CS-310 Scalable Software Architectures

Lecture 16:
Asynchronous Processing

Steve Tarzia

Recap – Choosing a data store

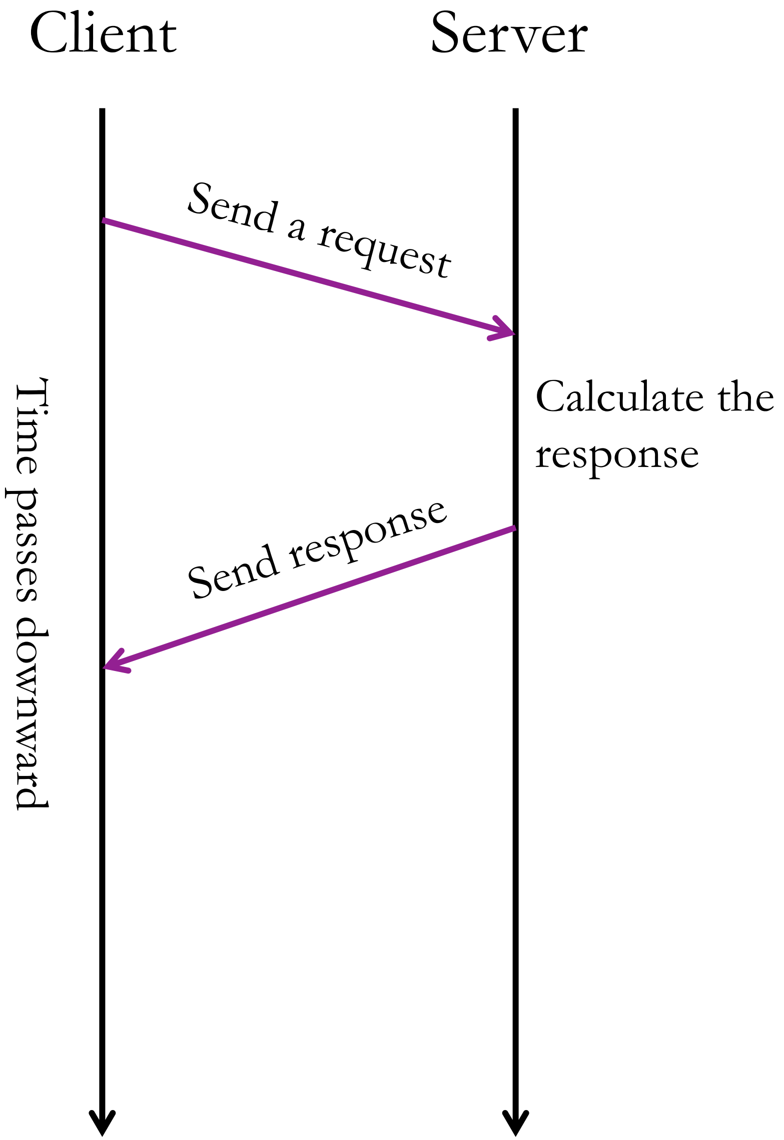
Data store	Examples	Data abstraction	
SQL Relational DB	MySQL, Oracle	Tables, rows, columns	Highly structured
Column-oriented DB	Snowflake, BigQuery	Tables, rows, columns	
Search engine	Elastic search	JSON, text	Semi-structured
Document store	MongoDB	Key → JSON	
Distributed cache	Redis	Key → value (lists, sets, etc.)	
NoSQL DB	Cassandra, Dynamo	2D Key-value (pseudo-cols)	
Cloud object store	S3, Azure Blobs	K-V / Filename-contents	Files with data "blobs"
Cluster filesystem	Hadoop dist. fs.	K-V / Filename-contents	
Networked filesystem	NFS, EFS, EBS	Filename-contents	

Your choice depends mainly on the **structure** of data and pattern of **access**.

