Asynchronous Processing

Arnav Gupta

December 10, 2024

Contents

1 Motivation

Responses acknowledge request received and can contain data or info about failure that client needs to react to.

Synchronous APIs have **blocking** request where the client waits for the server to finish processing, hence the two are synchronized.

If request very long, client may not care to wait until message delivery is verified.

2 Asynchronous Processing

Acknowledge request received first, then finish work later.

Allows client to quickly move on, but client does not learn whether request succeeded.

In synchronous, delivery is acknowledged. In asynchronous, attempt is acknowledged.

Sometimes client will proceed immediately after send, but later want to now whether request succeeded or to get response data.

2.1 Request Record

Server can store request record in DB and return unique ID. When done, server updates request record in DB.

Client can later check on results using request ID.

2.2 Callback to Client

Client provides a callback function (webhook) where it expects to receive response. Only works if client can listen for responses (always running, not NATed, etc).

2.3 Side Channel for Feedback

Let clients send requests when failure is rare.

If failure occurs, report error to another system (like log).

3 Message Queues

Provide asynchronicity and decoupling.

If client doesn't care about response status, it can just put request on a queue.

Adding a message to a queue very fast since just data copy, without any parsing or business logic.

Prevents slowdown of upstream service due to downstream congestion, so system can handle short bursts of traffic beyond system capacity.

Putting message on queue similar to making API request, so content of message defines request.

Rules for formatting are a contract like an API.

Producer: pushes/publishes/produces messages

Consumer: pulls/pops/consumers/subscribes-to messages

Message queue can be partitioned into several virtual queues by assigning **topic** to each message, where consumers subscribe to some subset of all topics.

3.1 Tradeoffs

Tightly coupled (synchronous) services simpler to design/build.

Loosely coupled (asynchronous) services faster but:

- failures must be unimportant and ignored
- errors might be stored in DB and somehow checked later
  - can be difficult to sensibly react to error at later time
- errors can lead to alert to user later

  Decoupling helps scaling, since many producers and consumers can connect to the queue.

  Basic queues run on 1 machine and distributed queues run on clusters. Like load balancer, queue allows work to be distributed.

  Producer and consumer can be scaled separately as needed.

  Queue smooths demand peaks by deferring work.

3.2 Types of Queues

3.2.1 Passive Queue

Accepts and stores messages until they are requested (like a specialized DB).

Consumers must periodically request messages (poll).

Producer pushes and consumer pulls.

3.2.2 Active Queue

Queue knows where to send messages and actively pushes messages out to subscribers.

Subscribers must listen for messages.

Producer pushes and queue pushes to consumer.

3.3 Architectural Levels

In-app Queue: app defines its own queue to store work that it will do later, perhaps in a different thread

- simple and no separate app to deploy
- usually not stored on disk, so app crash/reboot may drop queued messages

Separate Queuing App: process that listens for pushes/fetches on a network connection

- can often run as a process on the same VM as the application pushing to it (communication stays local)
- can reside on existing app VM and can write queued messages to a file
- scalability limited to 1 machine so machine/disk failure drops messages

Distributed Message Queue: cluster of nodes that together implement a robust, scalable queue, allowing all work to go to the big queue

- massively scalable as messages are replicated on many nodes
- provides a single point of coordination for many producers/consumers
- complexity is an issue as well as consistency side effects (must choose delivery guarantee to be at least once or at most once, never exactly once)

3.4 Back Pressure

If queue fills up, it should be possible for queue to give error response to producer adding message.

This will stall service, so DevOps/operations staff should monitor queue sizes to anticipate such problems.

3.4.1 Ordering

Distributed queue cannot guarantee strict FIFO ordering of messages.

If messages must be ordered, send multiple as a single big message instead.

3.5 Implementation

Message queues should be connected to backend rather than frontend, since they are not designed to accept public requests or connections from 1000s of clients.