Message Queues: Architecture & Technical Aspects

Maayan Hanin

Software Architect

maayanh@codevalue.net

@MA_Hanin

OzCode

Your Road to Magical Debugging

CodeValue

Inspiring Code, Creating Value.

https://pixabay.com/en/people-foot-waiting-line-queue-431943/

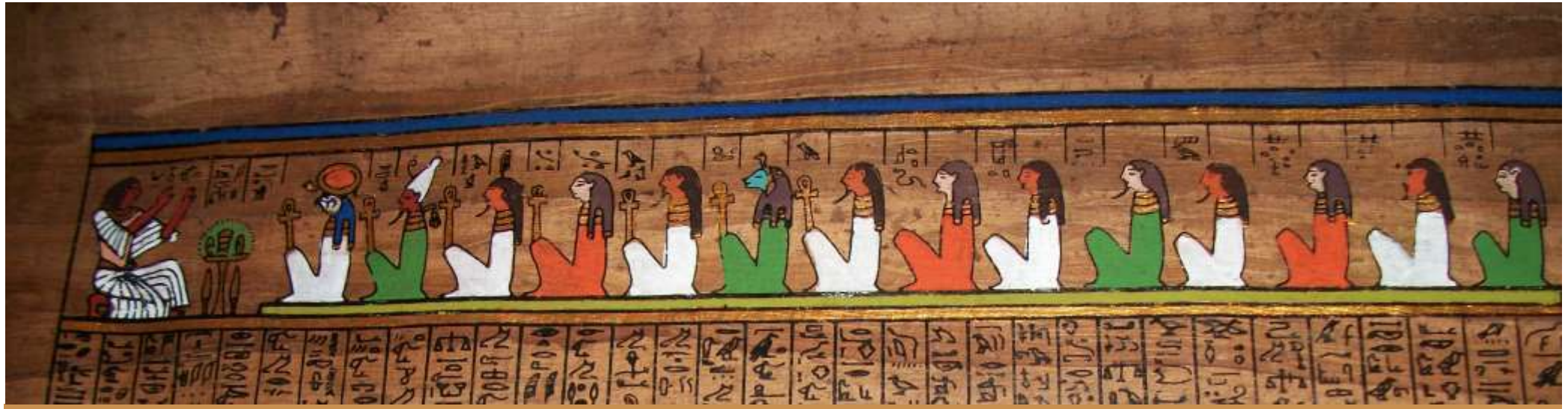
https://commons.wikimedia.org/wiki/File:Residents_of_Ponce,_Puerto_Rico,_line_up_at_an_ATM_in_hopes_of_getting_some_cash._More_than_a_week_after_Hurricane_Maria_struck,_residents_are_waiting_in_long_lines_to_withdraw_money_and_for_gasoline.jpg

https://pixabay.com/en/airport-tourism-flying-air-traffic-1515434/

מאת הנס חיים פין - www.gpo.gov.il, נחלת הכלל, https://commons.wikimedia.org/w/index.php?curid=3351742

https://picryl.com/media/cabo-rojo-puerto-rico-waiting-in-line-at-the-surplus-commodities-office-fatback-4
https://picryl.com/media/men-and-women-with-luggage-waiting-in-line

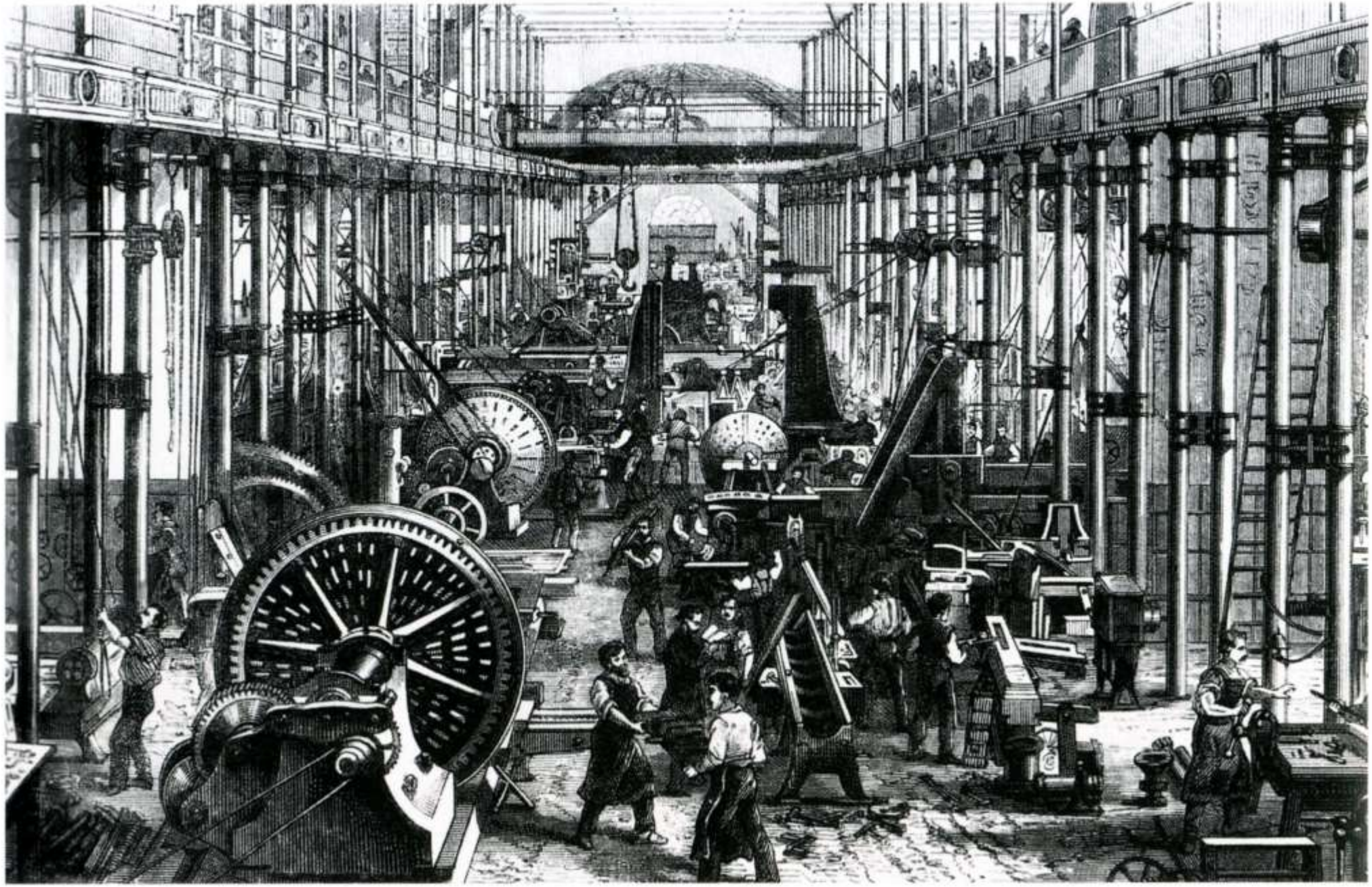
https://commons.wikimedia.org/wiki/File:Noahs_Ark.jpg *Noah's Ark*, oil on canvas painting by Edward Hicks, 1846



Not Queues!

https://en.wikipedia.org/wiki/Maat
https://www.goodfreephotos.com/egypt/cairo/old-egypt-hieroglyphs-cairo-egypt.jpg.php

https://commons.wikimedia.org/wiki/File:Last_Judgement_(Michelangelo).jpg



https://commons.wikimedia.org/wiki/File:Hartmann_Maschinenhalle_1868_(01).jpg
By Unknown author (scan by Norbert Kaiser) [Public domain], via Wikimedia Commons

https://commons.wikimedia.org/wiki/File:Noahs_Ark.jpg *Noah's Ark*, oil on canvas painting by Edward Hicks, 1846

About Me

- Maayan Hanin
  - maayanh@codevalue.net
  - @MA_Hanin
- Software architect, developer and debugger
- Work at CodeValue since 2017
- 12 years in the software industry
  - Infrastructure
  - Backend
  - Distributed systems

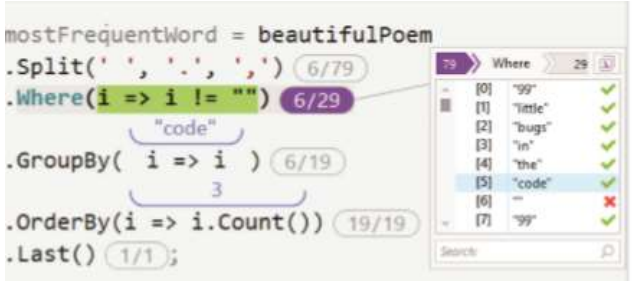
About CodeValue

- Awesome software company!
- About 200 employees, most of which technology experts
- High quality software development solutions

OzCode – Debug Like a Wizard

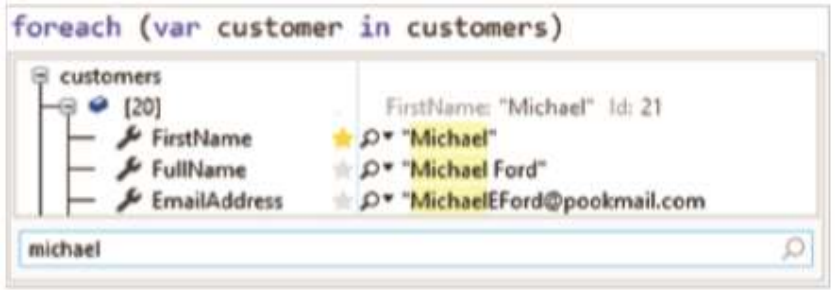
Quit debugging, spend more time writing brilliant software

LINQ Debugging / Know the flow of your LINQ queries

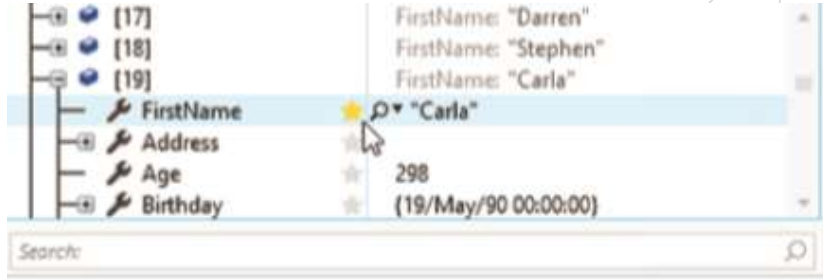


Magic Glance / Figure out complex expressions

Search/ Find that needle in a haystack of data



Reveal / Focus on data that actually matter



With our Visual Studio extension for C#, follow the road to a bug-free world

oz-code.com | @oz_code

OzCode
Magical Debugging

Agenda

- Systems Integration
- What are message queues?
- What are message queues good for?
- Pub/sub
- Technical aspects
- Message streaming platforms

Systems Integration

Systems Integration

- *Or integration of apps, components, services, subsystems*
  - *Terms used interchangeably during this talk*

- Predominant methods:
  - File transfer
  - Shared database
  - Remote Procedure Calls (RPC)
  - Messaging

File Transfer & Shared Database

- Similar to messaging in nature
- Aren't meant for communication
  - Designed for data storage, retrieval and manipulation
- Applications have to compensate
  - Polling
  - Locking
  - Protocol
- Hard to scale
  - Centralized beasts in nature

The trouble with RPC

- Treating remote systems as if they were local
  - Leaky abstraction
  - Ignoring the fallacies of distributed computing
- Demands remote system to act *now*
  - Can’t defer execution
- Caller held responsible
  - Compensation actions when sequence fails
  - Synchronization when in doubt
- *Tight coupling*

The fallacies [edit]

The fallacies are:[1]

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

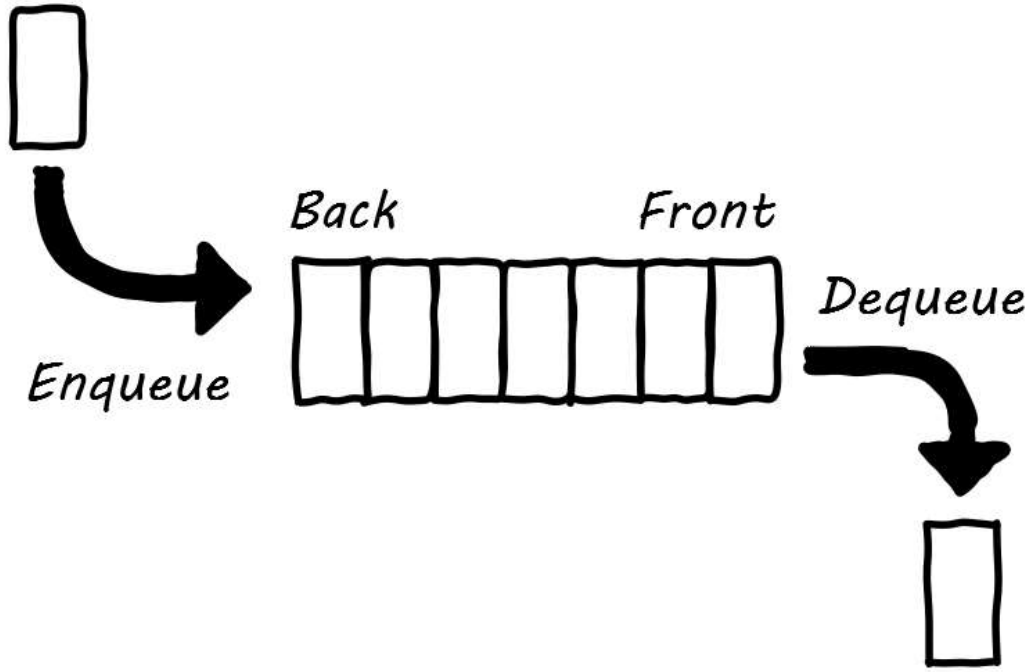
What are Message Queues?

Queues

- First in, first out data structure (FIFO)

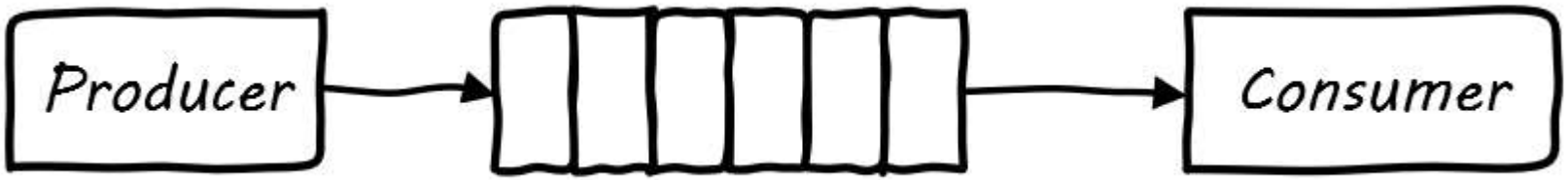
- Queue Operations
  - Enqueue
    - Put item in the back of queue
  - Dequeue
    - Take item out from front of queue

- Enqueue and Dequeue are fast
  - O(1) operations
  - Simple data storage actions
    - No complex processing



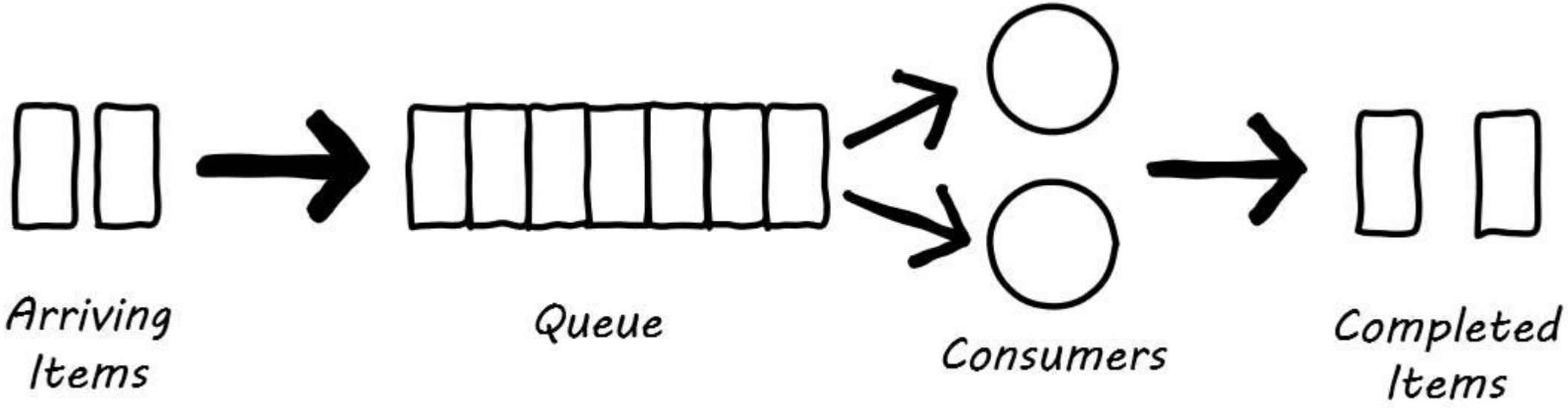
Queues

- Two logical roles
- Producer
  - Produces items
- Consumer
  - *a.k.a server, receiver*
  - Consumes items



Queues

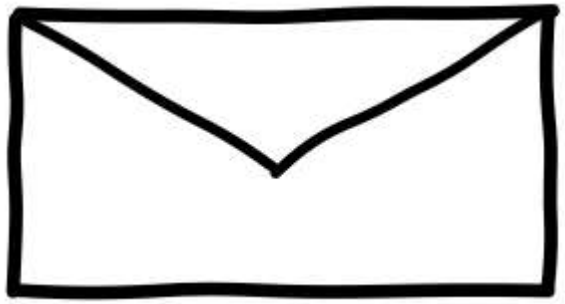
- There can be multiple consumers
  - Competing consumers
  - Performing the same function



Queues

- Additional queue operations
  - Count
  - Peek
- Things we can’t do
  - View / add / remove items from middle of queue
  - Query items in the queue