- **Transactions** are easy on SQL DBs, available but slow on some NoSQL DBs

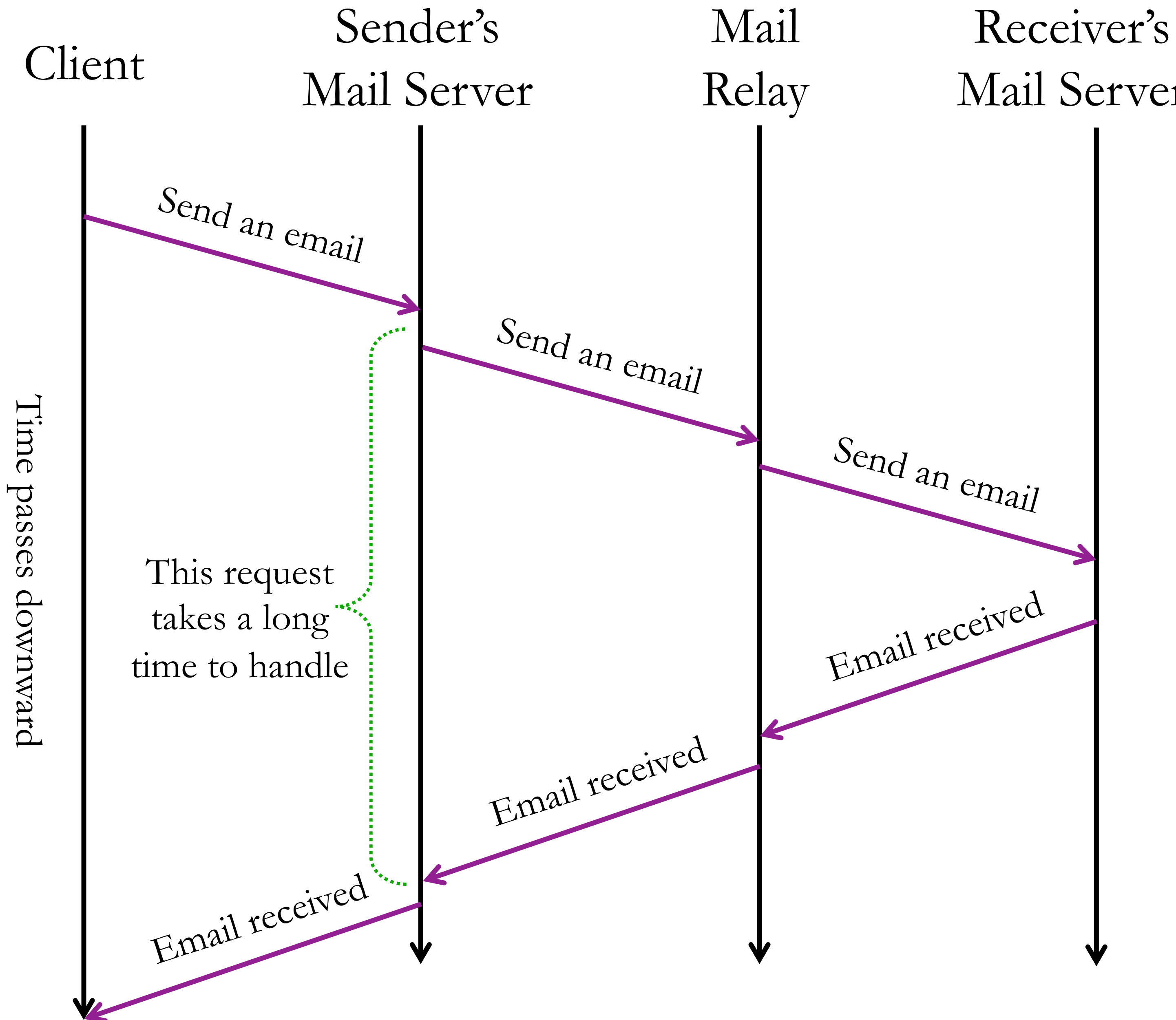
Ways to couple services

- **Responses** are important because:
  - Responses **acknowledge** that the request was received.
  - Responses may contain **data** that the client needs.
  - Responses may indicate a **failure** which the client must somehow react to (perhaps by retrying).
- So far, we have examined **synchronous** APIs.
  - The client waits for the server to finish processing, hence the two are *synchronized*.
  - Also called a **blocking** request.
  - This follows the pattern of HTTP and REST, which fetch documents/data.



But what if the request is really long?

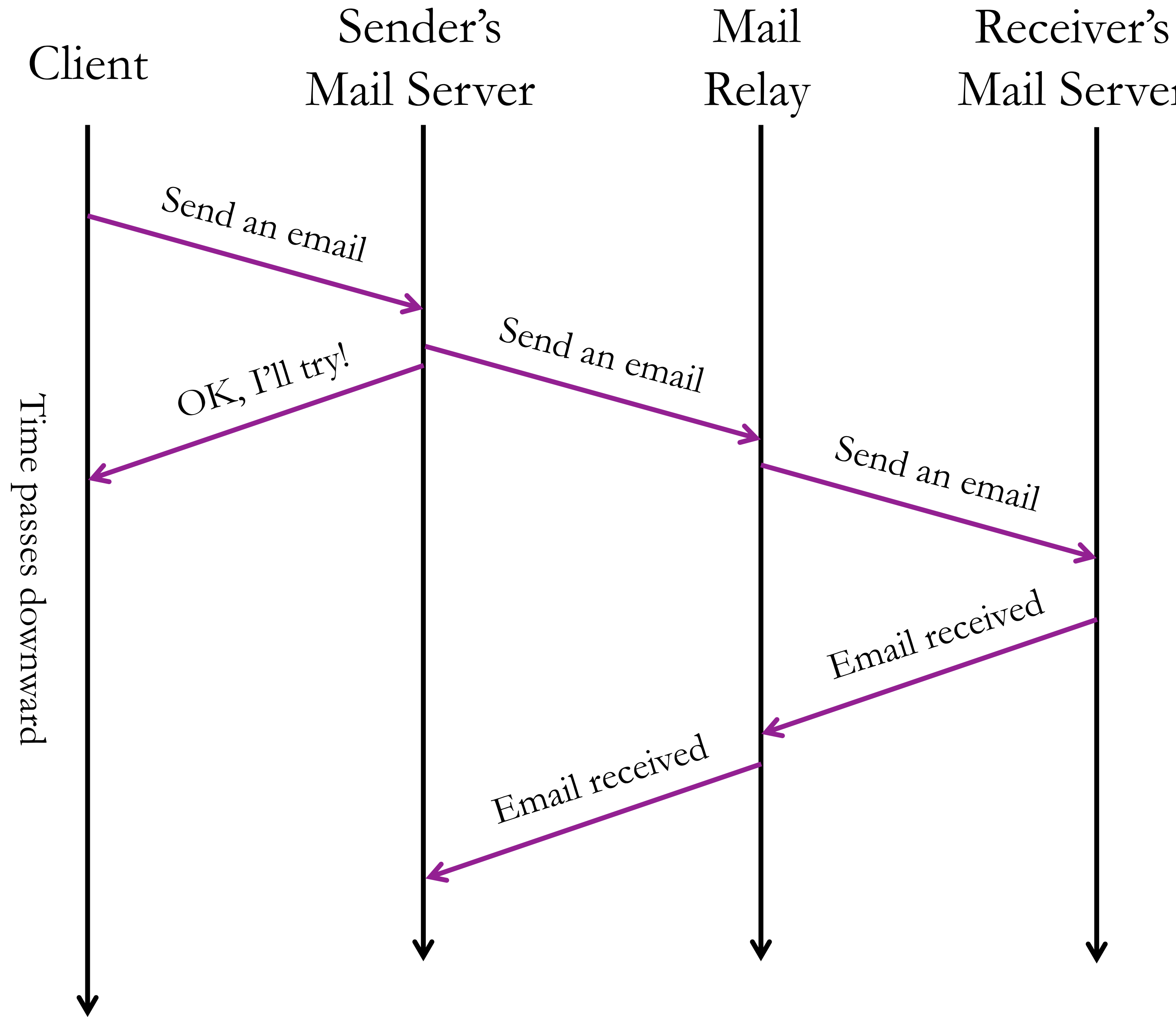
- For example, a request may involve lots of other services.
- The client may not care to wait until delivery of the message is verified.



