latex input: mmd-article-header Title: Messaging Notes Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: distributed computing, messaging CSS: http://fletcherpenney.net/css/document.css xhtml header: Copyright: 2016 Ethan Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

# MOM -- Message Oriented Middleware

- Using **asynchronous message passing** instead of *request-response* architectures allow us to achieve greater **decoupling** between our program *components*
- "Message Brokers are usually application stacks with dedicated pieces covering the each stage of the exchange setup. From accepting a message to queuing it and delivering it to the requesting party, brokers handle the duty which would normally be much more cumbersome with non-dedicated solutions or simple hacks such as using a database, cron jobs, etc. They simply work by dealing with queues which technically constitute infinite buffers, to put messages and pop-and-deliver them later on to be processed either automatically or by polling." -- Digital Ocean
- To achieve the highest level of decoupling, we need an independent "receive" queue to buffer messages sent when the receiver or the network is down (e.g. Kafka and RabbitMQ)
  - If this is going to get really big (e.g. because the message volume is high and the receiver is unavailable for a long time) then it should spill to disk when it gets too big to efficiently hold in memory
- To achieve the highest speeds (but increased coupling), we should send *directly* to the reciever (e.g. ZeroMQ can even do this via a UNIX-style "named pipe" instead of using the TCP stack infrastructure)
- Some MOMs provide an API to internal routing logic so that the MOM itself decides which receiver queue to place an incoming message on, instead of the sender deciding

## Feature Set of MOMs

- **Durability** -- whether buffered messages can be written to disk or DBMS so that they survive crashes
- **Subscription Models**
  - **Point-to-Point** -- one sender, one receiver
  - **Publish-Subscribe** -- senders send to a "topic", and receivers *subscribe* to new messages posted to that topic
- **Push or Pull** -- whether consumers pull messages destined for them, or the broker sends the message through an open connection when it receives it from the sender

- **Message acknowledgement** -- this helps with reliability of message delivery
- **Delivery policies** -- e.g. at-*least*-once, at-*most*-once, *exactly* once
- **Purging policies** -- when does the broker delete messages; could be
    - after they are consumed/ACKd,
    - once the queue gets to a certain size
    - after a certain amount of time
- Supported **message size, format, and throughput** -- they may support XML or JSON or some binary thing, up to 20MB, up to 100K/sec etc.
- **Routing policies** -- whether the broker itself can filter messages for particular consumers so that the consumer only receives the exact subset of messages it is interested. Of course this logic can always be done by the producer and consumer themselves as well, but that might be more headache and lower performance.
- **Message ordering** -- e.g. FIFO, or per-partition FIFO (e.g. Kafka), or no guarantees
- **Batching policies** -- e.g. wait to send messages in bulk until time limit, size limit, or message count
- **Scalability** -- if I just throw more machines at the system, how can I expect system performance to change?
- **Availability** -- what parts of the system will still be functional in cases of different sorts of (e.g. network or hardware) failures
    - E.g. if the lead Kafka broker for a partition fails, another broker will become leader and take over
        - in general, though it can be configured that this new lead broker *must* be in the In-Sync Replicas set or whatever
- **Operational and financial overhead** -- how many compute/memory resources do we need, how long do we need to spend configuring, maintaining, reconfiguring over time, etc.
- **Cross platform interoperability** -- e.g. runs on Windows (why?)

### Alternative: Use a database as a queue

- Put messages into the database to "send"
- Get/poll (possibly also delete/update) messages from the database to "receive"
- This is **not** what databases were designed to do efficiently
    - So it's not going to *scale* well
- All in all, you could probably bend this solution to have whatever of the above attributes you want, but again, that's not what typical modern databases were designed to do

