and consumer instances.
  - Messages are delivered to the consumer *at least once.*
  - While a consumer instance is processing a message, the message remains in the channel but is *hidden from other instances* for the duration of a **visibility timeout**.
    * This timeout guarantees that if the consumer instance crashes while processing, the message will *become visible again* to other instances when the timeout triggers.
  - When the consumer instance finishes processing the message successfully, it *deletes the message* from the channel, preventing it from being received by any other consumer instance.

7. Exactly-Once Processing

- **The Challenge:** True “exactly-once message delivery” is practically impossible due to the risk of crashes at various stages.
  - If a consumer instance *deletes the message before processing it*: there’s a risk it could crash after deleting but before processing, causing the message to be *lost for good.*
  - If a consumer instance *deletes the message only after processing it*: there’s a risk it could crash after processing but before deleting, causing the *same message to be read again* later by another instance (or the same one after restart).
- **Achievable Goal: Simulate Exactly-Once Message Processing**
  - The best a consumer can do is aim for *at-least-once delivery* and then handle potential duplicates.
  - This is achieved by:
    1. Requiring messages/operations to be **idempotent** (processing a message multiple times has the same effect as processing it once).
    2. Deleting messages from the channel *only after they have been successfully processed.*

8. Handling Failures

- When a consumer instance fails to process a message, the visibility timeout eventually triggers, and the message is *delivered to another instance* (or retried by the same after recovery).
- **Problem of “Poison Pill” Messages:** What if processing a specific message *consistently fails* with an error?
  - To guard against such a message being picked up and retried repeatedly

  in perpetuity, we need to *limit the maximum number of times* the same message can be read/delivered from the channel.
  - **Retry Counting:**
    - The broker can stamp messages with a counter tracking delivery attempts.
    - If the broker doesn't offer this, the consumer application can implement it.

- **Dead Letter Channel (DLC) / Dead Letter Queue (DLQ):**
  - Once the maximum retry count for a message is reached, we still need to decide what to do.
  - A consumer *shouldn't just delete a message without processing it*, as that would cause data loss.
  - Instead, the consumer can *remove the message from the main channel after writing it to a DLC.*
  - A DLC is a separate channel that buffers messages that have been retried too many times.
  - **Benefits:**
    - Messages that consistently fail are *not lost forever.*
    - They are put aside so they don't *pollute the main channel* and waste consumer resources by being retried endlessly.
    - A human operator can later *inspect these messages* to debug the failure. Once the root cause is identified and fixed, messages can potentially be *moved back to the main channel* to be reprocessed.

9. Handling Backlogs

- **Benefit of Broker:** One of the main advantages of using a message broker is that it makes the system more *robust to outages.* The producer can continue writing messages to a channel even if the consumer is temporarily degraded or unavailable.
- **Backlog Formation:** A backlog builds up when the *arrival rate of messages is consistently higher than their deletion rate* (i.e., the consumer can't keep up with the producer).
- **Bimodal System Behavior:** A messaging channel can introduce a bimodal behavior:
  - **Mode 1 (Healthy):** There is no backlog, and everything works as expected.
  - **Mode 2 (Degraded):** A backlog builds up, and the system enters a degraded state regarding message processing timeliness.
- **Issue with Backlogs:** The *longer a backlog builds up, the more resources and/or time it will take to drain it.*
- **Common Reasons for Backlogs:**
  - More producer instances come online, and/or their throughput increases, and the consumer *can't keep up* with the increased arrival rate.
  - The consumer's *performance has degraded* (e.g., due to a bug, resource contention, or slow downstream dependency), and messages take longer to be processed, decreasing the effective deletion rate.
  - The consumer *fails to process a fraction of the messages* (e.g., due to transient errors or "poisonous" messages). These messages are picked up again until they eventually end up in the dead letter channel. This *wastes consumer resources* and delays the processing of healthy messages.

- **Detecting and Monitoring Backlogs:**
  - Measure the *average time a message waits* in the channel to be read for the first time (often called "age of oldest message" or similar metrics).
  - Brokers typically attach a *timestamp* when a message was first written to it.
  - The consumer can use that timestamp to compute how long the message has been waiting by comparing it to the timestamp taken when the message was read.
  - Although the two timestamps are generated by physical clocks that aren't perfectly synchronized, this measure generally provides a *good warning sign* of developing backlogs.

10. Fault Isolation

- A single producer instance emitting "poisonous" messages (that repeatedly fail to be processed) can *degrade the consumer* and potentially create a backlog. These messages are processed multiple times before they (hopefully) end up in the dead-letter channel.
- **Importance:** It's important to find ways to deal with such poisonous messages *before* they cause widespread degradation.
- **Strategy for Isolation:**
  - If messages are decorated with an *identifier of the source* that generated them (e.g., user ID, tenant ID), the consumer can treat them differently based on this source.
  - **Example:** Suppose messages from a specific user (or tenant) start failing consistently.
    - The consumer could decide to *write these problematic messages to*

an alternate, low-priority channel and remove them from the main channel without extensive processing attempts.

∗ The consumer would still read from this “slow” or “error” channel, but perhaps *less frequently* than the main channel.

∗ This helps to *isolate the damage* a single misbehaving user or data source can inflict on others using the system.