Message Passing



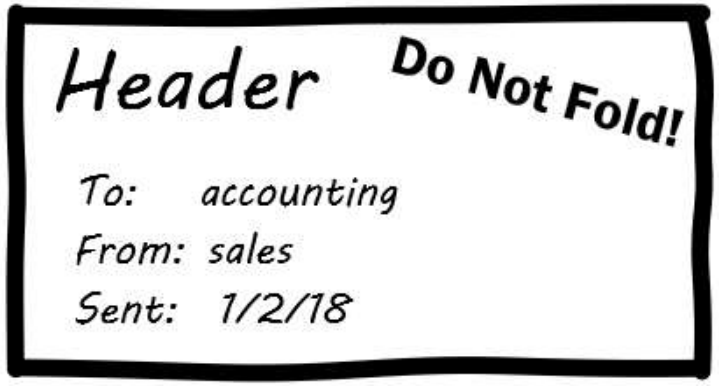
- *a.k.a messaging*

- A technique for invoking behavior

- Programs communicate by sending messages to each other
  - As opposed to calling each other directly
  - Invoker relies on receiver to execute the appropriate code

Message Structure

- Header
  - *The text written on the envelope*
  - Describe the data, its origin, destination etc.
    - May include application metadata
  - Understood and used by the messaging infrastructure



- Body (payload)
  - *The letter inside the envelope*
  - The actual data being transmitted
  - Typically ignored by the messaging infrastructure

Message Type Semantics

- Command
  - Invoke a procedure of another application
  - Encapsulates request as object
  - *“Do X”*
- Document
  - Pass data to another application(s)
  - *“Here is X”*
- Event
  - Pass notification to another application(s)
  - *“X happened”*

Message Type Semantics

- Document
  - Delivery is very important
    - Guaranteed delivery
    - Persistence
  - *Always* significant
    - No expiration
  - Timing is less important
- Event
  - Timing is very important
    - Low latency, high volume
  - Significant *now*, insignificant later
    - Message expiration
  - Delivery is less important
    - Message loss is more acceptable

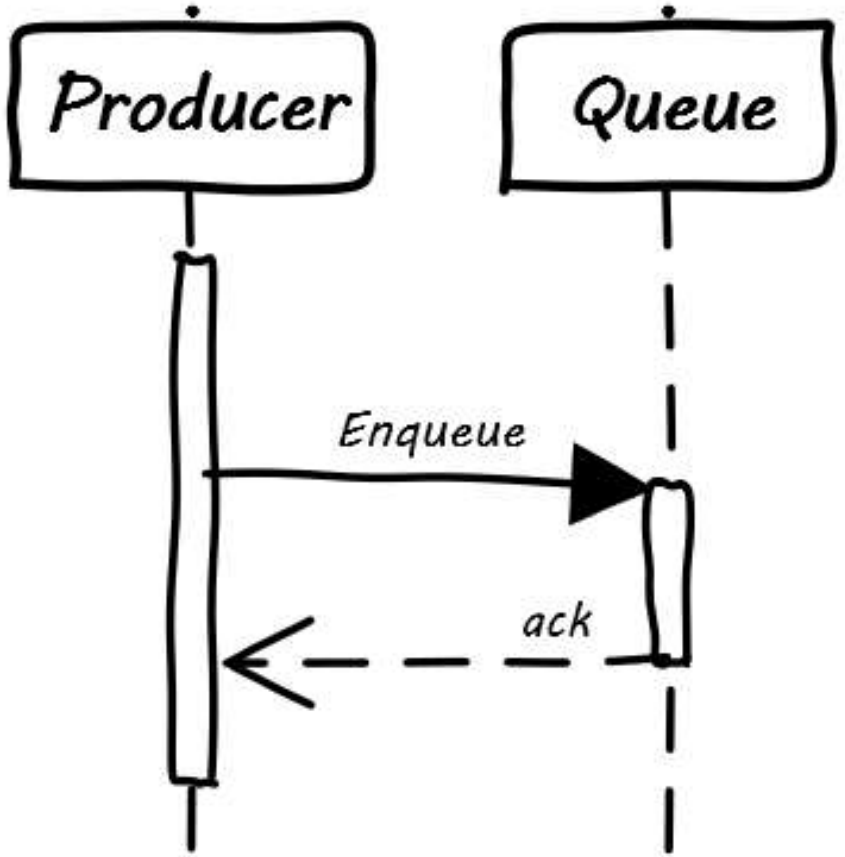
Message Queues

- Queueing + Messaging = Message Queueing
  - A queue of messages

- Messages in message queueing are:
  - One-way
  - Asynchronous
  - Typically small
    - Kilobytes to a few megabytes
    - Claim-check pattern for large payloads

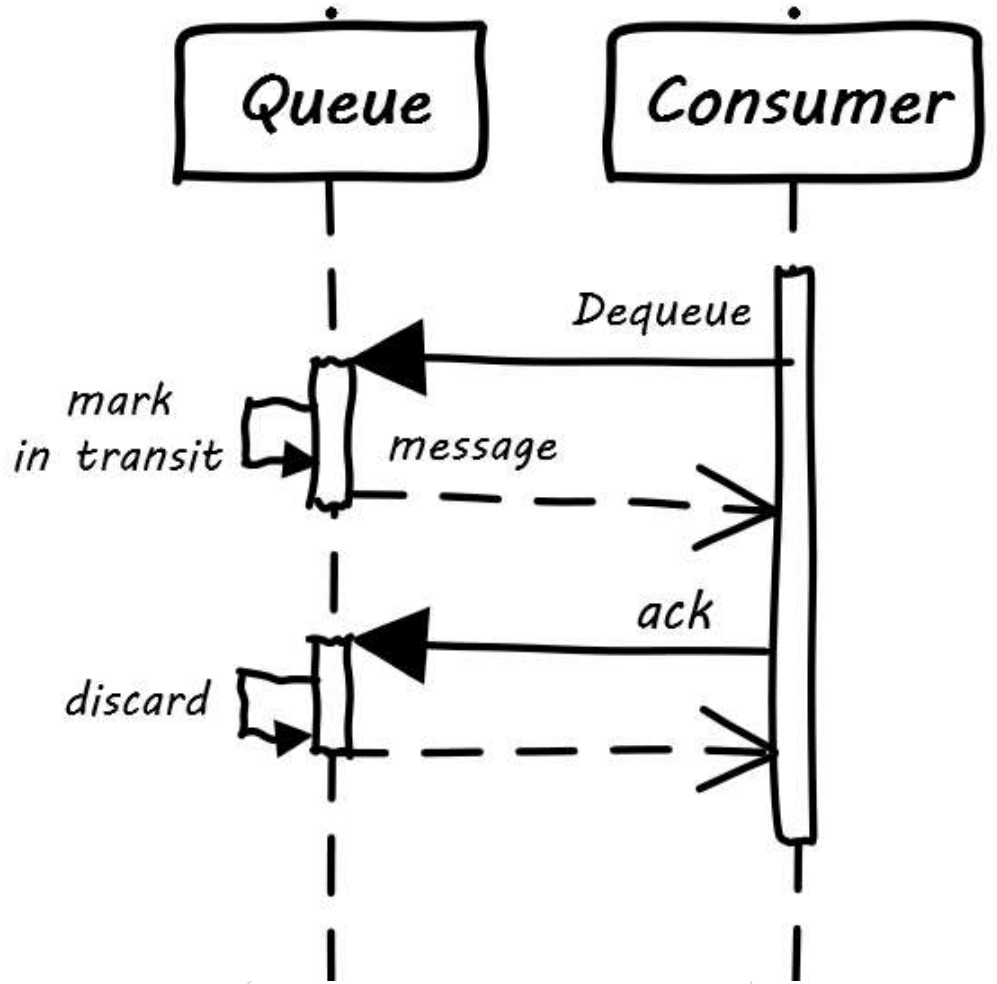
Message Queue Flows: Production

- Producer enqueues message
- Queue returns “ack”
  - Acknowledged
  - Message accepted
- Queue might return “nack”
  - Not Acknowledged
  - Message rejected



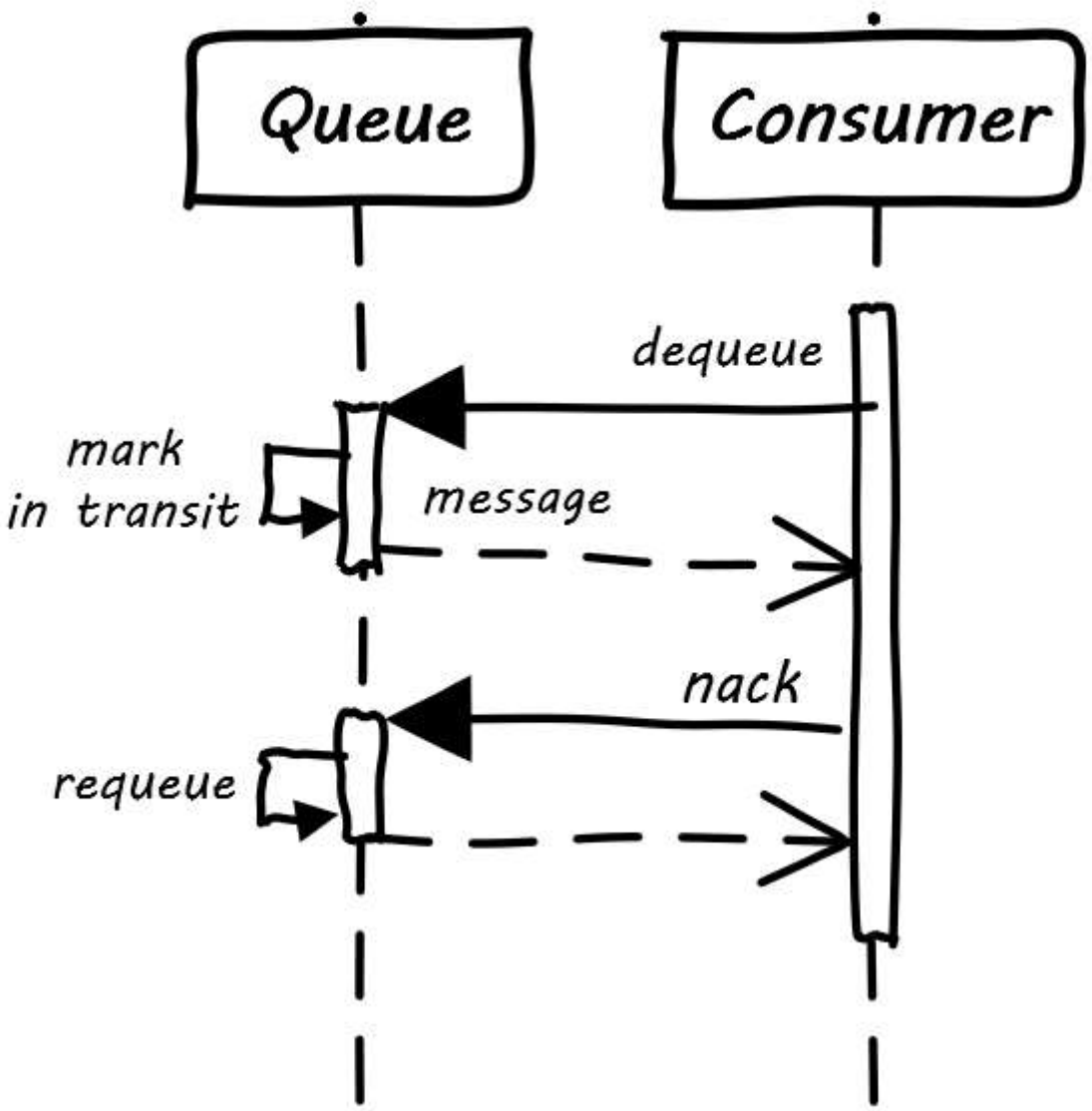
Message Queue Flows: Consumption

- Consumer dequeues message from queue
- Message marked as “in transit”
  - Won’t be delivered to other consumers
- Consumer can “ack” the message
  - Notifying message handled
  - Queue will delete message



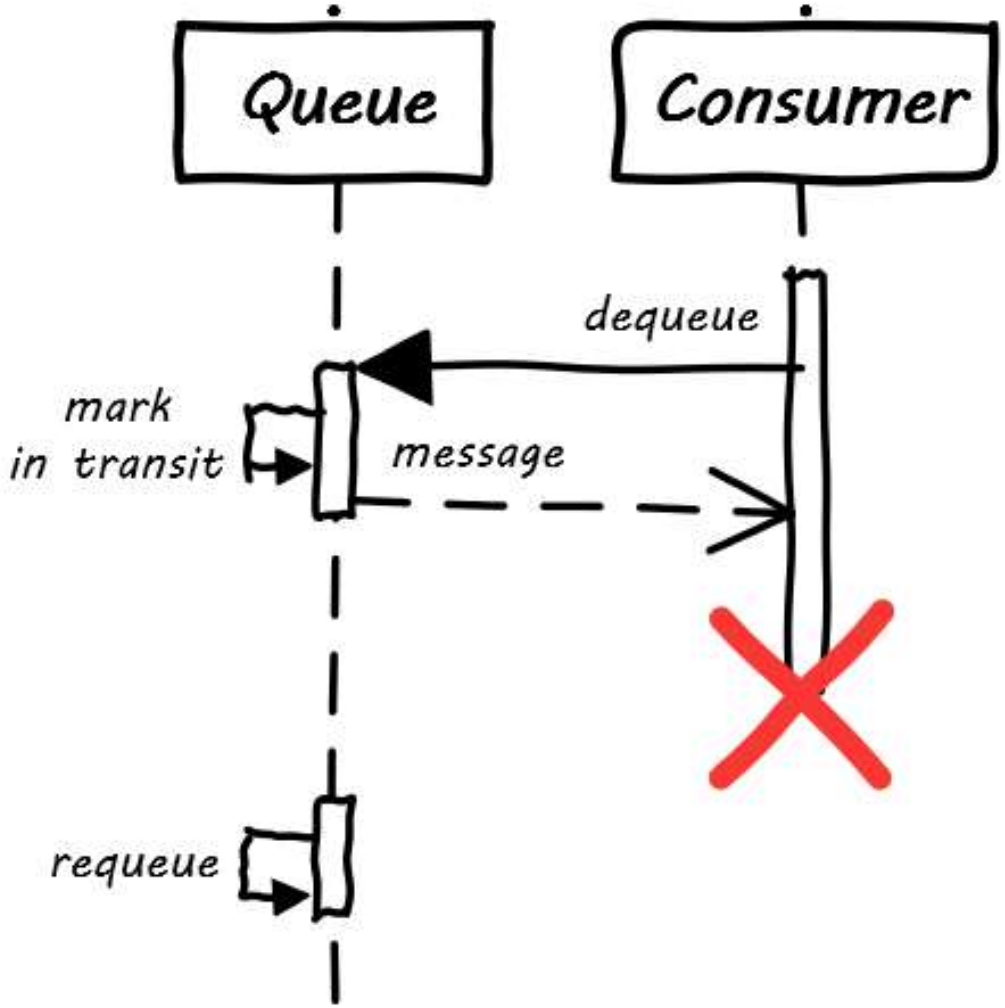
Message Queue Flows: Consumption

- Consumer may nack the message
  - Rejects message
  - Cannot be handled
- Transient nack / requeue
  - *“Can’t handle it now, perhaps later”*
  - Message will return to queue
- Permanent nack
  - *“Can’t handle it, ever”*
  - Queue will discard message



Message Queue Flows: Consumption

- Consumer may not reply at all!
  - Crashed
  - Timed out
  - Network disconnected
- Queue will detect issue
- Message will reappear in queue



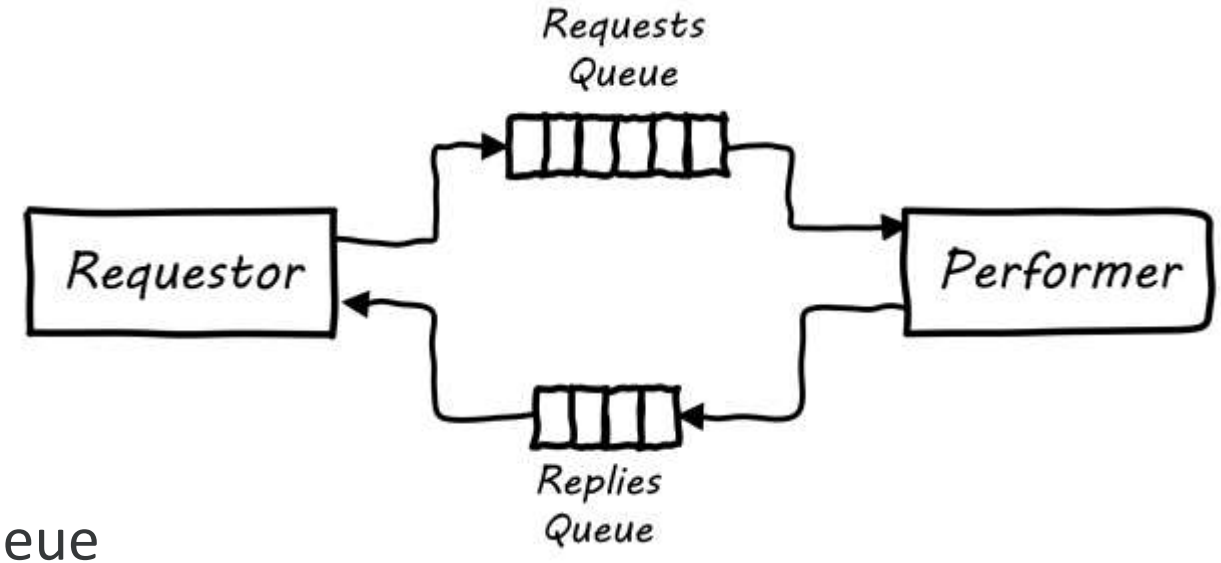
Bidirectional Messaging: Request/Reply

- Not all interactions fit the “one-way” paradigm

- Queries need to return data
- Commands may need to return results
  - Or errors
- Notifications may require acknowledgement

Bidirectional Messaging: Request/Reply

- Replies Queue
  - *a.k.a response queue*
  - Destination for returned values and errors



- Requestor sends command
  - Enqueues to performer's queue
- Performer executes command
- Performer sends response
  - Enqueues to requestor's replies queue
- Requestor processes the reply

