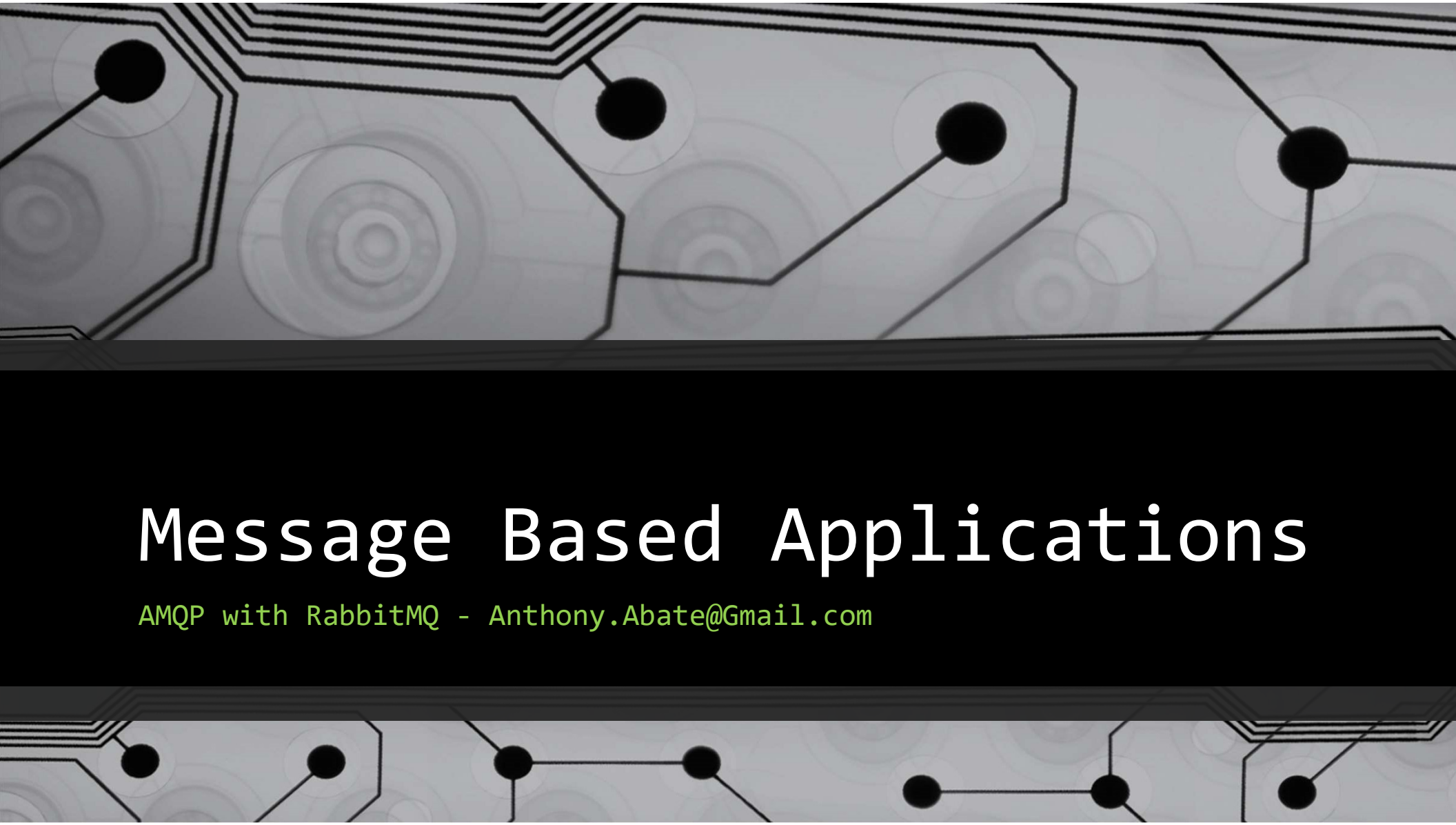


Message Based Applications

AMQP with RabbitMQ - Anthony.Abate@Gmail.com

CODE CAMP NYC

MARQUEE SPONSOR

pluralsight

hardcore dev and IT training

CODE CAMP NYC

PLATINUM SPONSOR

CODE CAMP NYC

PLATINUM SPONSOR

neuron esb

CODE CAMP NYC

PLATINUM SPONSOR

SURGE

CODE CAMP NYC

GOLD SPONSORS

twilio

QuickLearn
TRAINING

redgate
ingeniously simple tools

westMONROE

Overview

- Why Messaging?
- Why Advanced Message Queuing Protocol (AMQP)?
- Why RabbitMQ?
- AMQP Basics
  - Publishing / Exchanges
  - Subscribing / Queues
- Advanced Topics
  - Message Patterns
  - Error Handling
  - Anti-Patterns
  - Distributed Setups

Abridged Evolution of Application Architectures

- Single Stand Alone Application
  - One process / Multiple Threads
- 2 Tier (Application + Database)
  - Could span Multiple Process
  - Simple Distributed Applications
- 3+ Tier (Applications Layers + Services + Storage)
  - Many disconnected services
  - Many Interdependencies
  - Client resources: threads or tasks to make each call
- Cloud Based Applications
  - Better suited for messaging

Why Messaging?

- Loosely coupled architecture
  - The messaging infrastructure handles routing, duplication, and queuing
  - Allows for publishers to have zero knowledge about subscribers
  - Clients can subscribe to any event
- Scalable and Performance
  - Your application will become 'Event' based with callbacks from the Broker indicating a message is ready for processing
  - Different parts of the app can be tuned accordingly
- Transfer of Responsibility
  - Can 'Fire and Forget' - when the message has been transferred
  - Guarantees no lost message and eventual delivery (no guarantees on processing)

Messaging sounds useful, so Why AMQP?

- There are numerous MQ Technologies out there:
  - TIBCO, JMS, MQ-Series, MSMQ, Zero-MQ, etc