Asynchronous Alternative

- Maybe it's OK to just acknowledge that the request was received, and finish the work later.

- **Pro:** Allows client to quickly move one.
- **Con:** Client does not learn whether the request succeeded.



Synchronous to Asynchronous – *what changed?*

- In both cases, a response was sent to the client.
- Both styles can be implemented with HTTP/REST.
- The difference is just the meaning of the requests and responses:

Synchronous style:

- *Request:* Deliver an email.
- *Response:* Delivery acknowledged.

Asynchronous style:

- *Request:* Send an email.
- *Response:* Attempt acknowledged.

What if client needs to know the results?

- The previous example was a *fire and forget* request, but sometimes the client wants asynchronous access to the results.
- Client wants to proceed immediately, but later will want to know whether request **succeeded** or to get response **data**.
- How can we support this?

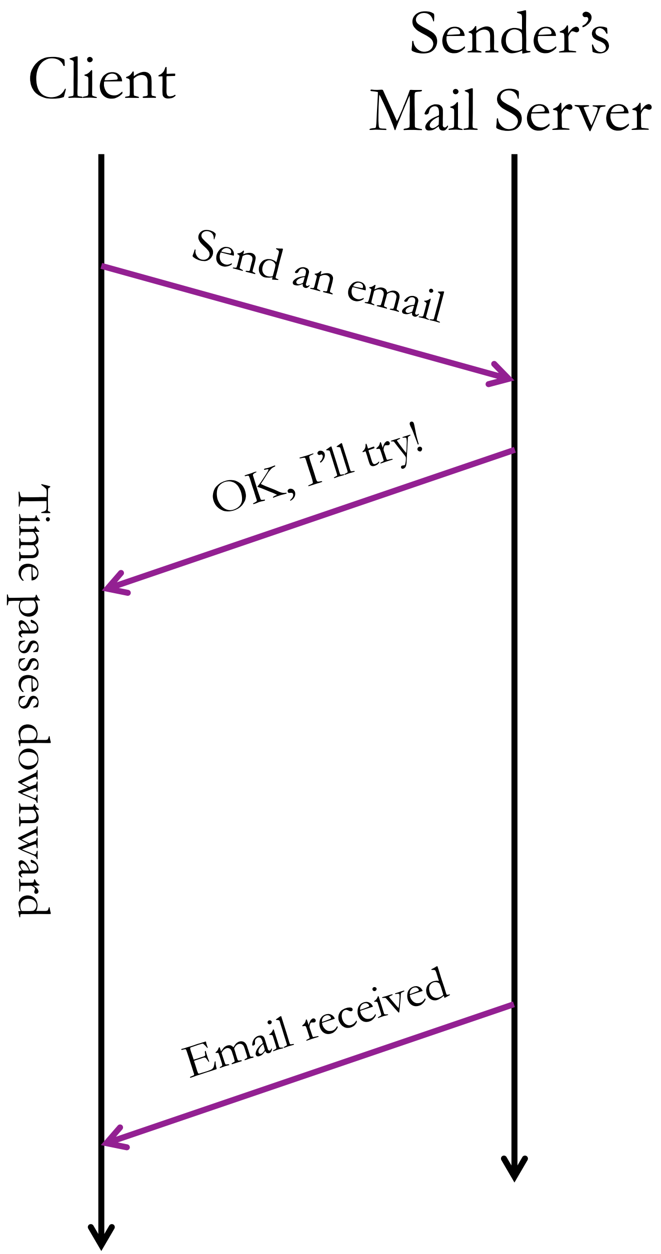
Option 1: Request Record

- Server can store a **request record** in a DB and return the unique id.
- When done, the server updates the request record in the DB.
- Client can later check on the results using the request id.
- Request → Response examples:
  - POST /messageAttempt → {“email_id”: 4390293}
  - GET /message/status/4390293 → {“status”: pending}
  - GET /message/status/4390293 → {“status”: failed,
“error”: “invalid address”}