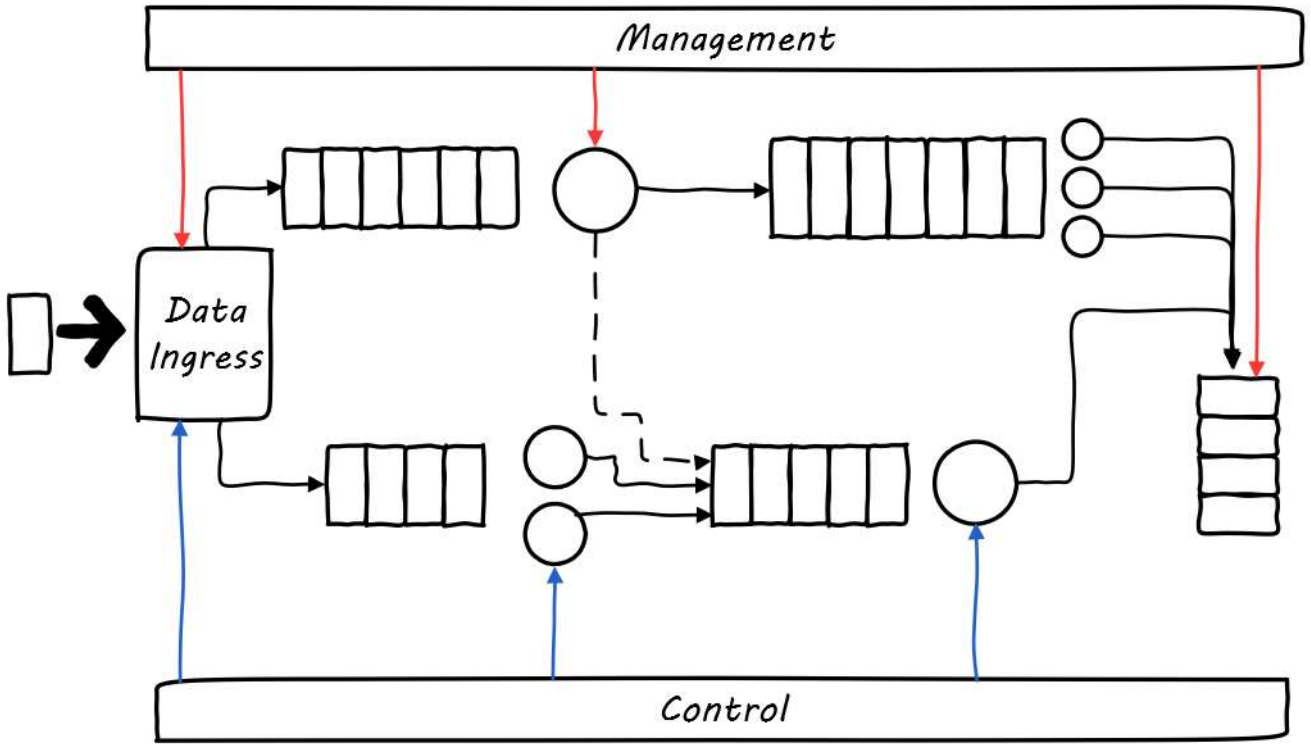
Message Queueing Systems

Message Queueing Systems

- A message queueing system to a queue is like database to a table

- From concept to product
  - Create & manage queues
  - Users & Permissions
  - APIs
  - Dashboards
  - Clustering
  - High availability
  - Pub/sub

- *MQ henceforth*



Message Queueing Systems

- *Sometimes known as*
  - *Message Oriented Middleware (MOM)*
  - *Messaging Systems*
  - *Queueing Systems*

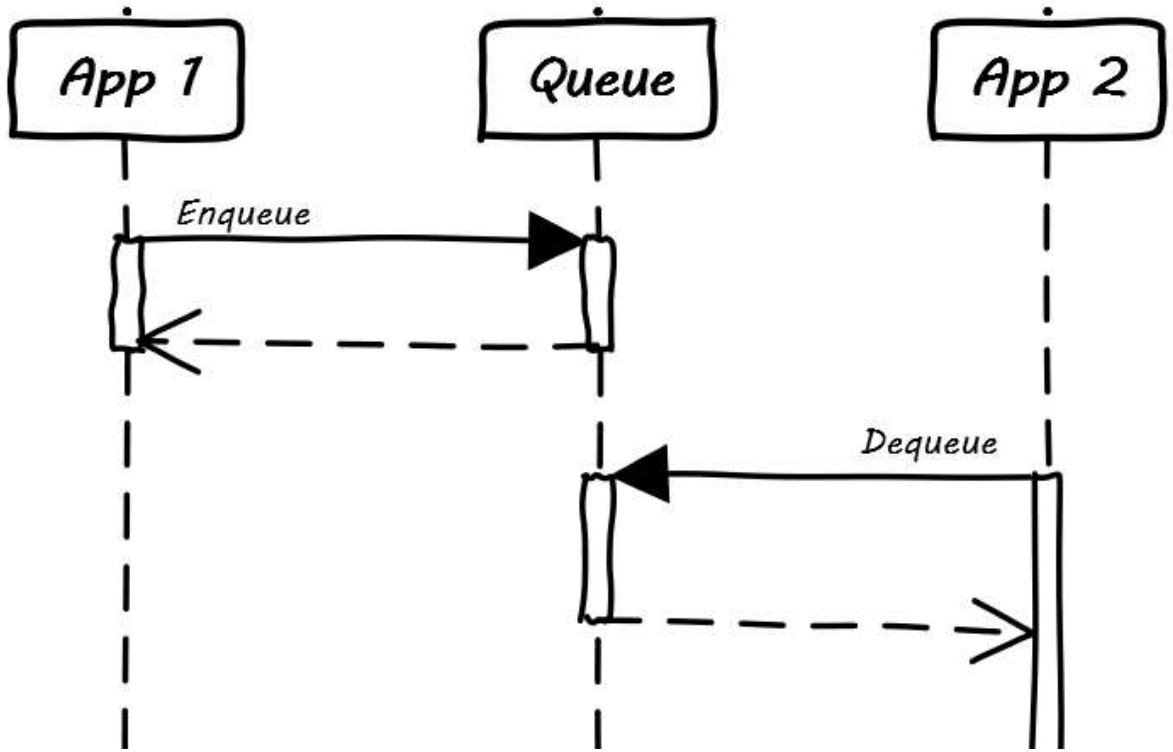
Message Queueing Systems

- Not to be confused with Enterprise Service Bus (ESB)
  - ESB provides functions that MQ doesn't
    - i.e. Transformation, Validation, Split, Merge, Ordering
    - "al-in-one" approach
  - MQ provide barebone functionality
    - Application deals with other concerns

Benefits of Message Queues

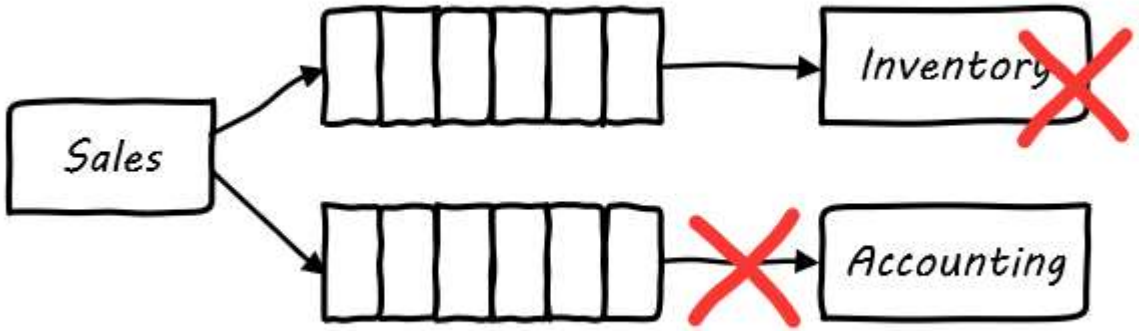
Message Queueing Benefits

- Temporal Decoupling
  - "Do X + Y + Z *now* and *in order*" stresses the system
  - Easier to do small, separate transactions



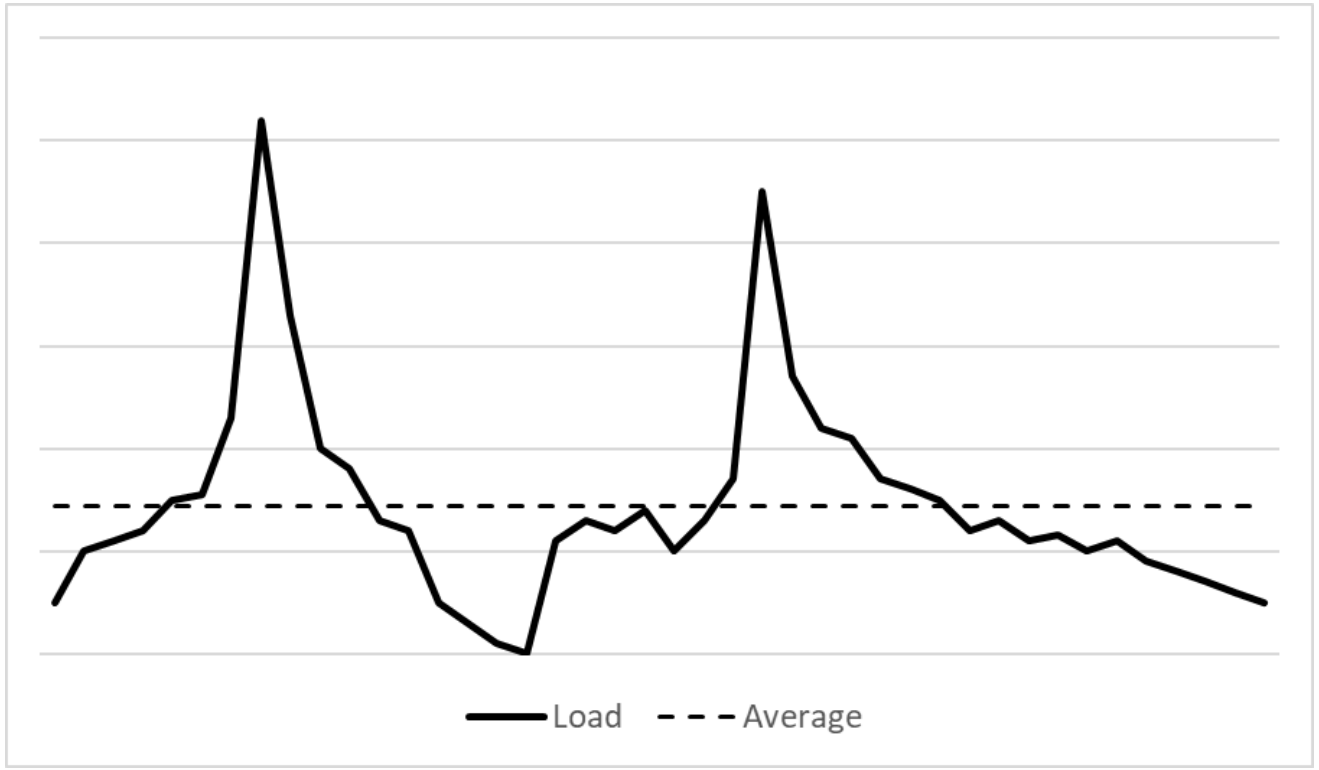
Message Queueing Benefits

- Reliability, Availability, Fault Tolerance
  - Separate parts can work when others are down
  - Can tolerate network disconnections
  - Can tolerate consumer failures
    - Messages will be re-delivered
  - Dead consumers simply don't pull messages



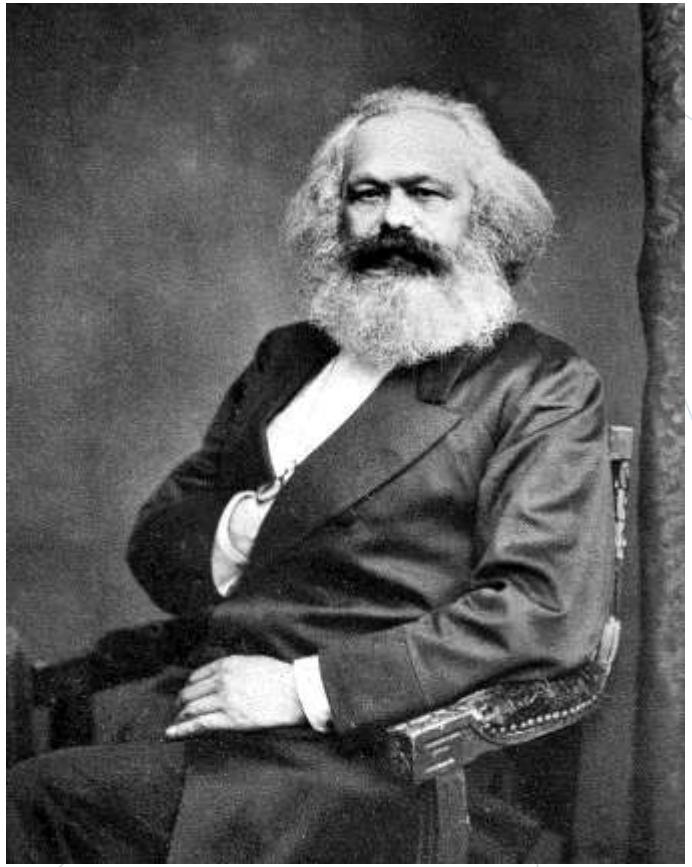
Message Queueing Benefits

- Load Leveling
  - System only needs to deal with average load
  - In peak load, messages accumulate in queue
    - Will be dealt with during the ebb



Message Queueing Benefits

- Load Balancing
  - Consumers pull work according to their ability to process it
    - Account for their own resources
  - No need for a load balancer
  - Communist utopia



Karl Marx

https://commons.wikimedia.org/wiki/File:Karl_Marx_001.jpg

Message Queueing Benefits

- Scalability
  - Can scale out efficiently
  - Add consumers and producers to handle greater load
    - Enqueue/dequeue locks are very short
- Performance Modeling
  - Queueing Theory enables modelling system performance
    - Predictions
    - Planning

Message Queueing Benefits

- Observability & Measurement
  - Load can be observed by various metrics
    - Arrival rate
    - Queue length
  - Performance can be measured
    - Wait time
    - Service time
    - Throughput
    - Utilization
  - Basis for auto-scaling

Message Queueing Benefits

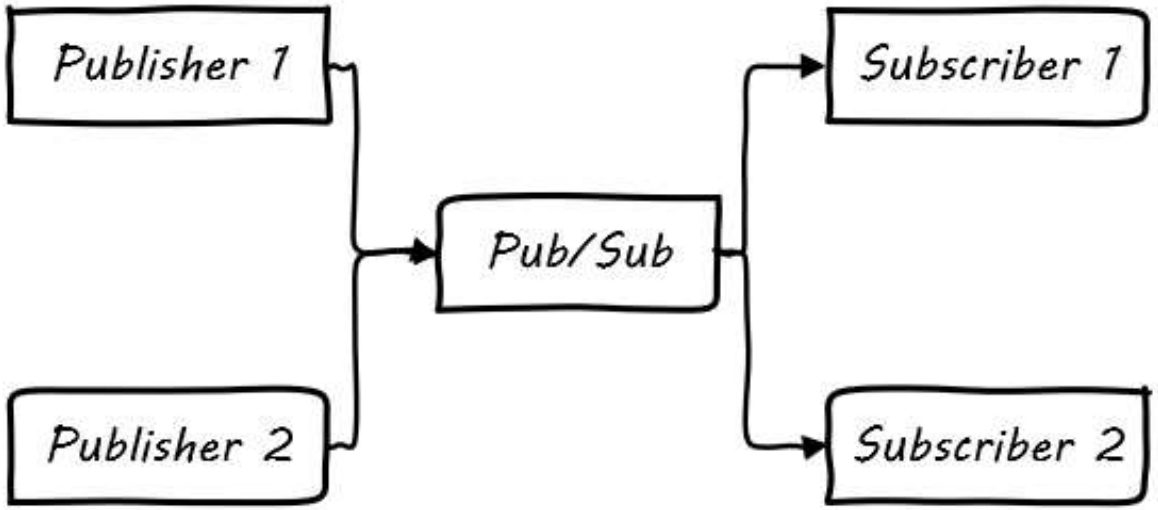
- Simplified programming model
  - Reduced plumbing code
  - Reduced error handling code
  - Reduced responsibility
    - Easier to argue about correctness

Pub/Sub

Beyond one-to-one relations

Pub/Sub

- *Publish-subscribe* design pattern
  - Publishers publishes messages
  - Subscribers subscribe to messages
    - Usually a subset of published messages
  - Pub/sub delivers published messages to relevant subscribers

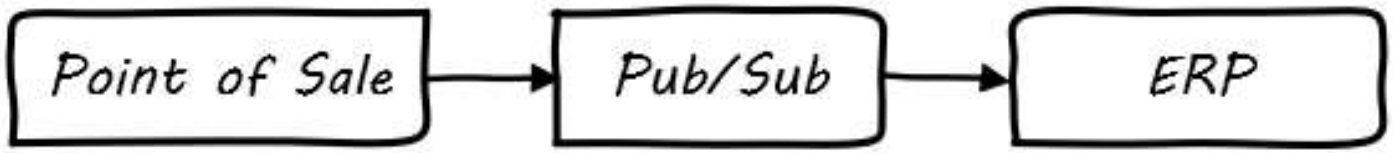


Pub/Sub: advantages

- Promotes loose coupling
  - Publishers don’t know subscribers
  - Subscribers don’t know publishers
  - Only data contract is shared
    - Message schema
- Basis for system extensibility
  - Add new functionality by adding publishers or subscribers
- Nature of interaction
  - Choreography vs. orchestration