- Open standard application layer protocol for message-oriented middleware
  - Queuing
  - Routing
    - Point-to-Point
    - Publish-and-Subscribe
  - Reliability
  - Security
  - Extendable

Why RabbitMQ?

- RabbitMQ is an implementation of AMQP with extensions written in Erlang
- Many official and unofficial clients
  - .Net, Java, Erlang, Python, Etc
- Extensions / RabbitMQ Specific
  - High Availably Nodes / Clustering
  - Confirmation Mode
  - Message TTL
  - Dead Letter Exchanges

Erlang

- Erlang is a general-purpose concurrent, garbage-collected programming language and runtime system
- Developed in 1986 by Ericsson and open sourced in 1998.
- It was designed to support distributed, fault-tolerant, soft-real-time, non-stop applications.
- Supports hot swapping - code can be changed without stopping a system!
- Designed for concurrency!
- Internally, processes communicate using message passing instead of shared variables, which removes the need for locks.

Message Architecture Planning

- Before you start, plan out the message workflow end to end
- What to send?
  - The message types must be defined
- How much to send?
- Who needs to know?
  - Potential clients and queues should be identified
- How fast can messages be processed?
  - RabbitMQ is fast, but if you produce more than you consume, you will run out of memory
- Before any messages can be published, the Broker objects must be created.

What makes a good message?

- Independent unit of work
  - An order for a Customer X is independent of an order for Customer Y
- Self Contained
  - Should include as much info required to process that can always remain true
  - Avoid extra data lookups when processing
- Avoid relying on the order of delivery if possible
  - If order is important, then try to group them logically
  - Order 1 for Customer X must be processed before Order 2
  - Customer Y's orders have no relevance Customer X

