Building Scalable Enterprise Messaging Systems using AMQP

Rajith Attapattu
rajith@redhat.com
Bio

- Actively involved in several open source projects
- Karma on Apache Qpid, Axis2, Tuscany
- Contributed to Apache Synapse, Geronimo
- Involved in AMQP spec group
Agenda

- Introduction
- Look at user level problems in messaging
- And see how AMQP solves them
- Questions
User Level Problems

- Interoperability – Vendor Lock in
- Message Routing
- Broker Management
- Reliability
- Queue Browsing vs Consume
- Performance
- Scalability
- Transactions
- Hardware Monitoring
Introduction

- Advanced Message Queuing Protocol
- Open standard with royalty free use
- Strong focus on financial services industry
- AMQP Spec group (www.amqp.org)
Demo

- Simple demo to show interoperability
  - A client talking different broker implementations
  - A broker talking to different client implementations
- Involves a C++ & Java Broker
- Clients Java(JMS), C++, Python and Ruby
Interoperability

- What does it take to achieve this?
- All brokers need to behave the same way
- All clients need to behave the same way
- Use a standard for commands on the wire
- Use a language neutral type system
Interoperability

- AMQP solves this by defining
  - A network wire-level protocol
  - A defined set of messaging capabilities (AMQP model)
  - A type system
AMQP

- Broker semantics are defined explicitly
- Defines an explicit set of commands
- Grouped by class of functionality
- Commands modify state in a peer
Message Routing

- Pre AMQP models had several issues
  - Opaque routing models – not explicitly defined
  - Rigid monolithic routing engines
  - Cannot manipulate the RM using the protocol
AMQP Routing Model

- RM is explicitly defined, and that permits
- Management commands to manipulate RM
- Exchange, Queue and Binding
- Composed to define processing chains
- Similar to how Email works
AMQP Routing Model

Publisher

Exchange

Consumer

Virtual Host

Message Queues

Server
Exchange

- Accepts messages from producers
- Routes based on bindings to
- Queues or other exchanges
- Analogues to a Mail Transfer Agent
Message Queue

- Stores messages in memory or disk
- And delivers to consumers
- Can have the following properties
  - Durable/Temporary, Private/Shared, Auto-Delete
- Analogues to a Mailbox
Routing Key

- Special Field (Header) present in Message Delivery Properties
- It's a virtual address
- Analogues to 'To' or 'CC' in Email
- The exchange may use this to route
Binding

- Relationship btw Queue & Exchange
- Defines routing criteria
- Simple case, criteria == Routing Key
- Same queue can have multiple bindings
- Analogues to Routing Tables
Exchange Types

- Defines a set of standard exchanges
  - Direct
  - Topic
  - Fanout
  - Headers
Direct Exchange

- $R == \text{routing\_key supplied in publish}$
- $K == \text{routing\_key supplied in binding}$
- If $k == R$, the message is sent to that queue
- Can match more than one queue
- Common case, $\text{queue\_name} = k$
Topic Exchange

- \( R == \) routing_key supplied in publish
- \( K == \) routing_key supplied in binding
- If routing pattern \( k \) matches \( R \), (wild card)
- The message is sent to that queue
- Ex routing_key == stocks.us.RHT
Fanout Exchange

- Queues are bound with no criteria
- All messages are routed to every queue
Headers Exchange

- Routes based on headers
- Queues bind with arguments for matching
- The routing_key may not be used
Common Messaging Cases

- Point-to-point,
  - \texttt{routing\_key} == queue name
- Pub-Sub
  - \texttt{routing\_key} == topic hierarchy value
Extending the Routing Model

- Can create your own exchange type OR
- Routing criteria ..etc
- Within the general rules of the protocol
Broker Management

- Explicit commands to manipulate
  - Queues, Exchanges, Bindings
  - Ex. create, bind, delete queue/exchange
- Language neutral management API for free
- The ability to manipulate runtime
Reliability

- Most applications require messages to be
  - Delivered Exactly Once (no duplicates)
  - Guaranteed Delivery
- Let's look at the reliability use cases
Reliability Use Cases

- When a producer sends a message to broker
  - Needs confirmation that is accepts responsibility
  - Needs confirmation it's on the queue
- When a broker sends a message to client
  - Needs confirmation that is accepts responsibility
  - Needs confirmation it's consumed
Let's understand the problem

- Needs to understand the different between
- Transfer of data vs Transfer of responsibility
- And message processed vs seen.
- Understanding this is key for reliability
- AMQP makes this distinction very clearly
Transfer of Responsibility

- Before/After actions permitted and required
- From a sender/recipient is very different
- After responsibility is transferred
- AMQP defines two modes
  - no-acquire
  - pre-acquire
no-acquire mode

- Only data is transferred, not responsibility
- No exclusive access to process the message
- Another client may see, acquire and consume
- Need to explicitly acquire before processing
pre-acquire mode

- Both data and responsibility is transferred
- Exclusive access to process the message
- No other client can see the message
- Can release to relinquish responsibility
Processed vs Seen

- Peer may see the message, BUT it could
- Crash before processing the message.
- So Semantic acknowledgment is required.
- The Broker should only dequeue messages
- On receipt of the semantic Ack
Disabling the Semantic Ack

- The broker may want to dequeue
- Immediately upon sending the message
- It may not care if the client gets it or not
- The delivery of message cannot be undone
- \textit{i.e.} Client cannot reject or release message
- Ex StockTickers
Confirm Mode

- Defines a confirm mode to disable
- Semantic Akcs to optimize for certain Apps
- On – requires semantic ack for dequeue
- Off – dequeue immediately upon acquisition
Summary