Option 2: Callback to Client

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

- Request → Response examples:
  - *Client sends:* POST /messageAttempt {"callback": "http://3.3.3.3:80/messageComplete"} … *time passes* …
  - Backend sends: POST /messageComplete

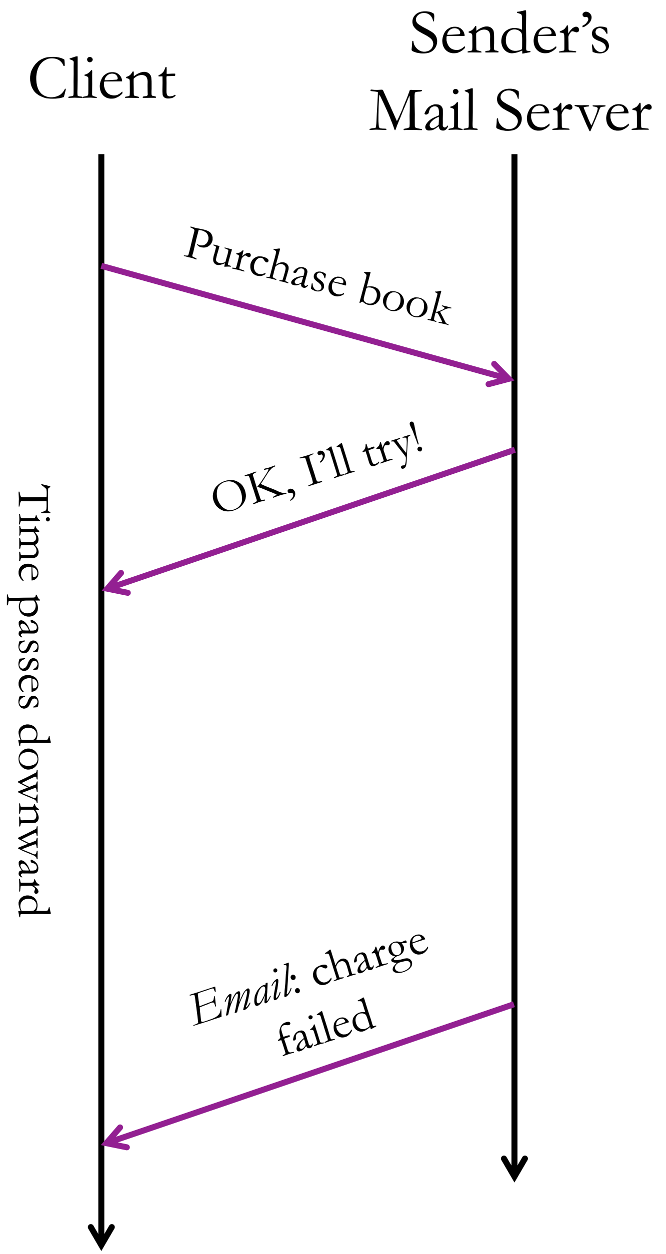


Option 3: Side channel for feedback

- Often, we let clients send "fire and forget" requests when failure is rare.
- When failure occurs, report the error to another system.

For example:

- Send customer an **email** if an order placed online fails (item was out of stock).
- **Log** an error for dev/ops or customer support staff to review.



Requests that would benefit from asynchronicity?

- Transfer a file over a network.
- Fetch a file from tape storage.
- Create a virtual machine (eg., EC2 on AWS)
- Purchase a shopping cart (ecommerce)
- Book a flight, concert, or sports ticket.
- LinkedIn connection request.

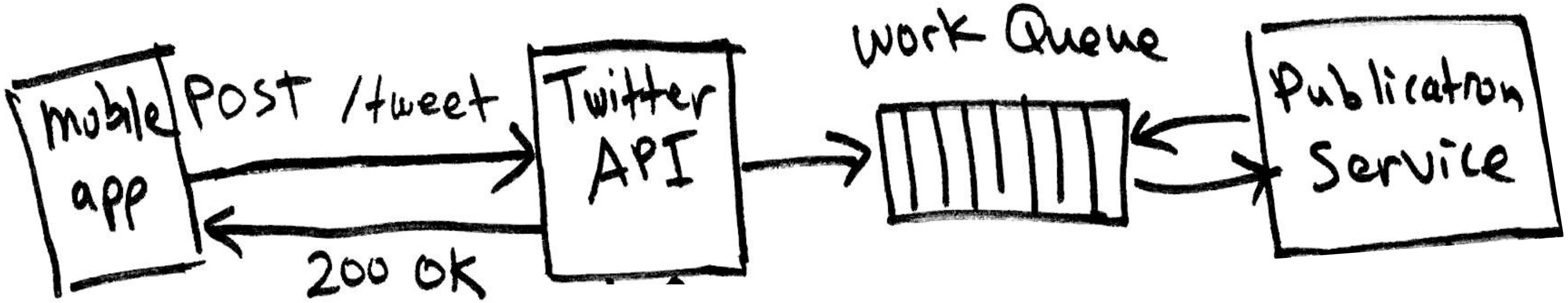
These use a different service (email) to communicate the response.

- Send a mobile push notification to another user.
- Send a text/SMS message.
- Copy tweet to 8 million follower's feeds.
- Google search: user clicked a particular search result
- Amazon.com: user searched for "dog toothbrush"

These will be used to refine the recommendation system. It's not urgent.