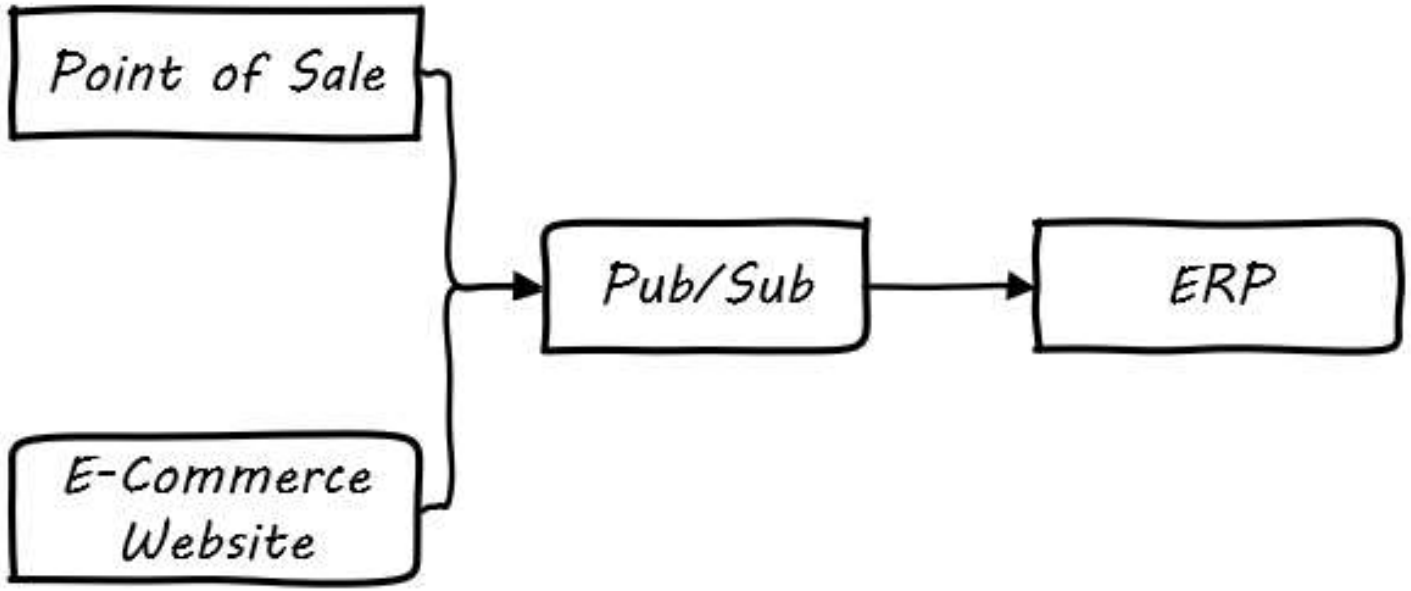
Pub/sub: example

- Initial system has Points of Sale (physical stores) and ERP system
  - Published data includes Invoices and Receipts



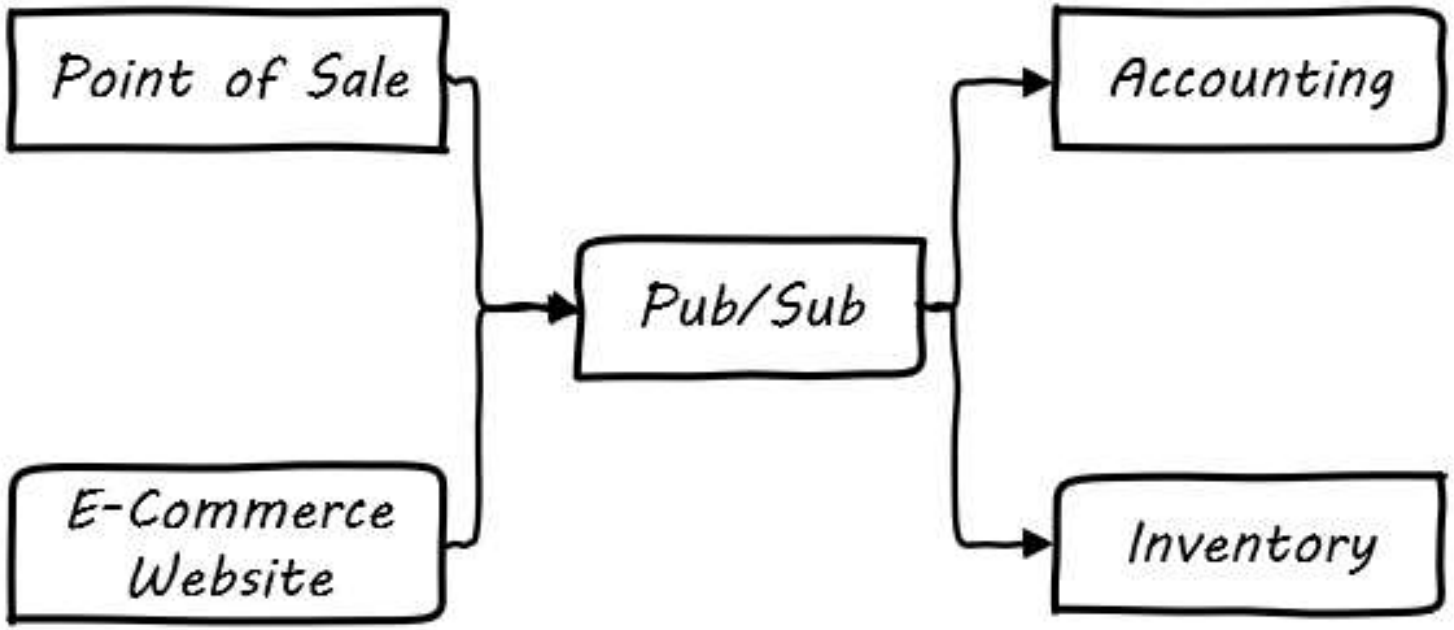
Pub/sub: example

- As the business evolves, it starts selling via e-commerce website
  - New publisher of the same content



Pub/sub: example

- Business continues to grow
  - Replaces ERP with dedicated Accounting and Inventory systems
  - New subscribers to the same content



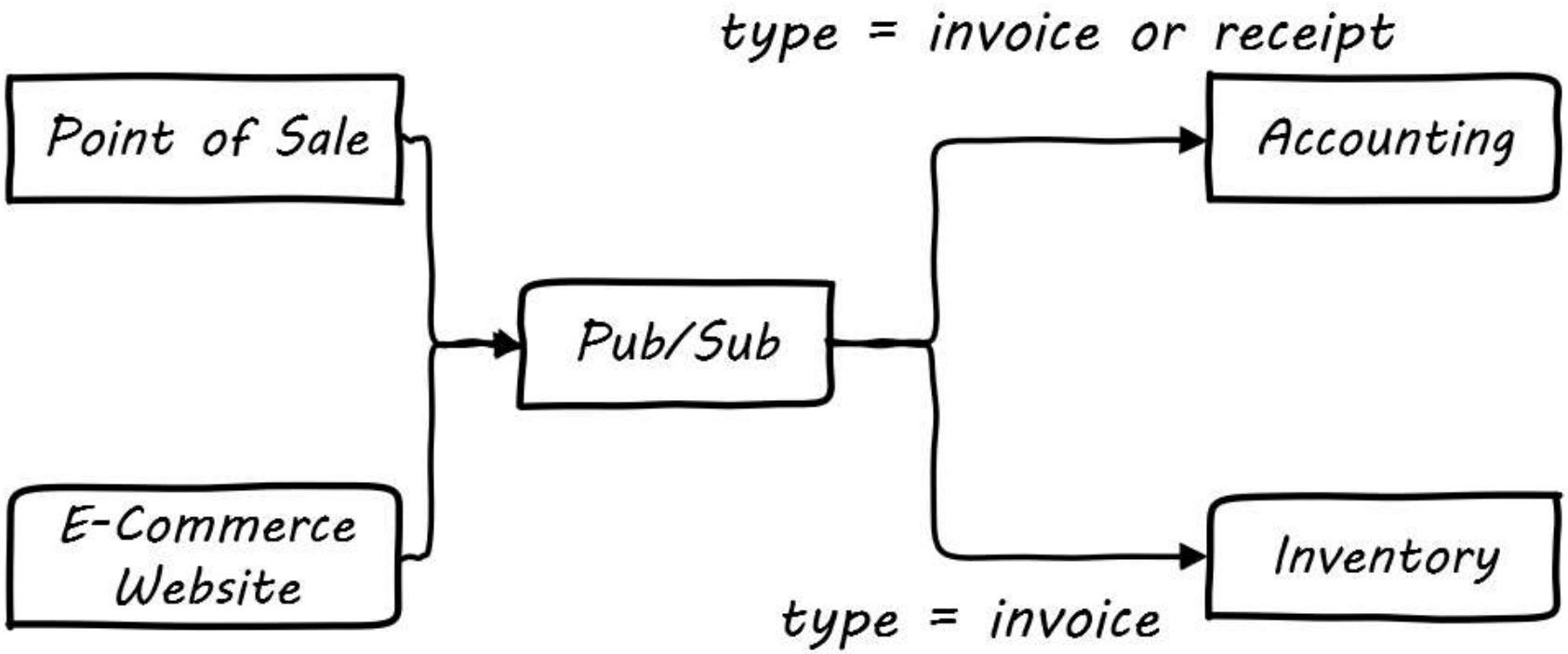
Pub/Sub: message filtration

- Often subscribers want just a specific subset of messages
  - Can simply drop irrelevant messages, but better filter ahead of time

- Two main forms of filtration:
  - Content-based
  - Topic-based

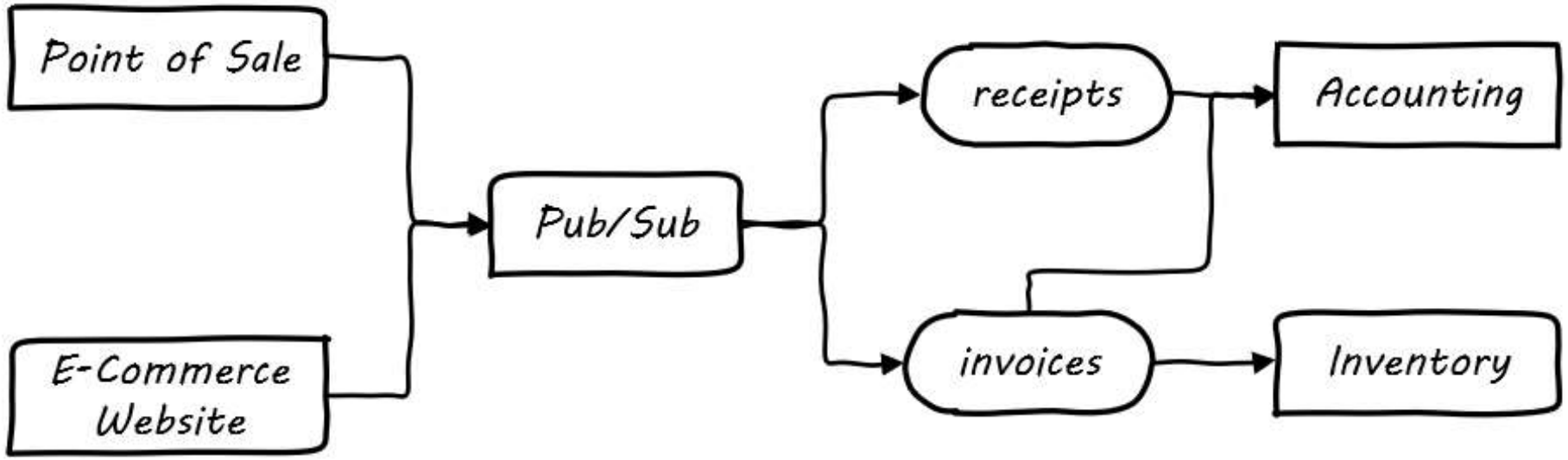
Pub/Sub: message filtration

- Content-based
  - Messages are filtered according to attributes defined by the subscriber
    - Headers, routing key etc.
  - Classification is done by the subscriber



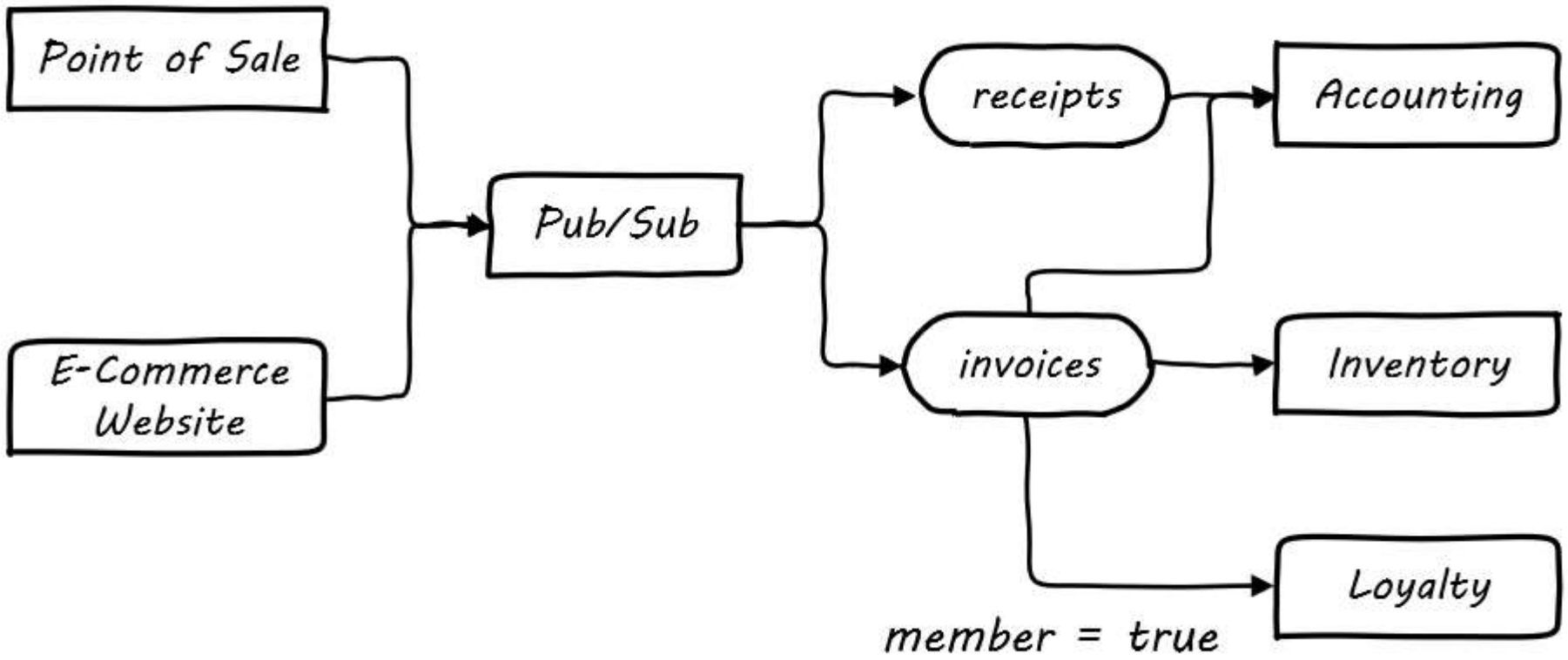
Pub/Sub: message filtration

- Topic-based
  - Publishers publish messages to “topics”
    - Named logical channels
  - Classification is done by the publisher



Pub/Sub: message filtration

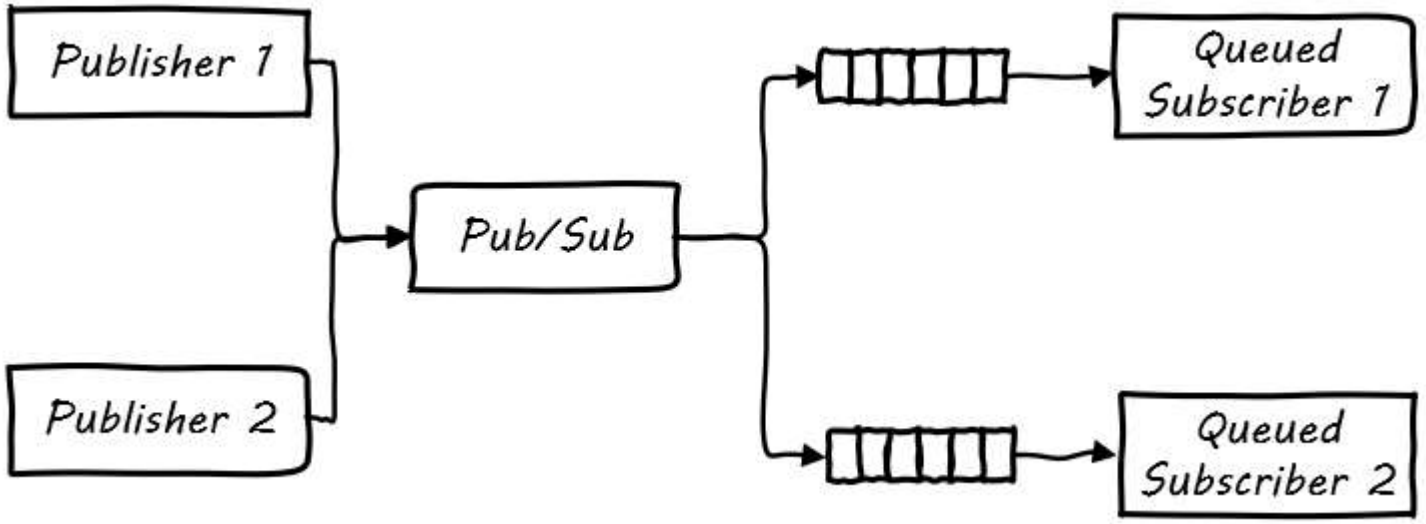
- A hybrid approach
  - Publishers publish to topics
  - Subscribers have content-based subscriptions to topics