## AMQP -- Advanced Message Queuing Protocol

- AMQP is a binary, application layer protocol, designed to efficiently support a wide variety of messaging applications and communication patterns
- It does *not* define a standard API; it *only* defines the the wire protocol
- Before AMQP, it was [even] more difficult to build applications that combine platforms, protocols, and patterns (e.g. using JMS, which only has good support for Java)
- It provides flow controlled, message-oriented communication with various optional message- delivery guarantees
- It assumes an underlying reliable transport layer protocol such as TCP
- AMQP was originated in 2003 by John O'Hara at JPMorgan Chase in London, UK
    - The working group grew to 23 companies by 2011
- Transfers are subject to a credit based flow control scheme, managed using "`flow`" frames. This allows a process to protect itself from being overwhelmed by too large a volume of messages
- Each transferred message must eventually be settled. Settlement ensures that the sender and receiver agree on the state of the transfer, providing reliability guarantees
- Changes in state and settlement for a transfer (or set of transfers) are communicated between the peers using the disposition frame
- The various standard reliability guarantees e.g. "(at-(least|most)|*exactly*) once" can be enforced this way
- A session is a bidirectional, sequential conversation between two peers that is initiated with a begin frame and terminated with an end frame
- A connection between two peers can have multiple sessions multiplexed over it, each logically independent
- Connections are initiated with an open frame in which the sending peer's capabilities are expressed, and terminated with a close frame
- Message headers may include time to live, durability, and priority info
- Messages may include a `Map` of **attributes** (meta-data) that the broker or consumer can use however you want
- Messages received by the consumer *may* be ACKed so that the broker knows that it can be removed from the queue
- You can decide what to do when a broker is not able to deliver a message to its intended recipients, e.g.
    - Drop it entirely
    - Place the message in a special "*dead letters queue*" for consumption by a separate consumer
    - Return it to the publisher

## Basic architectural components

- **Publisher/producer** -- publishes messages to *brokers*
- **Broker** -- contains an *exchange* and the *queues*

- **Exchange** -- decides how to put messages received into *queues* using *bindings*
    - **Types**
        - Direct
        - Fanout
        - Topic
        - Headers/match
    - **Attributes**
        - Name
        - Durability (*durable* or *transient*) -- whether msgs survive broker restart
        - Auto-delete -- whether msgs are deleted after delivered as-intended
        - Arguments -- other attributes, specific to the exchange used
- **Binding** -- a rule about how messages should be added to a queue
- **Queue** -- a buffer from which consumers consume
- **Consumer** -- may *either* **pull *or* get pushed** messages from *queues*

## RabbitMQ

- When using **auto-acknowledgement**, the consumer ACKs messages it receives, which tells the broker that it has the to delete them from the queue

### RabbitMQ vs Kafka

- Use Rabbit if you have messages (20k+/sec) that need to be routed in complex ways to consumers, you want per-message delivery guarantees, you don't care about ordered delivery, and/or you need HA at the cluster-node level now
- RabbitMQ is broker-centric, focused around delivery guarantees between producers and consumers, with transient preferred over durable messages
- RabbitMQ uses the broker itself to maintain state of what's consumed (via message acknowledgements) - it uses Erlang's Mnesia to maintain delivery state around the broker cluster
- Unlike Kafka, RabbitMQ presumes that consumers are mostly online, and any messages "in wait" (persistent or not) are held opaquely (i.e. no cursor)
- RabbitMQ pre-2.0 (2010) would even fall over if your consumers were too slow, but now it's robust for online and batch consumers - but clearly large amounts of persistent messages sitting in the broker was not the main design case for AMQP in general
- The AMQP 0.9.1 model says "one producer channel, one exchange, one queue, one consumer channel" is required for in-order delivery
- The whole job of Kafka is to provide the "shock absorber" between the flood of events and those who want to consume them in their own way

- Kafka currently blows away RabbitMQ in terms of performance on synthetic benchmarks
- Kafka is an early Apache incubator project
- It doesn't necessarily have all the hard-learned aspects in RabbitMQ
- The AMQP standard is 'a mess'
- RabbitMQ offers optional durability and persistence (two separate features)
- The persistence uses an optimized logging format that provides the fastest possible write throughput (similar to things that Redis and Riak are doing)
- You never have to touch the persistent message store, and in the majority of cases, RabbitMQ will never read the persistent store
    - It is just there for insurance in the case of server crash or a low memory situation
- Kafka's "partitions" feature allows you to dynamically adjust the number of consumers concurrently reading a topic. Basically you just add a new consumer of a topic to an existing consumer group, and Kafka will "rebalance" automatically so that the responsibility of reading partiticular partitions within that topic is shared well amongst the consumers in the group. Of course this is only useful in those rare cases that you are building a system to dynamically deal with a rising message influx and overload. I'm not sure, but I kind of doubt RabbitMQ lets you handle that case so effectively.

## References

- Async Comm in SOA -- this is an excellent resource
- Wikipedia: Message broker
- Wikipedia: Message oriented middleware
- Wikipedia: AMQP
- Rabbit tutorial
- Quora: What are the differences between Apache Kafka and RabbitMQ?
- Digital Ocean: How to Install and Manage RabbitMQ