Message Queues provide asynchronicity & decoupling

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



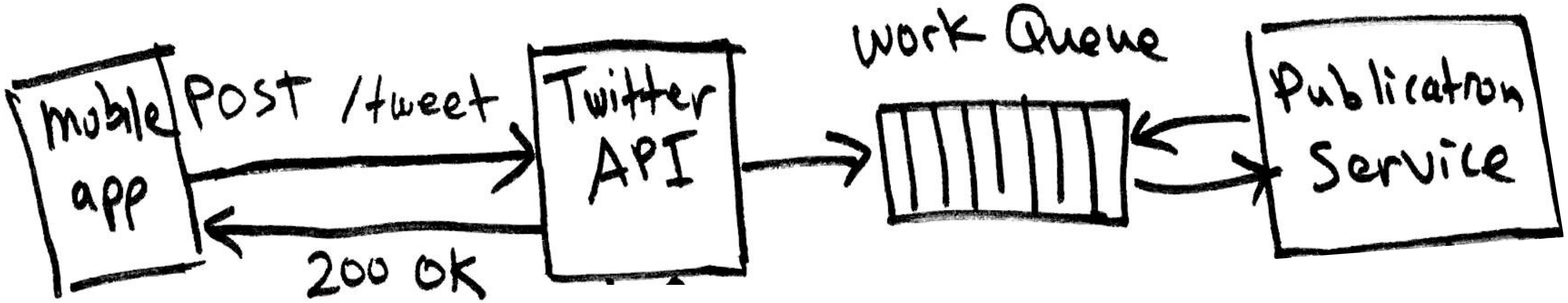
- Adding a message to a queue is very fast because it's just a data copy, without any parsing of the message or business logic.
- Prevents slowdown of upstream service due to downstream congestion. Can handle short bursts of traffic beyond system capacity.

Message Queues store requests ready to be handled

- Putting a **message** on a queue is like making an API request.
- The content of the message defines the request.

Message format == API

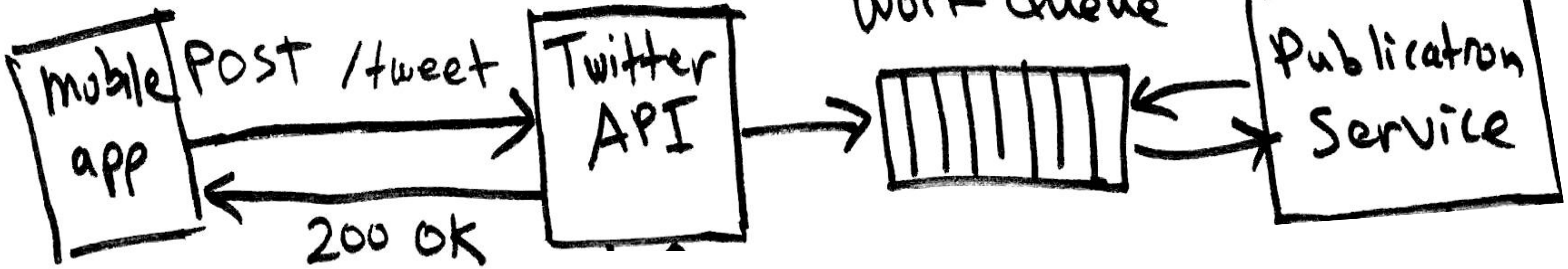
- The rules for formatting messages are a **contract** like an app's API.



Queue terminology

- **Producer**: pushes/publishes/produces messages.
- **Consumer**: pulls/pops/consumes/subscribes-to messages.

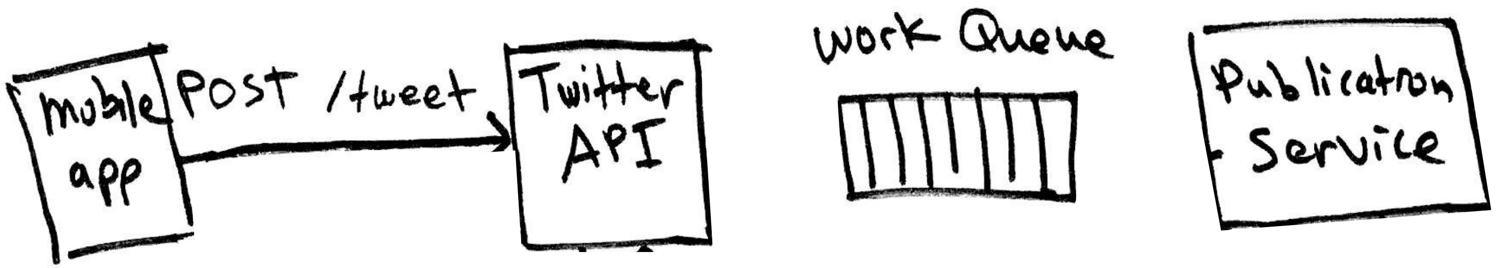
- Optionally, a message queue can be partitioned into several virtual queues by assigning a **topic** to each message.
  - Consumers may subscribe to just one or a subset of the topics.