Pub/Sub and Message Queues

- Pub/sub is symbiotic with message queues
  - Often packaged together

- Queued Subscribers
  - Messages are placed in a per-subscription queue

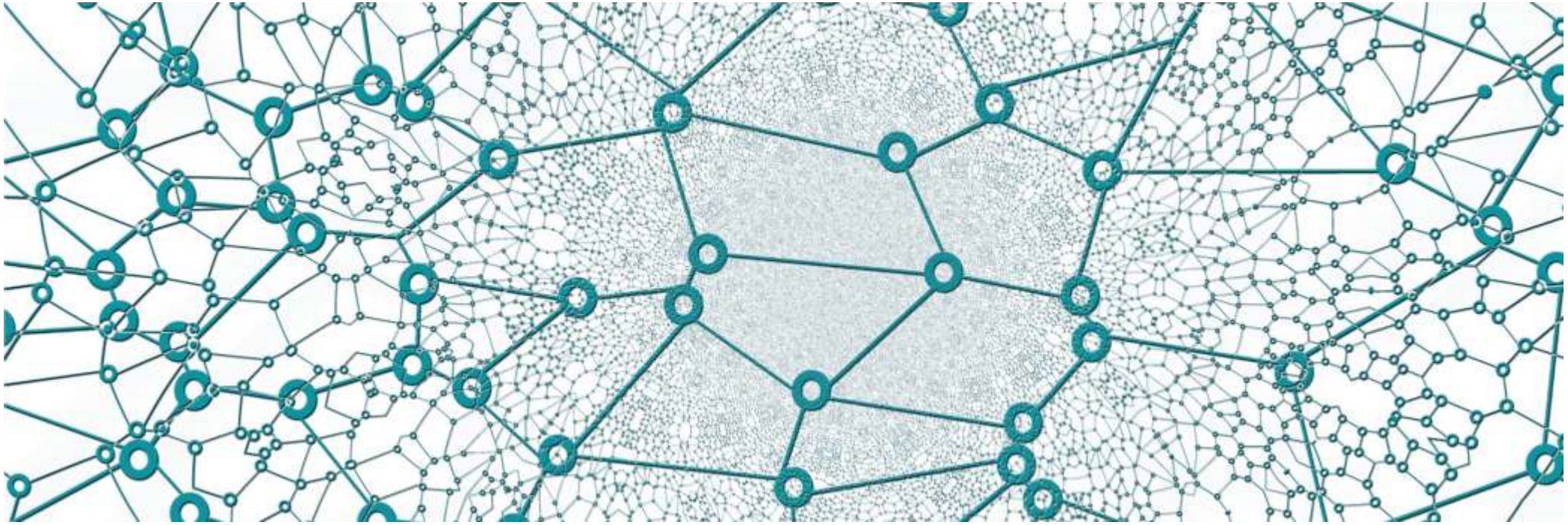


Technical Aspects

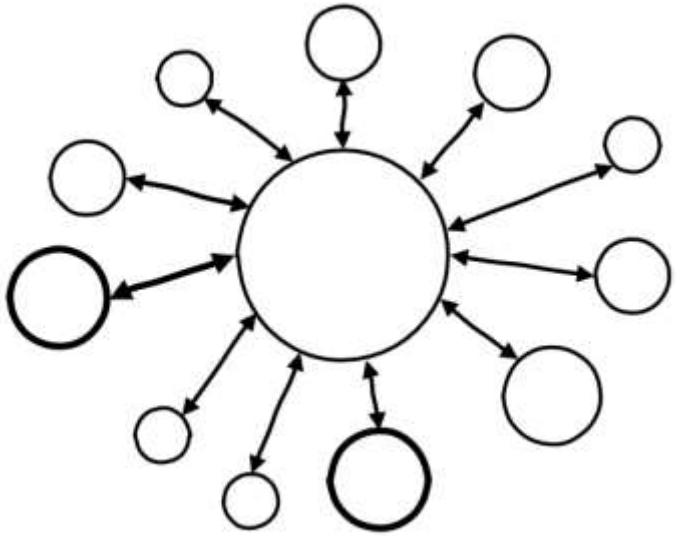
Down the rabbit hole

Message Queue Topology

- Two main topologies:
  - Broker
  - Federation

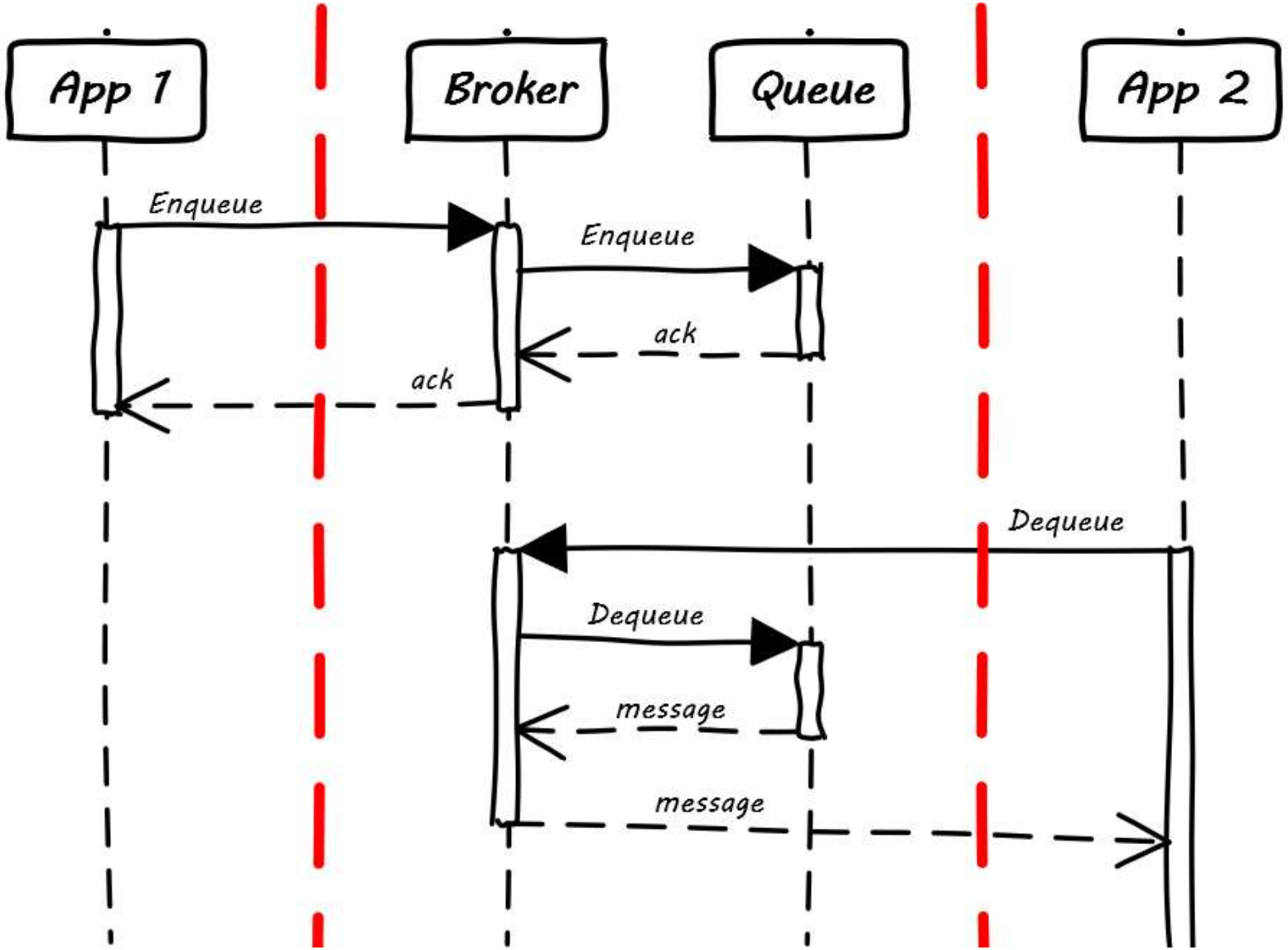


Broker-based topology



- Centralized architecture
- Broker service manages the queues
  - Mediates between producers, consumers and queues
  - Indirection
- Applications call the broker to enqueue/dequeue messages
  - Cannot operate when the broker is unavailable
  - Central point of failure
  - Can be clustered

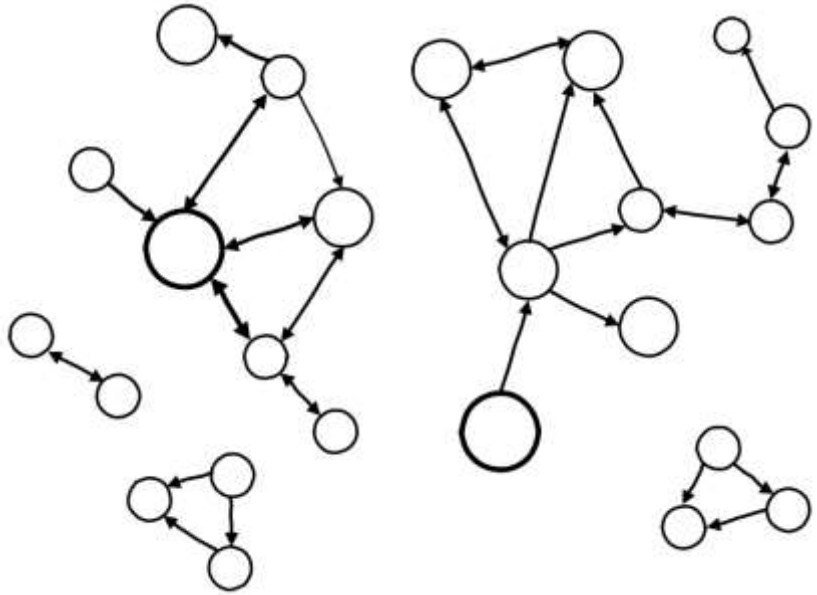
Broker-based topology



Federated topology

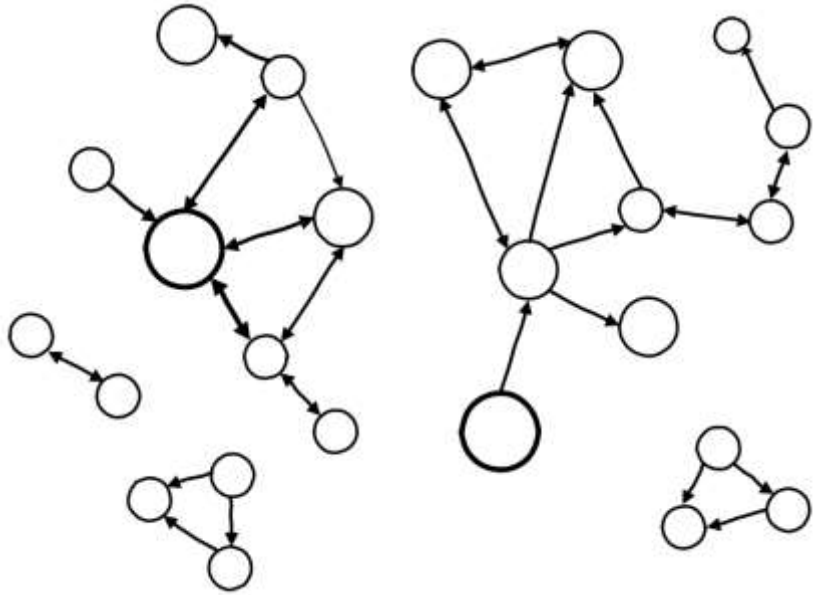
- Decentralized architecture

- Messages stored in local outgoing queue
  - Transferred to destination queue asynchronously
  - Store and forward
  - Queue always available to application

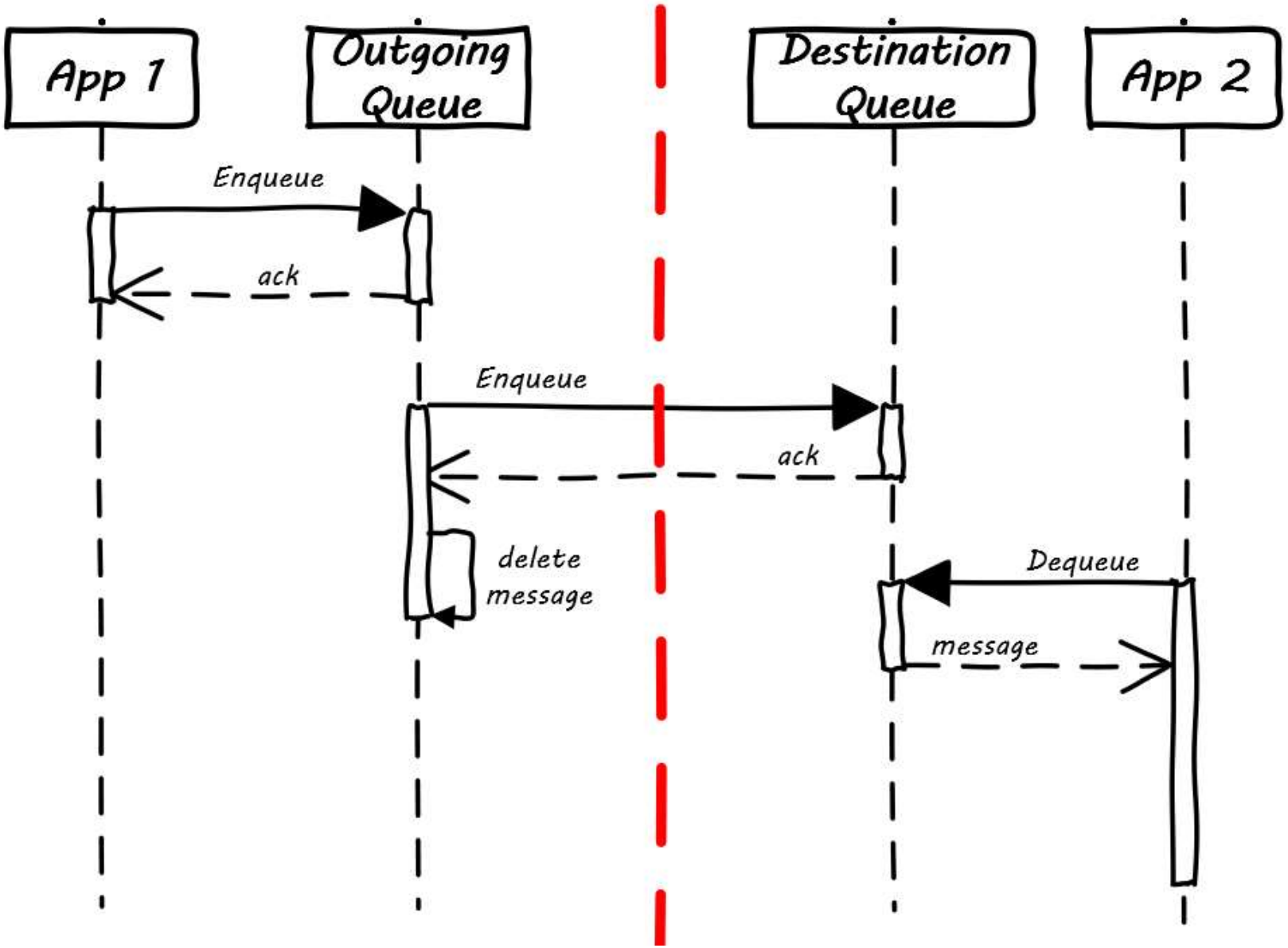


Federated topology

- Requires more intimate knowledge
  - Producers/consumers know location of queue
- How are identities federated?
- Harder to see the entire picture