- no-acquire, confirm-mode = on
  - Dequeue message only upon semantic ack
  - After message is explicitly acquired

- pre-acquire, confirm-mode = on
  - Dequeue message only upon semantic ack
Summary (Continued)

- no-acquire, confirm-mode = off
  - Dequeue message immediately
  - Upon explicit acquisition
- pre-acquire, confirm-mode = off
  - Dequeue message immediately upon sending
Queue Browsing vs Consume

- Applications may need to look at
- Messages in a non destructive way, *i.e.*
- Only to examine the messages on the queue
- Not to process them
- Ex Queue Browsing, Client Side Selectors
Queue Browsing vs Consume

- Only transfer of data is required.
- Transfer of responsibility is required
- Only if it decides to process the message.
- Can implement this with no-acquire mode
- And explicitly acquire if need to process
Performance Requirements

- Need to prefetch messages
- Need to throttle message flow
- Need for Asynchronous Processing
- Control the window of unacked commands
Message Prefetch Problem

- To improve performance, Clients may need
- To optimistically fetch messages.
- If the client stops message processing
- The client will end up having valid
- Messages that it cannot/will not process
Message Prefetch Solution

- If we subscribe in no-acquire mode, we
- Don't need to release explicitly.
- We will only acquire them message
- If the client wants to process it
Message Prefetch Solution

- If we subscribe in pre-acquire mode, we
- Simply release the message.
Release vs Reject Message

- **Release**
  - Relinquish responsibility for processing message
  - Message can be safely delivered to other clients

- **Reject**
  - Indicates a problem with processing a message
  - Will result in a DLQ of the message
Throttling Message Flow

- A peer may need to control the flow of Messages it receives from its partner.
- This prevents from the partner pushing more Messages than the peer actually needs.
Throttling Message Flow

- A Peer may also need to control the flow of Messages it sends to its partner.
- This prevents from sending more messages than the partner actually needs and may
- Control the window of unacked messages
Credit Based Solution

- Sender maintains credit balance with recipient
- Credit Balance consist of a
  - Message Count
  - Byte Count
- When a Message is sent
- Both counts are decremented
Credit Based (Continued)

- When either value is zero,
- No more messages are sent until,
- Further credit is received from peer.
- If byte count is insufficient,
- No partial messages can be sent.
Window Based Solution

- Identical to credit based, except
- Message acknowledgment implicitly grants
  - A single unit of message credit
  - And size of message in byte credits
- Controls the window of unacked messages.
Asynchronous Processing

- Application may need to create 50 queues
- Without waiting for confirmation, But
- Eventually would like to know the outcome.
- So we have a requirement for correlating
- Acks sent by the broker asynchronously.
Execution Layer

- The execution layer provides correlation of
  - Semantic Acknowledgments by exchanging
  - An execution mark (EM) between peers.
  - An abnormal termination of a command M,
  - Should be notified explicitly before EM \geq M.
Unacked Command Window

- For performance (and other reasons) a peer
- May want to control the unacked window by,
- Synchronization of state
  - Peer needs to block on completion of a command
- Requesting the partners current execution mark
  - So it can discard unwanted state
Synchronization of State

If I create 50 queues, eventually I want to know the outcome, so I can release resources that maybe locked until desired state is reached.

... Commands which might create memory issues if Client may also hold state related to unacked... It continues to grow and affect performance.
Solution based on Sync bit

- AMQP allows a peer to force its partner to
  Synchronize by sending its Execution Mark
  When the given command is executed.
- Ex. You can sync on the 50th Queue create.
Managing Unacked Window

- Assume that in a high volume pub/sub system
- The broker acks every 1000 messages unless
- Explicitly requested to do so.
- A particular publisher can only hold 250
- Unacked messages due to memory constraints.
- How would you get around this problem?
Possible Solutions

- Explicitly synchronizing state with the broker is
- Inefficient as you would block until the broker
- Process All 250 messages.
- This may result in a slow publishing rate.
- How about if you request a snapshot of
- Brokers current state (execution mark)?
Solution based on Flush bit

- If you know partners current execution mark,
- Can calculate the no of messages processed.
- You can then release state relating to those
- Messages and send more messages until you
- Reach 250 unacked messages again.
- AMQP provides a way to ask the current EM.
Scalability

- Transparent Failover via Session resume
- Failover Exchange provides updates about
- Cluster membership state.
Transactions

- Applications need to Enqueue and Dequeue,
- Messages within transaction boundaries.
- Supports local and distributed transactions.
- Only enqueue and dequeue are covered.
- Any other state change is not covered.
Transactions (Continued)

- If you create an exchange or bind a queue
- Inside a transaction, it will NOT be undone,
- If a rollback occurs.
Hardware Monitoring

- Hardware devices need to statelessly index
- Themselves into interesting points inside an
- AMQP payload at wire speeds to provide
- AMQP-aware hardware monitoring and QoS
Frame, Segment, Frameset

- AMQP defines the above concepts to
- Facilitate hardware monitoring and QoS.
- Frame is the smallest unit on the wire
- Segment is a collection of frames
- Frameset consists of one or more segments
Frame

- Consists of a header which is 12 bytes
- And a variable payload size
- Peers negotiate a max frame size
- Bits to indicate boundaries
- Space reserved for future extensions
Segment

- Smallest logical unit parsed by a peer
- Has a type, Ex Method, Header ..etc
- Bits identify first/last frame in segment
- And first/last segment in Frameset
- These bits are packed in to frame headers
Frameset

- Forms a logical unit for the model
- Marked independent of the model
- Intermediaries can do generic parsing
- Without explicit knowledge of the model
- Facilitates stateless parsing at wire speeds
Links