Producer

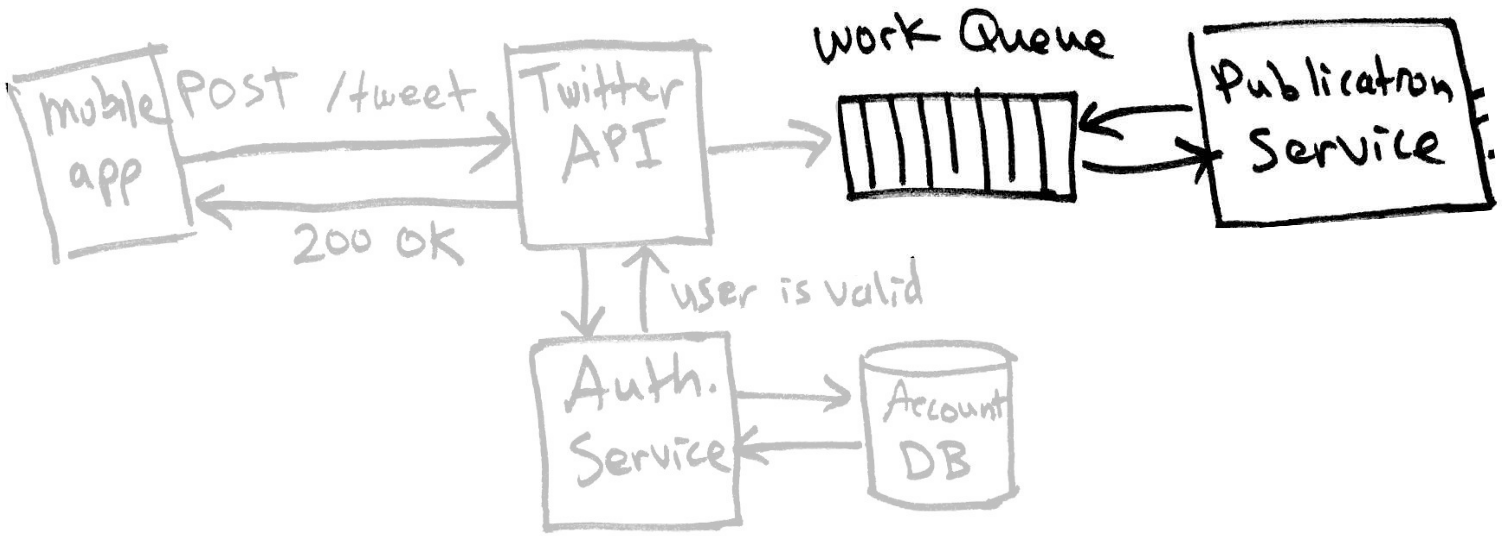
Consumer

Client posts request



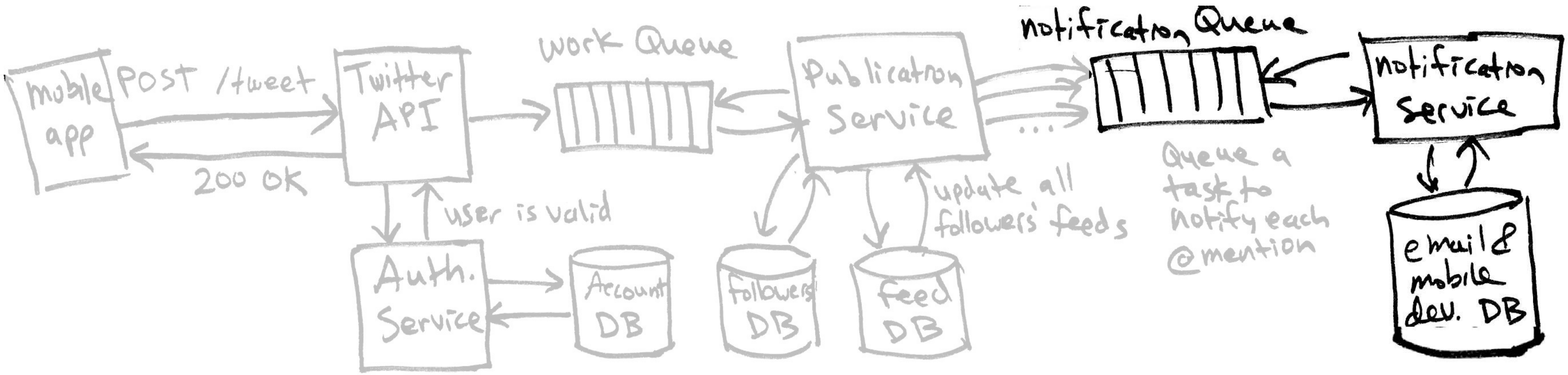
- API service receives request.
- Sends a request to auth. service to check that user token is valid.
- Puts a tweet-creation job on a queue, to be processed later.
- Sends "success" response back to user.
  - Is this premature?

Later, the Publication service handles the request



- Publication service fetches a job. It's a "publish tweet" job.
- Gets a list of followers to receive the tweet.
- Add the new tweet to all of the followers' feeds. *(Maybe millions!)*
- Queue another task to notify each of the @mentions and followers with alerts enabled. Notifications are not critical and may be slow.