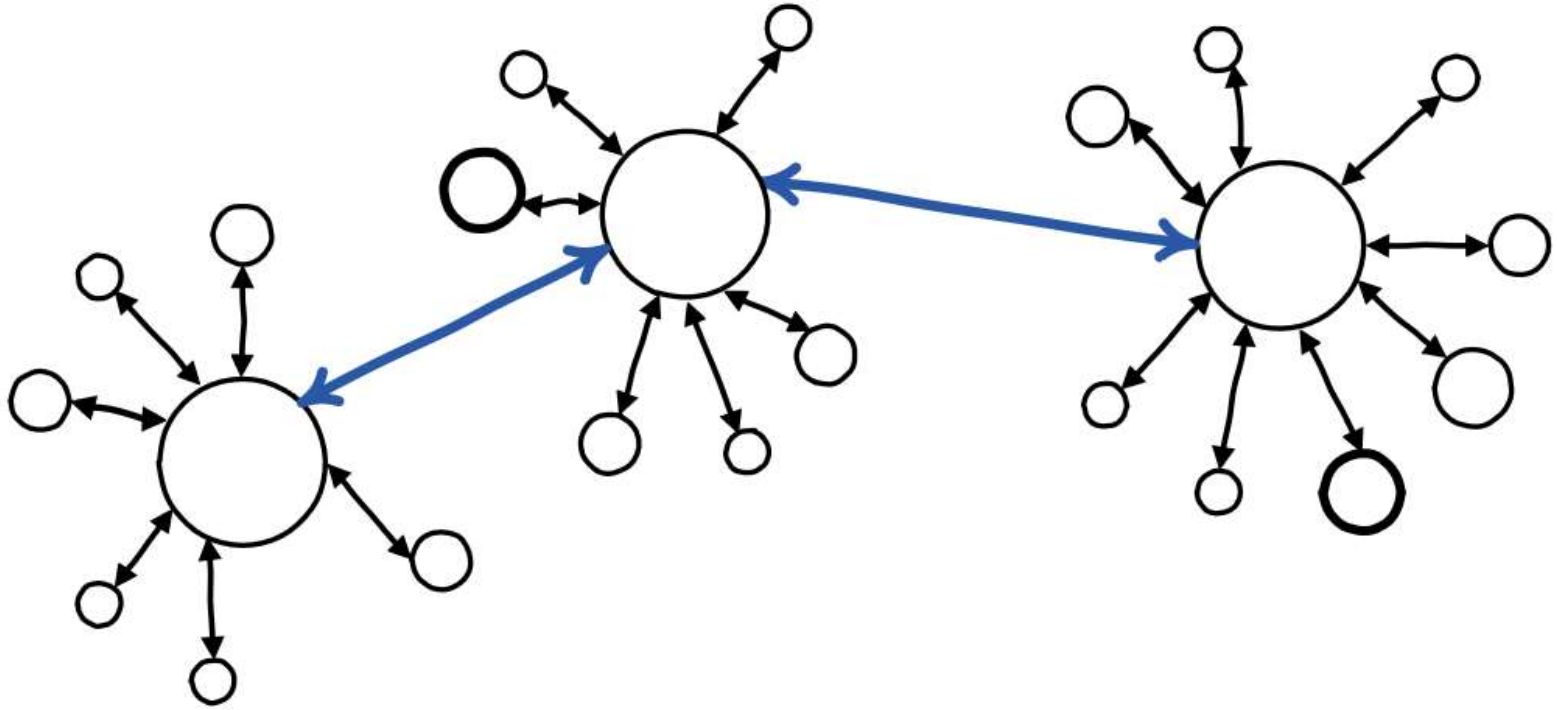
Federated topology



Combining Broker and Federation

- Broker and Federation are not mutually-exclusive

- Can have federated brokers
  - MQ feature
  - Custom-built



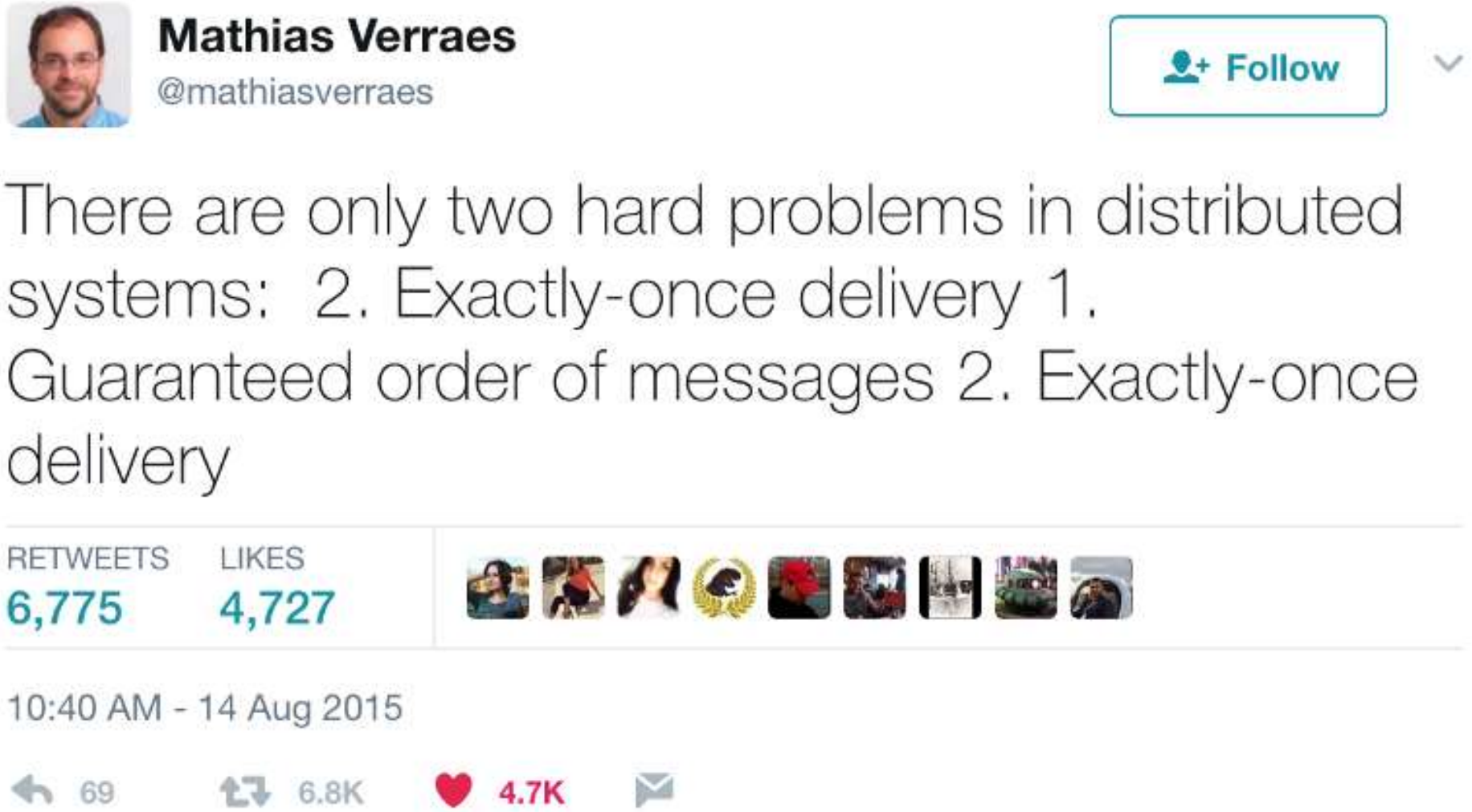
Consumption Methods

- Polling
  - Consumer calls queue
  - Optionally blocking until message arrives
- Push
  - Event based
  - Queue calls consumer
    - Hollywood principle

Consumption Methods

- Prefetch
  - Get a bulk of messages from queue
  - Chunky over chatty
  - Reduced latency
    - Consumption faster than delivery
  - Reduced overhead
    - Small payloads

Delivery & Order Guarantees



Delivery guarantees

- How many times will the message be delivered to a consumer?

- At most once
- At least once
- Exactly once

At most once delivery

- *a.k.a maybe once*

- The message is delivered once or not at all
  - Never twice
  - No attempt to deal with failures whatsoever
  - In-doubt transfers considered as completed

- Preferring message loss over duplication

- Suitable for events

Photo by Zeev Barkan
https://www.flickr.com/photos/zeevveez/4463827763

At least once delivery

- *a.k.a once or more*

- Message is redelivered upon failure
  - Until a consumer replies with ack or nack
  - In-doubt transfers considered to be failed
  - *In extremis may be delivered in parallel*

- Preferring message duplication over loss

- Suitable for documents

At least once delivery

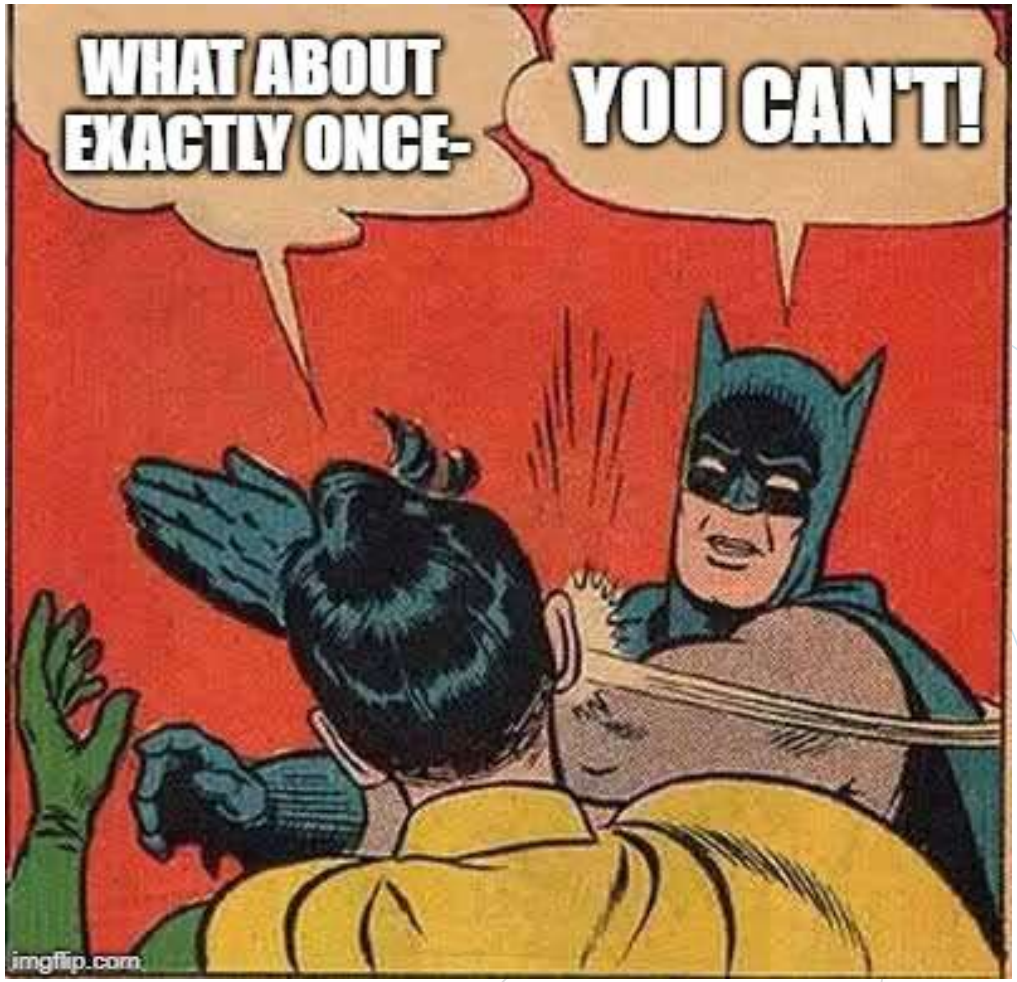
- How can we deal with multiple deliveries of the same message?

- Idempotence
  - Operation can be executed multiple times without issues
  - End result isn't changed
  - Implies rolling forward
  - Application-level responsibility

- Deduplication
  - Detect duplicate messages
  - Drop the duplicates to ensure single execution
  - Implies mutual exclusion
  - Application or infrastructure responsibility

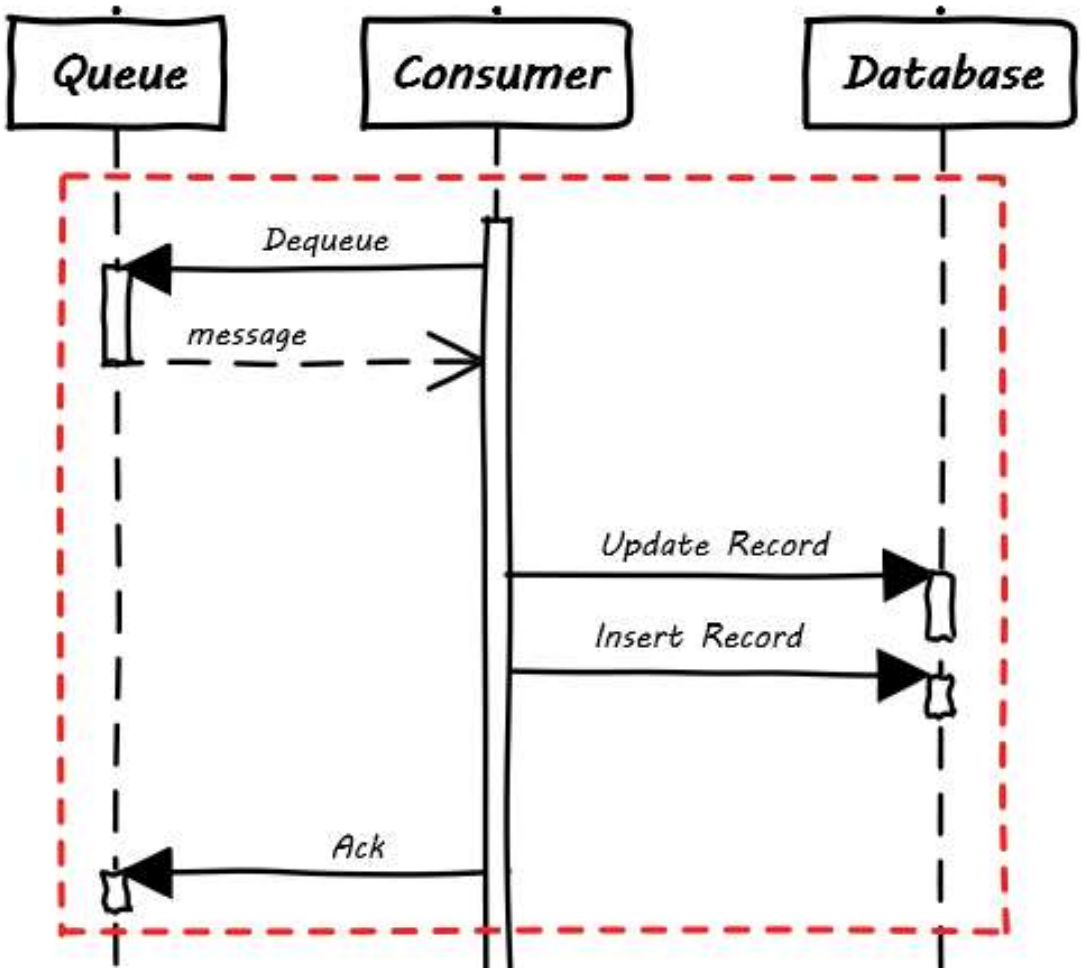
Exactly once delivery

- *a.k.a effectively once*
- Implies transactions
  - Severe performance penalty
- What is the transaction scope?



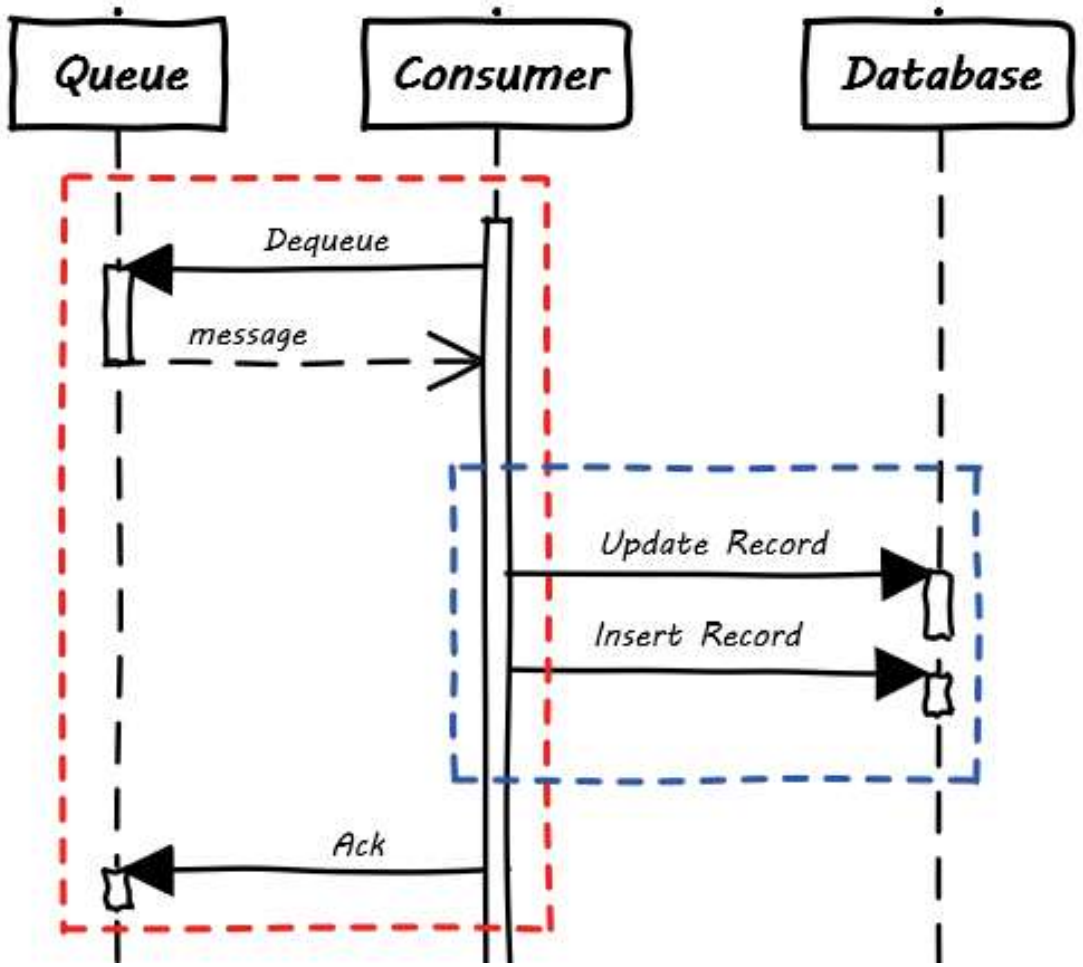
Exactly once delivery: transaction scope

- Queue + application transaction
  - Implies distributed transaction:
  - Rarely supported by MQs
  - Rarely supported by other services
    - Email, Storage, even databases
  - Even greater performance penalty
  - High complexity and nasty failure modes
    - Two-phase commit



Exactly once delivery: transaction scope

- Queue-only transaction
  - Rest of the procedure has separate fate
    - Commit/rollback independently
  - Pushes procedure to at least once or at most once
  - Useful for atomic operations against MQ
    - Combining enqueues and acks

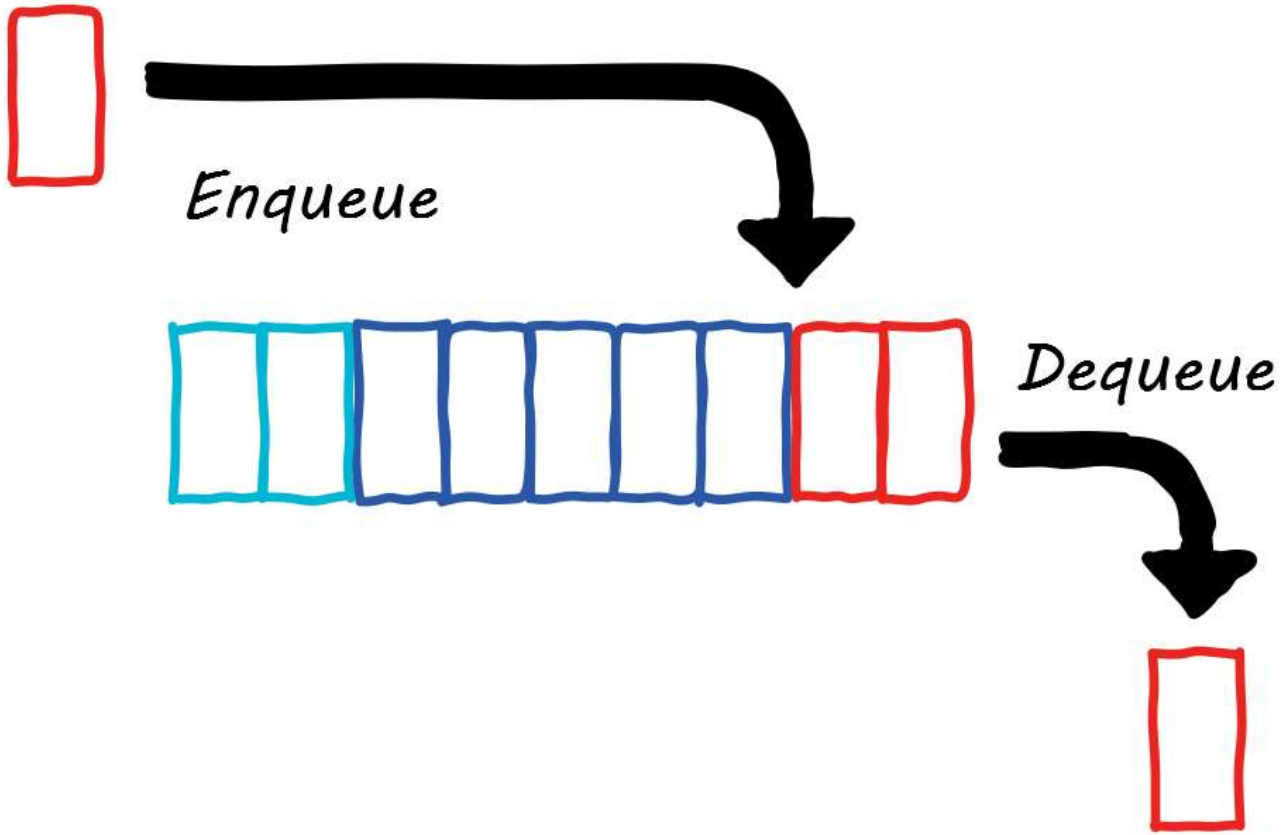


Order guarantees

- A queue is a FIFO data structure
  - First-in, first-out
  - *First come, first served*
- So messages in queue are ordered by arrival, *right?*
- Order is only *typically* guaranteed
  - Requeuing
  - Competing consumers
  - Routing (pub/sub)
  - Distributed systems issues