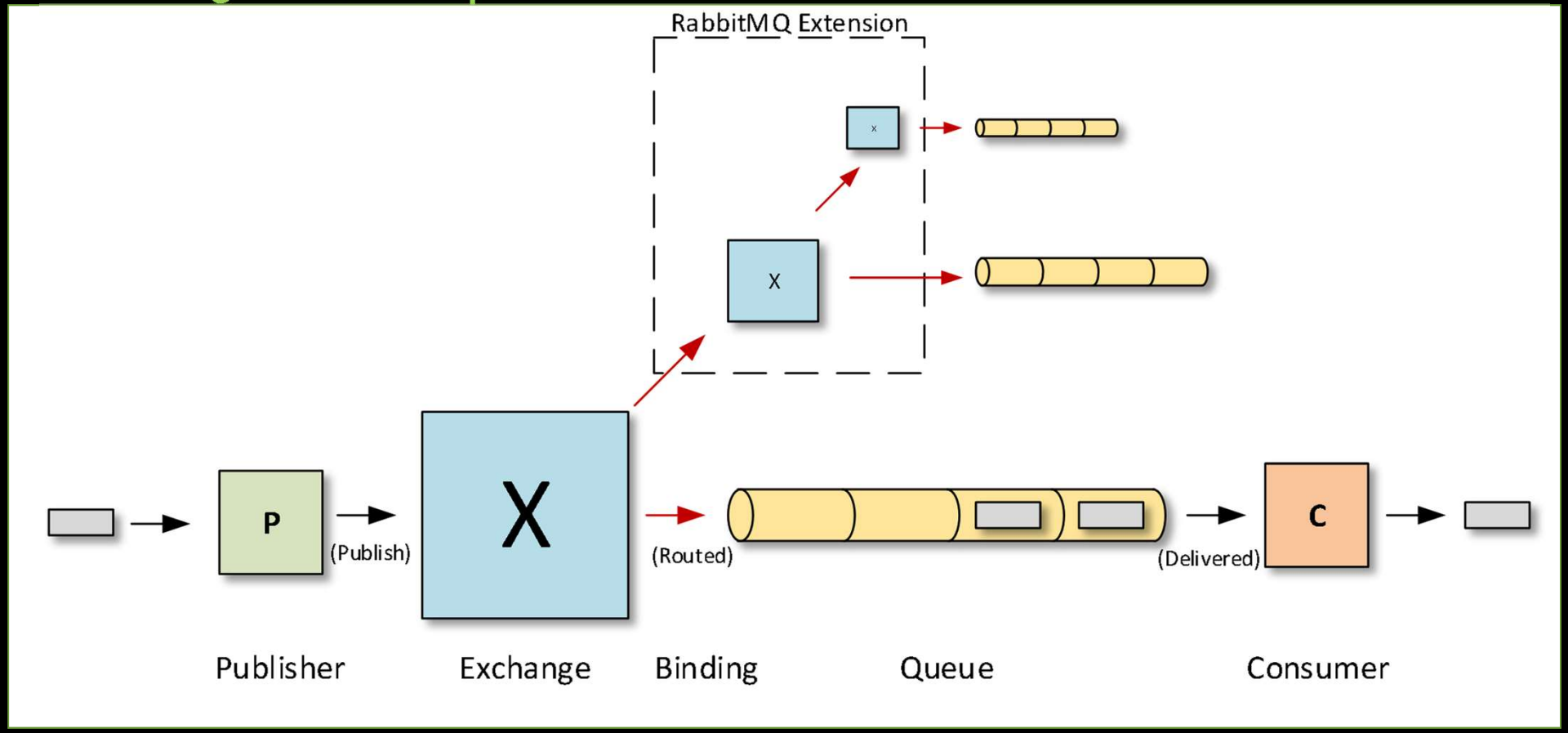
Idempotency

- Ability to process the same message more than once with no adverse side effects
- Expect to see messages more than once, that is by design
- Should design your messages and handlers to process the same message twice
- Scenarios
  - If there is a failure by a consumer, or connection interruption, the broker will decided to resend any unack'd messages
  - If a message is rejected by a particular client with requeue=true
  - If there is a failure in a RabbitMQ node, any unack'd messages will be resent

AMQP Concepts

- Connection
- Messages
- Publishing
- Exchange
  - Bindings
- Queue
  - Bindings
  - Consumer