Finally, the Notification service alerts users



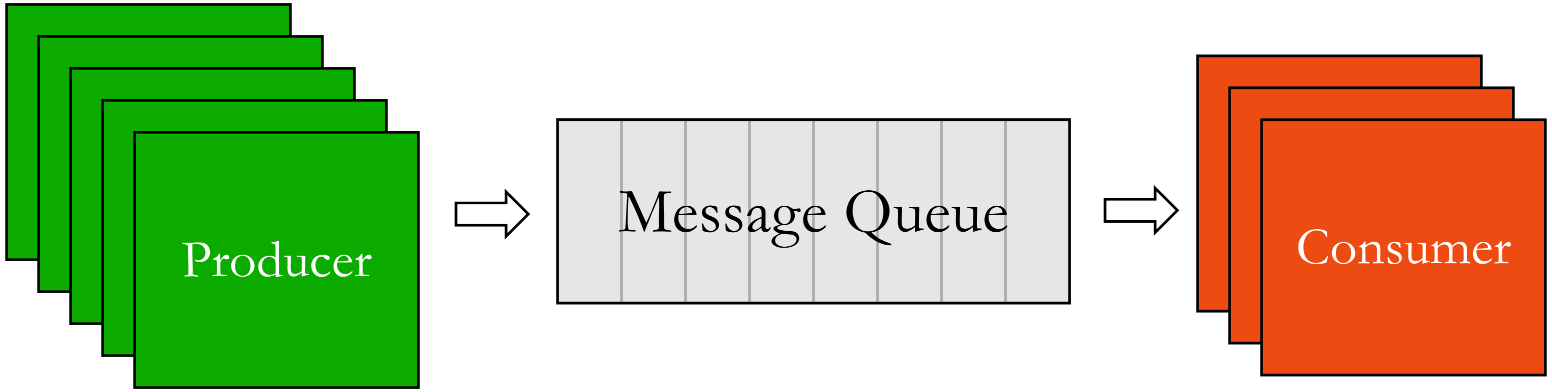
- There may be between zero and millions of notification tasks in the queue associated with the original tweet.
- Notification service dequeues each one and handles it.
- What happens if there is a failure?
  - Retry a few times, and then give up. The original tweeter does not care.

Tradeoffs

- Tightly coupled (synchronous) services are simpler to design & build.
- Loosely coupled (asynchronous) services can be faster, but either
  - Failures must be unimportant and ignored, or
  - Errors might be stored in a DB and somehow checked later.
    It can be very difficult to sensibly react to an error at a later time.
  - Errors might lead to some kind of an alert to user later (email?).

Decoupling helps scaling

- Msg Queues are a simple kind of database – store work requests.
- Many producers and many consumers can connect to the queue.
- Basic queues run on one machine & distributed queues run on clusters.
- Like a load balancer, a queue allows work to be distributed.
- Producer and consumers can be scaled separately, as needed.
- Queue *smooths* demand peaks by deferring work.



Active and passive queues

Passive Queue

- The queue accepts and stores messages until they are requested.
  - Queue is a specialized DB.
  - Maybe implemented as a DB table.
- Consumers must periodically request messages (poll).
- Producer pushes and consumer **pulls**.

Active Queue

- Queue knows where to send messages.
- Queue actively pushes messages out to subscribers.
- Subscribers must listen for messages.
- Producer pushes and queue **pushes** to consumer.

Some queue software supports both modes of operation.

Queues at different architectural levels

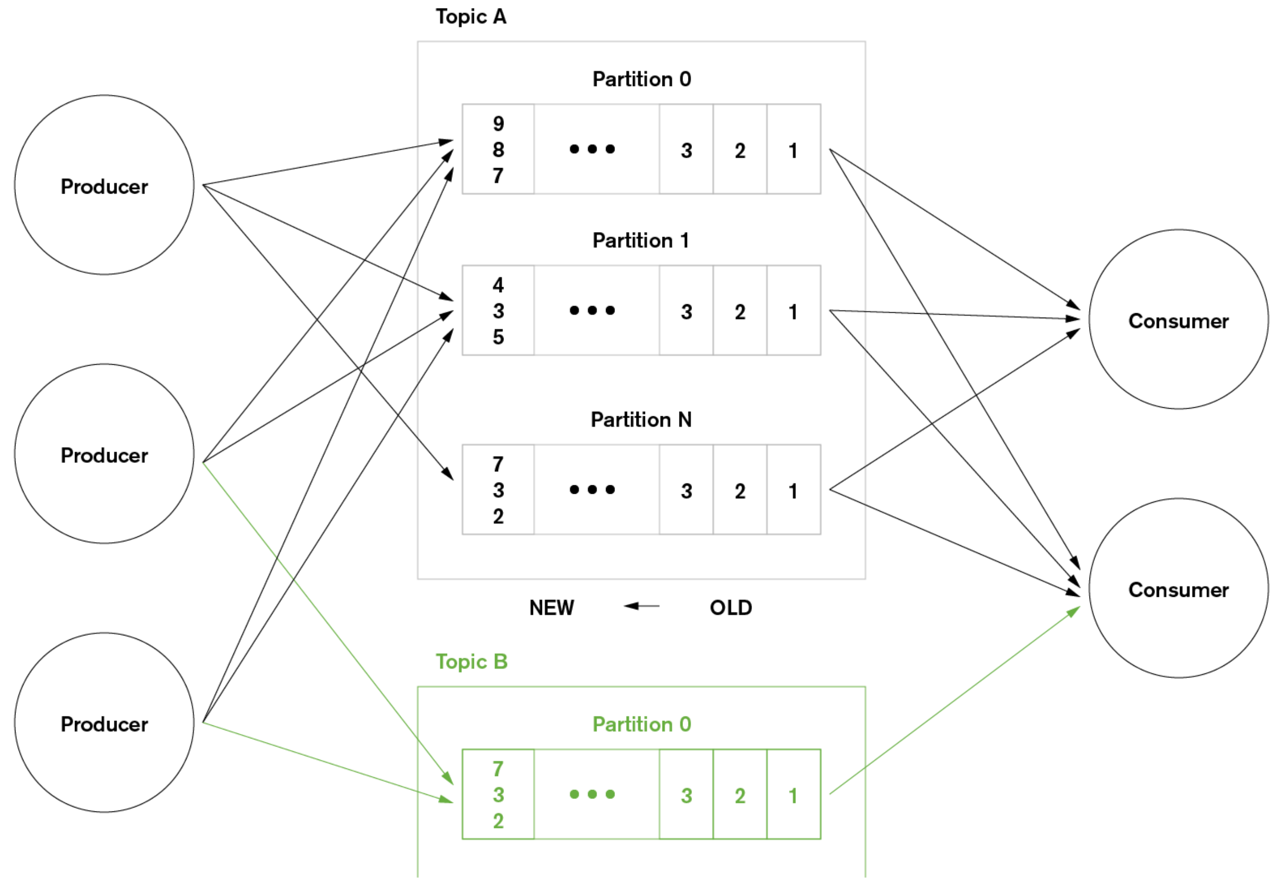
- **In-app queue**: an app can define its own queue to store work that it will do later, perhaps in a different thread.
  - For example, Java ExecutorService includes a work queue.
- **Separate queueing app**: a process that listens for pushes/fetches on a network connection.
  - Often it can run as a process on the same VM as the application pushing to it. In this case, the push's network communication is local.
  - For example, Netflix's Suro.
- **Distributed message queue**: a cluster of nodes that together implement a robust, scalable queue.
  - Allows all work to go to "one big queue."
  - For example, ActiveMQ, Kafka, RabbitMQ, AWS SQS