Priority Queues

- Messages with high priority are served first
  - Ordering occurs when inserting or when removing

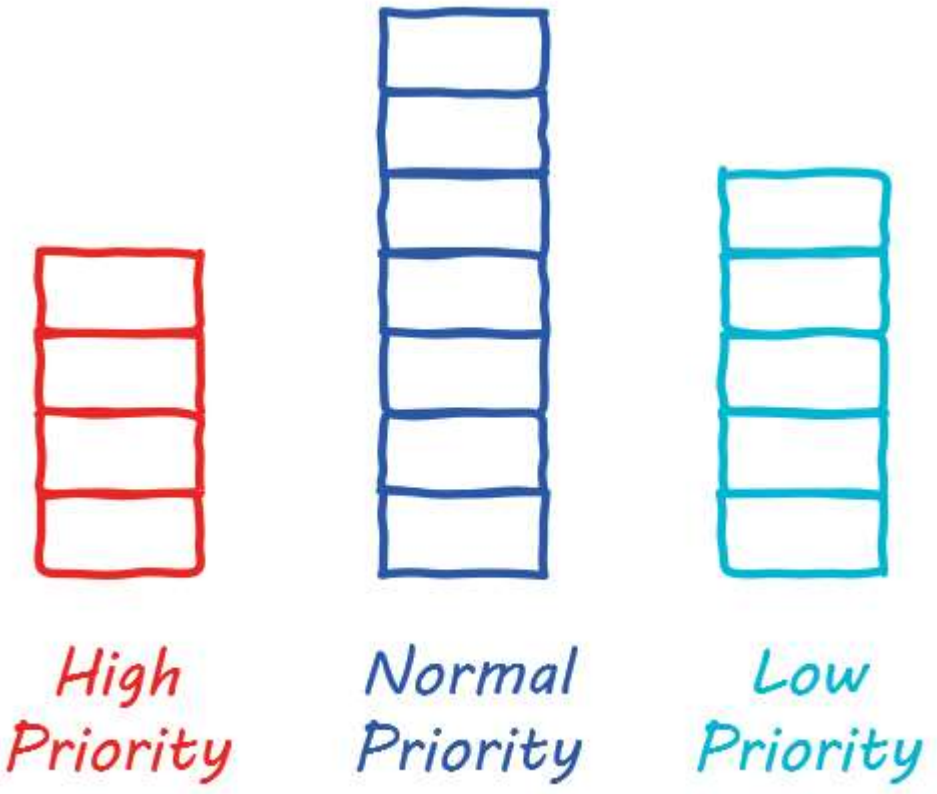


Priority Queues: caveats

- Performance penalty
  - Normally, insertion and removal are O(log n)
    - Sometimes insertion is O(1) and removal is O(n)
- Plan ahead
  - Must create the queue as priority queue
  - Priority levels predefined or declared during creation
- Ordering within same priority isn’t necessarily arrival order
  - Implementation specific

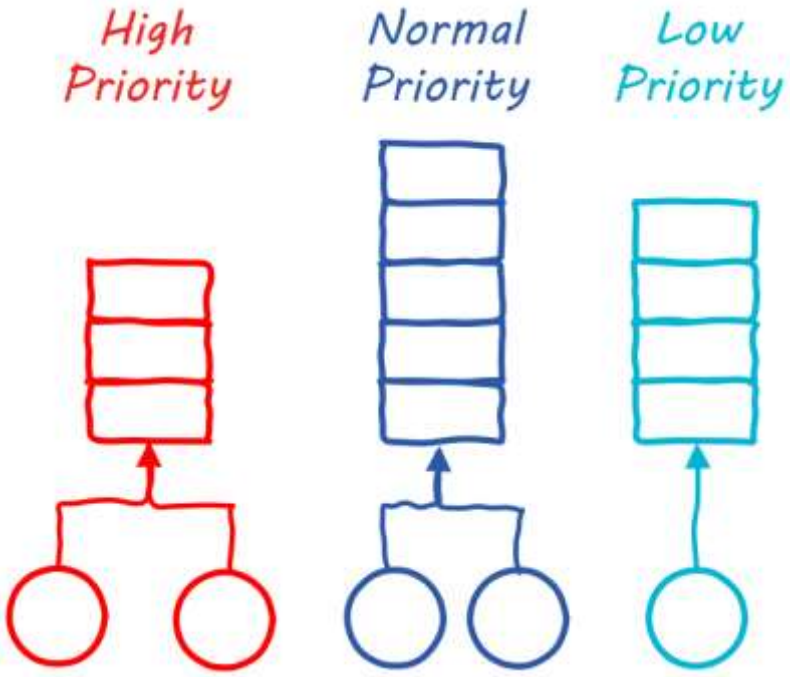
Priority Queues: alternative

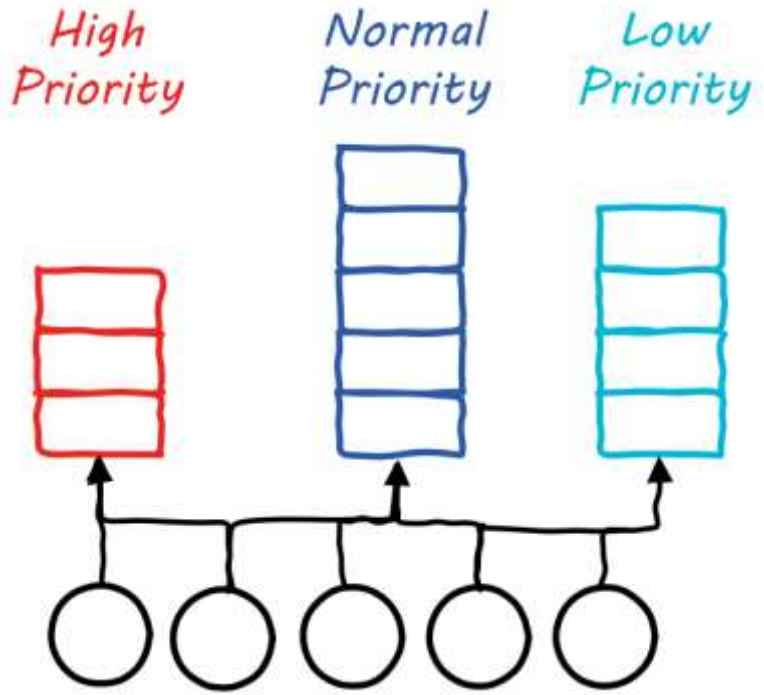
- Priority queues aren't common
  - Complex to implement
  - Inhibit other capabilities
- Queue per priority level
  - DIY prioritization
  - Recommended by cloud vendors



Priority Queues: alternative

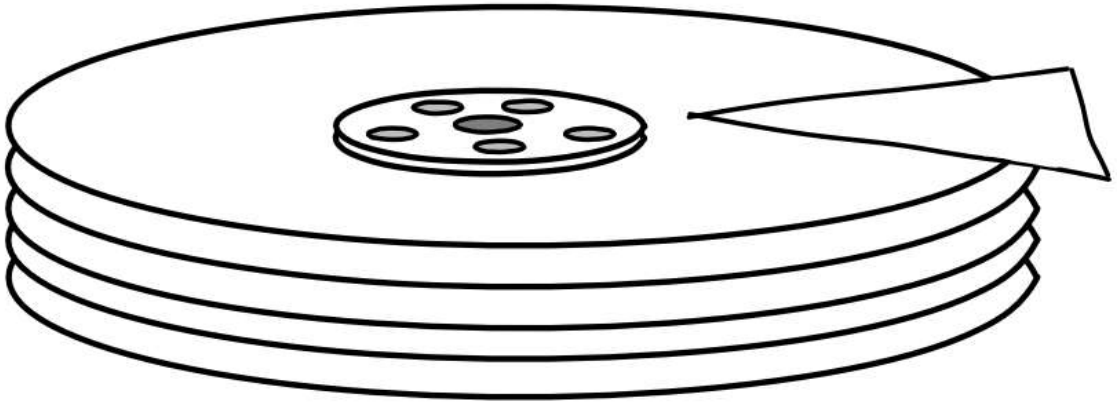
- Options for consumers:
  - Dedicated consumers per priority level
    - Under-utilization
  - Consumers pool which consumes by priority level
    - Risk of starvation





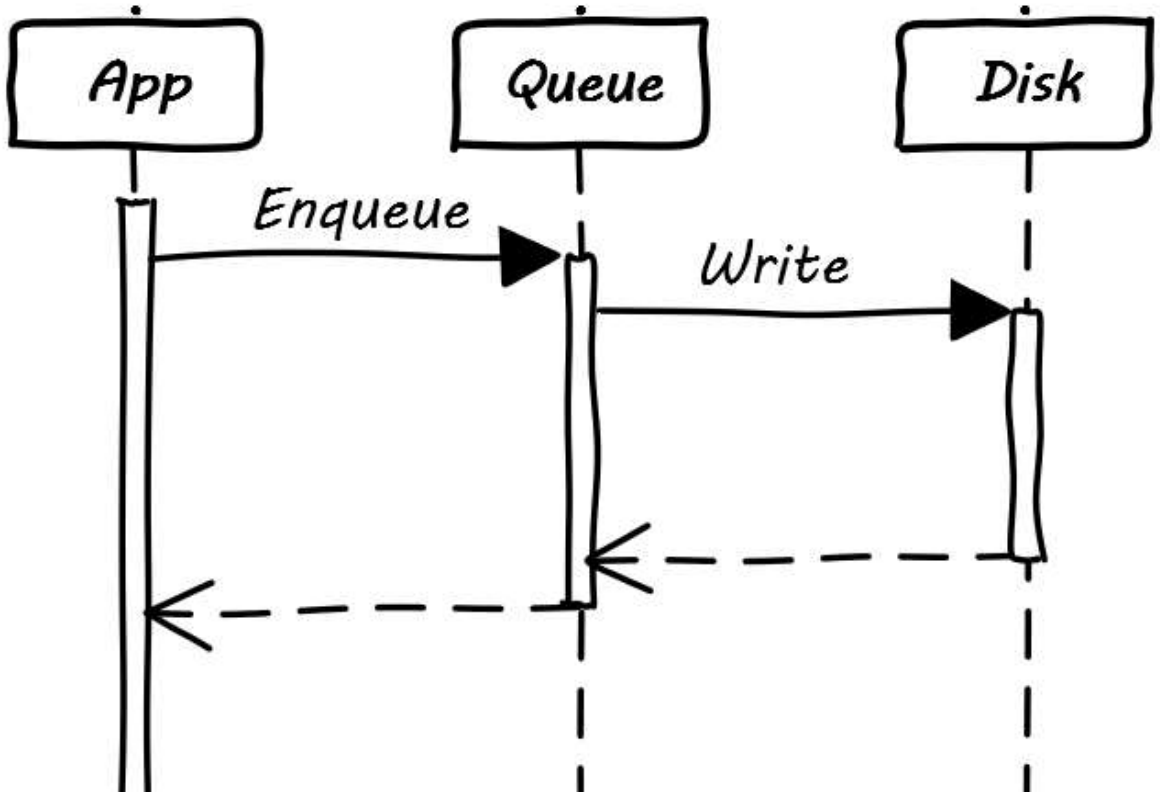
Persistence

- The message queue is persisted on disk
  - Durability
    - Surviving failures
    - Ultimately: crash-consistency
  - Bigger queues
    - Disk has more space than RAM



Persistence: the downside

- Negatively affects throughput
  - Orders of magnitude
- Disks are slow
  - High latency for random access
  - Milliseconds to `fsync`
  - RAM is ~100,000 times faster
- High availability complicates things
  - Need to persist to slaves as well

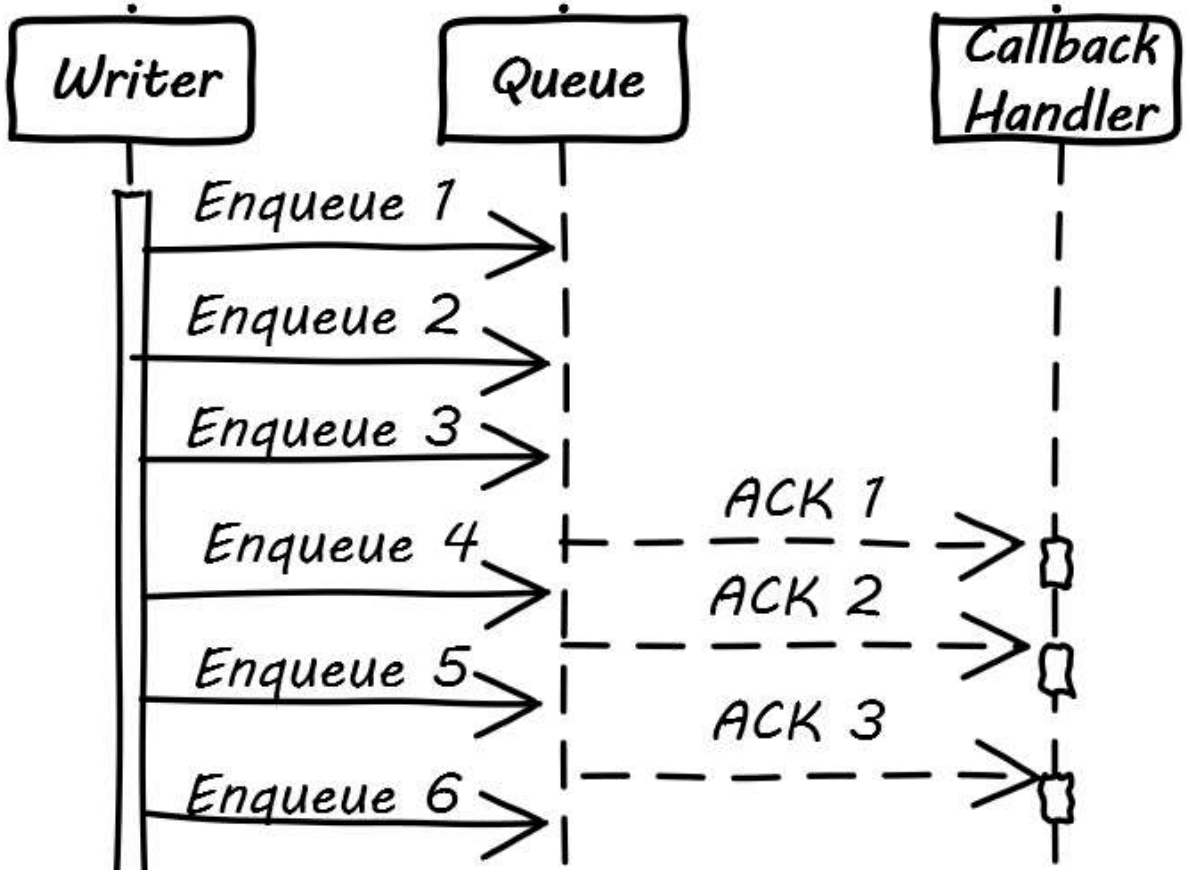


Persistence: the downside

- Accessing disk for each enqueue individually is slowest option
- Normally MQs buffer messages and write in batches
  - Will flush when idle or upon interval
  - May wait a few hundred milliseconds for ACK
    - even seconds

Persistence: the downside

- Can compensate by asynchronous delivery
  - Enqueue messages without awaiting the ACKs
  - Process ACKs asynchronously
  - Improves throughput
    - Doesn't solve latency



Persistence guarantees & tradeoffs

- Does ACK to producer means message was persisted?
- Does delivery to consumer means message was persisted?
- Depends on specific MQ guarantees and configuration
  - Tradeoffs
  - Read the fine print

Persistence guarantees & tradeoffs

- Example: Azure Service Bus

- Normal queues
  - Producer ACK'd only after persistence
  - Transfer to consumer only after persistence

- Express queues
  - Producer ACK'd immediately
  - Messages persisted only if not consumed after a few seconds
  - Ideally, message delivered to consumer without touching the disk
  - Risk of message loss & phantoms

Queue Size & Length Limits

- Resources are finite
  - Memory (RAM)
  - Storage (Disk)
- Resources are shared
  - Multiple queues rely on the same resources
- Specific queue may hog resources
- Can mitigate using per-queue size & length limits

Queue Size & Length Limits

- What happens when the queue is full?
- Drop messages in front of queue
  - Oldest messages are the victims
- Reject incoming messages
  - Newest messages are the victims
- Wait, what about message priorities?!
  - Priorities not always accounted for when queue is full
  - Read the fine print

Message Expiration (Time-To-Live)