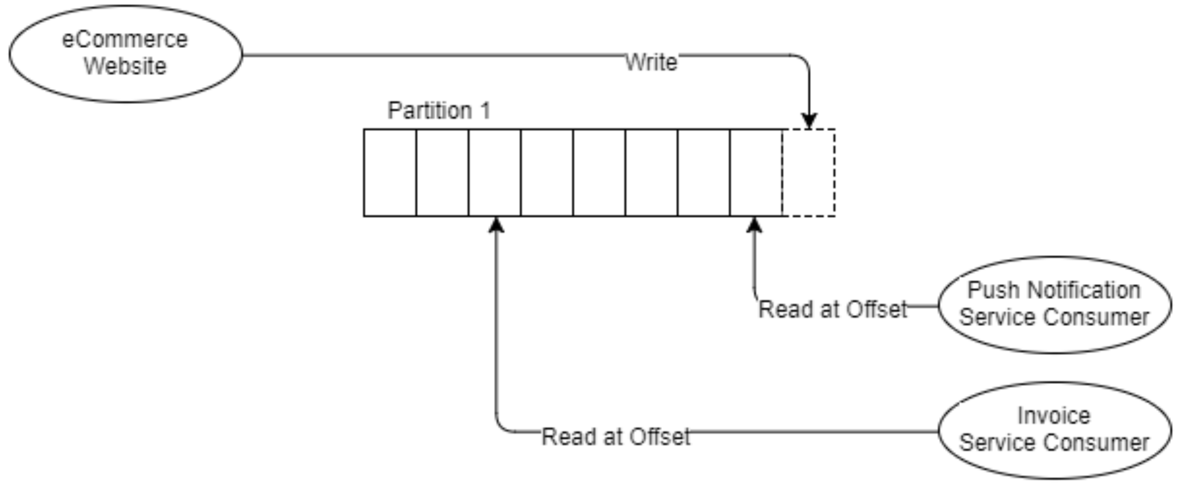
Pros and Cons

- **In-app queue**:
  - **Pros:** Simple. No separate app to deploy.
  - **Cons:** Usually not stored on disk. App crash/reboot may drop queued msgs.
- **Separate queueing app**:
  - **Pros:** Can reside on existing app VM. Can write queued msgs to a file.
  - **Cons:** Scalability is limited to one machine. Machine/disk failure drops msgs.
- **Distributed message queue**:
  - **Pros:** Massively scalable. Messages are replicated on many nodes. Provides a single point of coordination for many producers and consumers.
  - **Cons:** Complexity. *Consistency* side effects: eg., on RabbitMQ must chose delivery guarantee to be either "at least once" or "at most once" but cannot get "exactly once."

Eg., Kafka

- It's actually a distributed commit log, not a queue.
  - Messages are kept after reading.
- Like a DHT, data is partitioned onto multiple nodes.
- Multiple "Topics" are like separate queues.
- Uses Zookeeper for consumers to agree on point in the log to start reading.





Back pressure

- What happens if a queue "fills up?"
- It should be possible for the queue to give an error response to the producer trying to add to it.
- This is a bad thing because it will stall the service.
- DevOps/Operations staff should monitor size of queues to anticipate these problems.

Ordering

- Distributed queue cannot guarantee strict FIFO ordering of messages.
- *Tip*: If multiple messages must be ordered, send one big message.

Message Queues are backend creatures

- Like databases, messages queues are not designed to accept public requests or connections from thousands of clients.
- Your frontend should not connect directly to a Message Queue.

Recap – Message Queues.

- Services can be tightly or loosely coupled (synchronous or async.)
- Results from asynchronous calls are less apparent.
  - (fire-and-forget, request record, or callback)
- APIs can be asynchronous.
- Queues can be used to decouple systems.
  - Acts as a kind of deferred-work load balancer.
  - Allows producers and consumers to be scaled separately.
- Queues are useful at many levels:
  - In-app queues
  - Separate queueing apps
  - Distributed message queues.