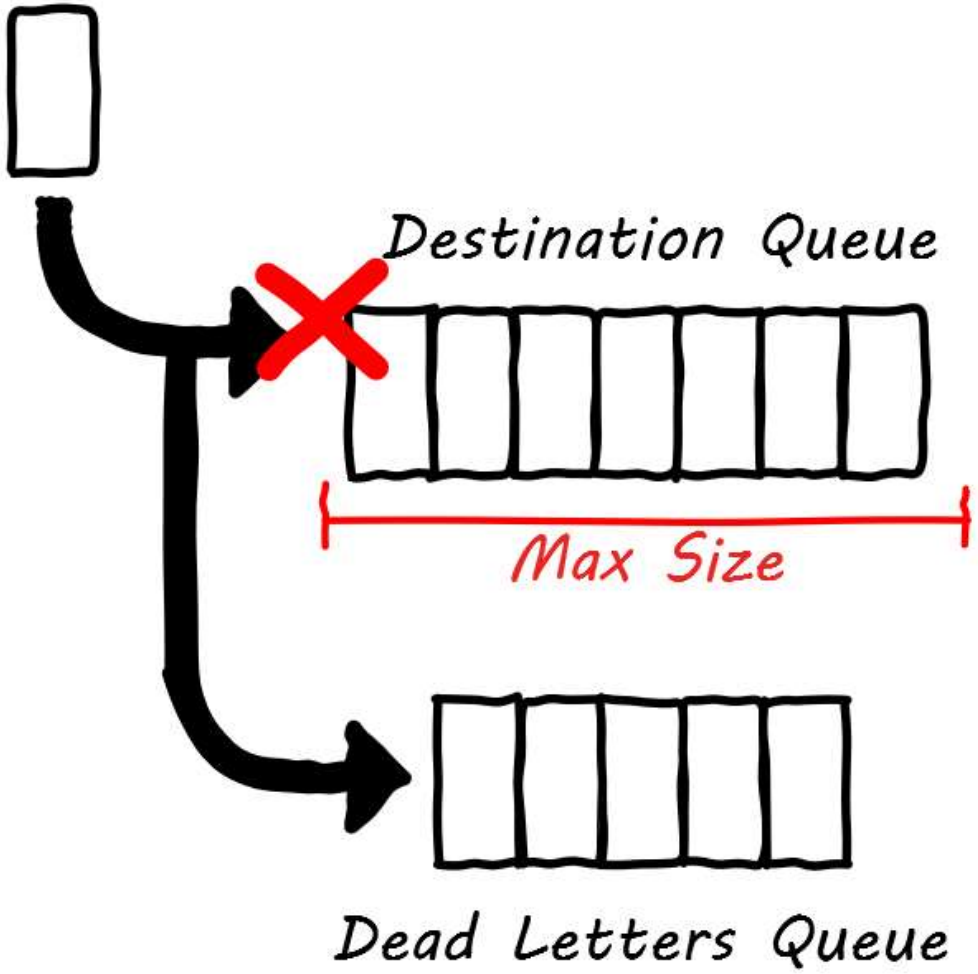
- Can set per-message Time-To-Live (TTL)
- Expired messages will not be delivered to consumers
- Very useful for events
  - Also periodic documents (i.e. state reports)

Message Expiration (Time-To-Live)

- When are resources freed?
  - Automatically?
  - When reaching end of queue?
  - Interaction with queue limits
- Can message expire after delivery?
  - Interaction with requeue feature
  - Eventual inconsistency

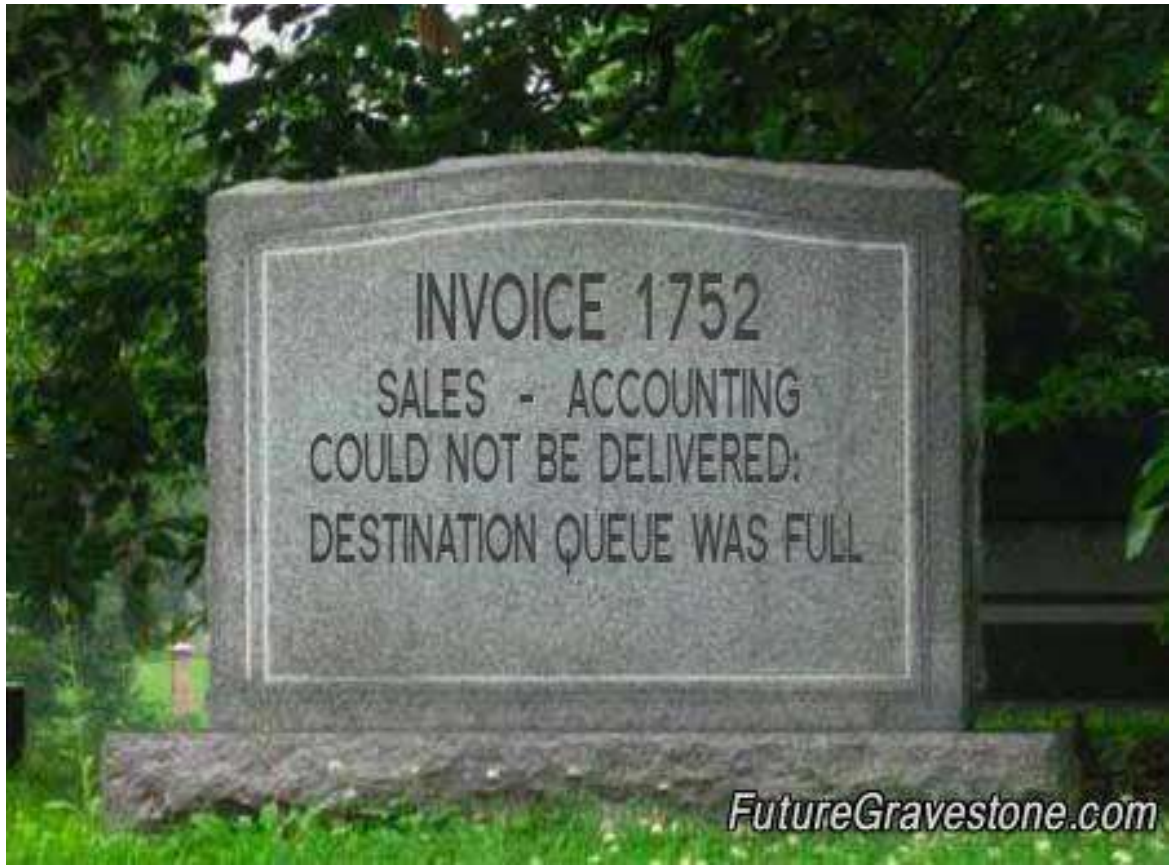
Dead Letters Queue (DLQ)

- A queue which holds dead messages
  - Those that could not be delivered
    - Including cause of death
- To be used by application, administrator, or developer
  - Examine & diagnose
  - Compensating action
    - Redelivery
    - Alerting
    - Logging
    - Discarding



Dead Letters Queue (DLQ)

- Common reasons for dead-lettering:
  - Message sent to non-existing queue
  - Queue length/size limit exceeded
  - Message size limit exceeded
  - Message rejected by consumer



Poison Messages

- Poison message = continuously fail
  - Poison the system with futile retries

- Common causes
  - Consumer bugs
  - Dependent services unreachable or unavailable

- The question isn't "will I have poison messages?"
  - It is "what am I going to do about them?"

Poison Messages

- Some MQs have built-in mechanisms for handling poison messages

- Bounded retries
  - Max amount of retries
  - Mean time between retries

- Handling behavior
  - Drop message
  - Move message to poison messages queue
  - Fault the queue

Clustering & High availability

- Multiple service instances
  - Load balancing
  - Redundancy

- High availability
  - Mirrored queues
    - Replicated between machines
  - Queue available despite machine failure

Clustering & High availability

- There are always tradeoffs
  - Some cast into design, some configurable
  - CAP theorem holds
- Is replication synchronous?
  - Does “ack” means message was persisted by all mirrors?
- How does the system handle network partitions?
  - Split brain scenario

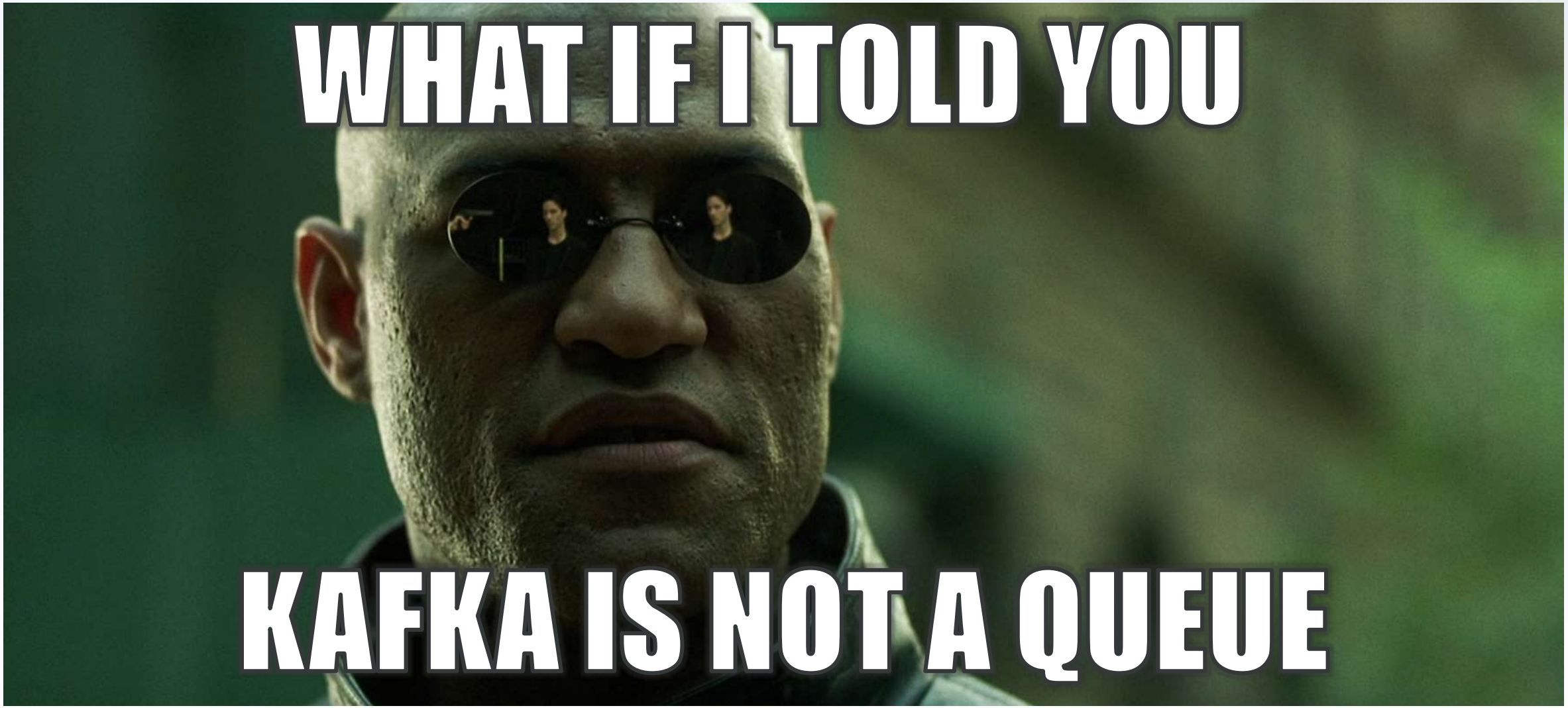
Message Streaming Platforms


“Check out Kafka,
there is a *huge buzz* around it”

“Apache Kafka is publish-subscribe messaging rethought as a distributed commit log”

-Apache Kafka website, circa 2016


2
Tomic G-MUFF
פה חשדתי.


WHAT IF I TOLD YOU
KAFKA IS NOT A QUEUE

Not a Queue

- No queue semantics
  - Can’t enqueue/dequeue/requeue/ack/nack
- Not FIFO enforcement
- Application manages most consumption state
- No dead-letters / poison message handling
- No competing consumers

Then what is it?

- 2013:
  - “Apache Kafka is a distributed publish-subscribe messaging system”
- 2014:
  - “Apache Kafka is publish-subscribe messaging rethought as a distributed commit log”
- 2017:
  - “Apache Kafka® is *a* ***distributed streaming platform”***

Then what is it?

- Same concept was adopted by other systems

- Microsoft Azure Event Hub
  - "[…] is a ***Big Data streaming platform*** and event ingestion service"

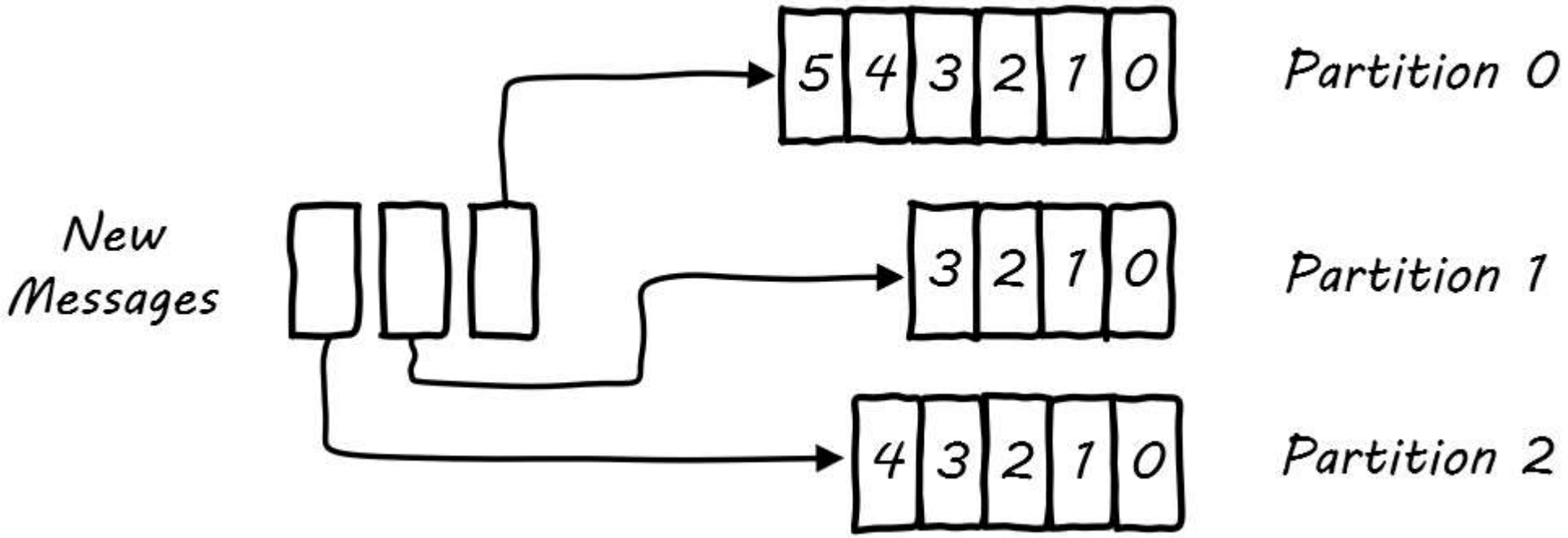
- Amazon AWS Kinesis
  - "[…] is a massively scalable and durable real-time ***data streaming service***"

How it works

- Core abstraction is the *topic*
  - Persistent streams of messages
  - *Category*
  - *Feed*
- Messages published to topic are *appended*
- Message kept for configurable retention period
  - Topic eventually *truncated*, old messages *discarded*

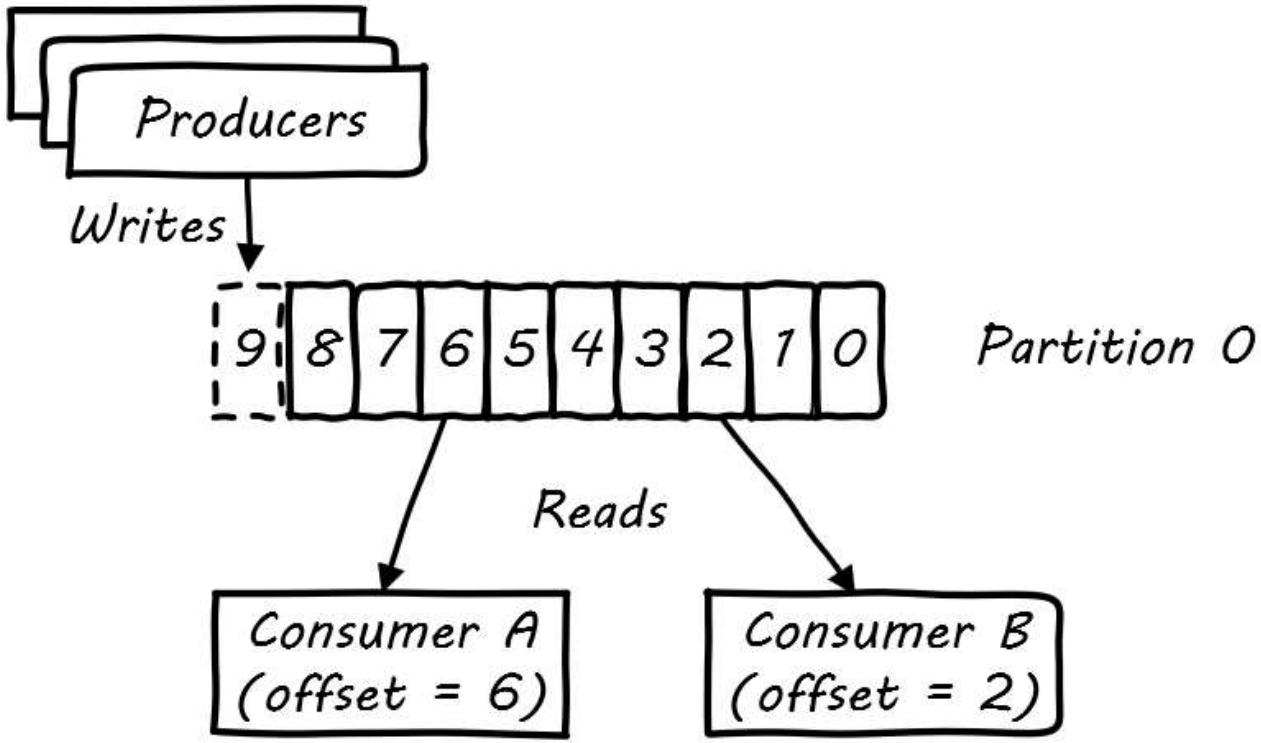
How it works

- Topic has one or more *partitions*
  - Each message is routed to single partition
  - Messages have *offset* inside partition
- Partitions can reside on different machines



How it works

- Consumers can read messages from the partition
- Each has its own offset within partition
  - “cursor” / “pointer”
- Consumer can change its offset
  - Back, forward, jump



How it works

- Each consumer belongs to one consumer group
  - Logical subscribers
  - Physical subscribers redundancy
- Only one consumer in group consumes from a specific partition
  - No competing consumers

Main Advantages

- Higher throughput
  - By order of magnitude
- Scalability & distribution
- Consumers have more control
  - Replay messages / *time travel*
  - Skip messages
- Stronger ordering guarantees
  - Messages **within partition** are ordered

Queues vs. Streaming Platforms

Queues vs. Streaming Platforms

- Streaming platforms and queues have many shared use cases
- In some cases the choice is trivial
  - One technological concept is just better suited for the case
  - *Stream processing*
  - *Big data ingress*
  - *Order processing*
  - *Financial transactions*
- In others there is no obvious choice
  - Tradeoffs

What this all boils down to

“for a business application, the real question is not `when to use queueing` ”

-Juval Löwy

Key takeaways

- Basic tools in the architect's toolbox
  - Message queues
  - Pub/sub

- The devil is in the details
  - Read the documentation
  - Validate your assumptions
  - Discover and choose the tradeoffs

Thank you!