AMQP Concepts



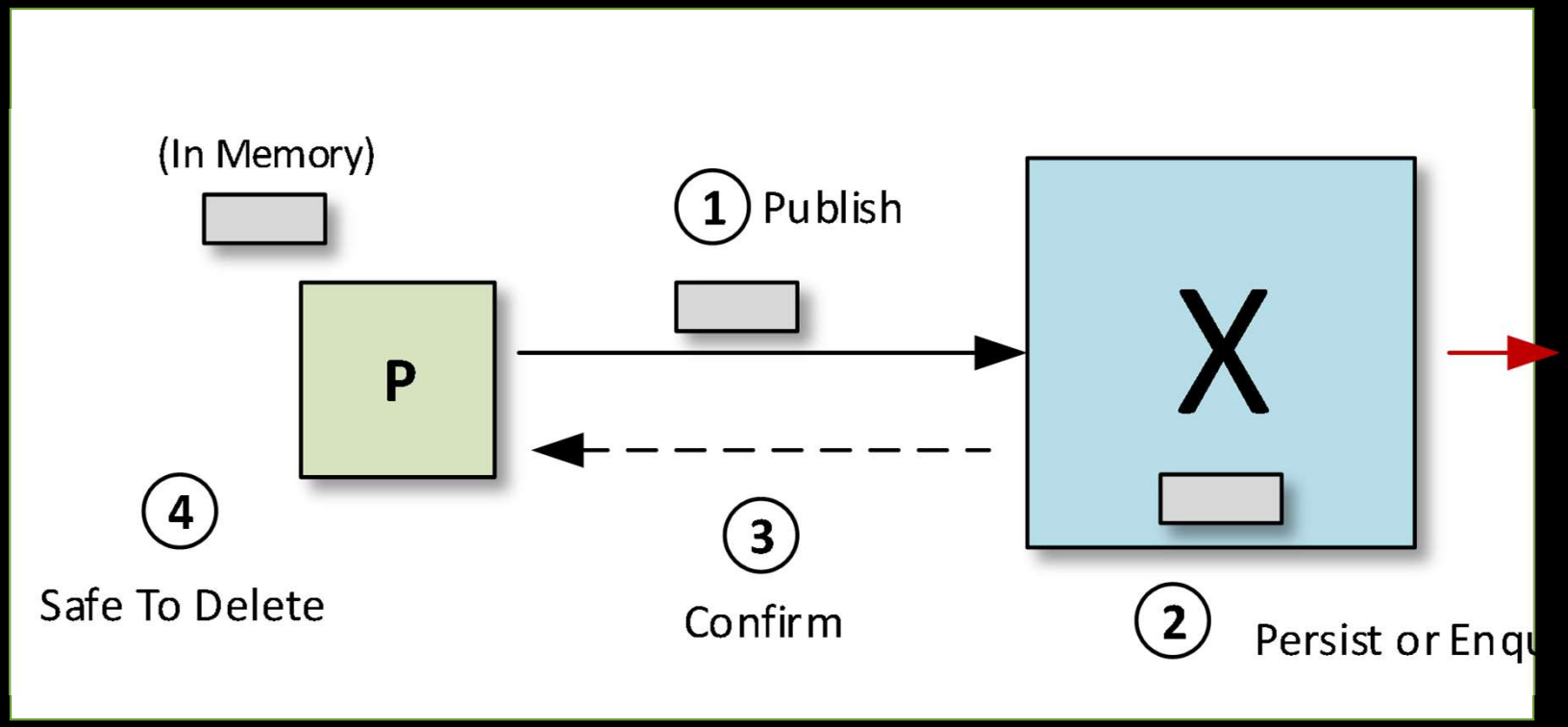
AMQP Client

- 1. Publish Messages
- 2. Consume Messages
  - Get
  - Consumer
- 3. Management
  - Declare Broker Objects (Queues, Exchanges)
  - Configure Broker Objects (Bindings)

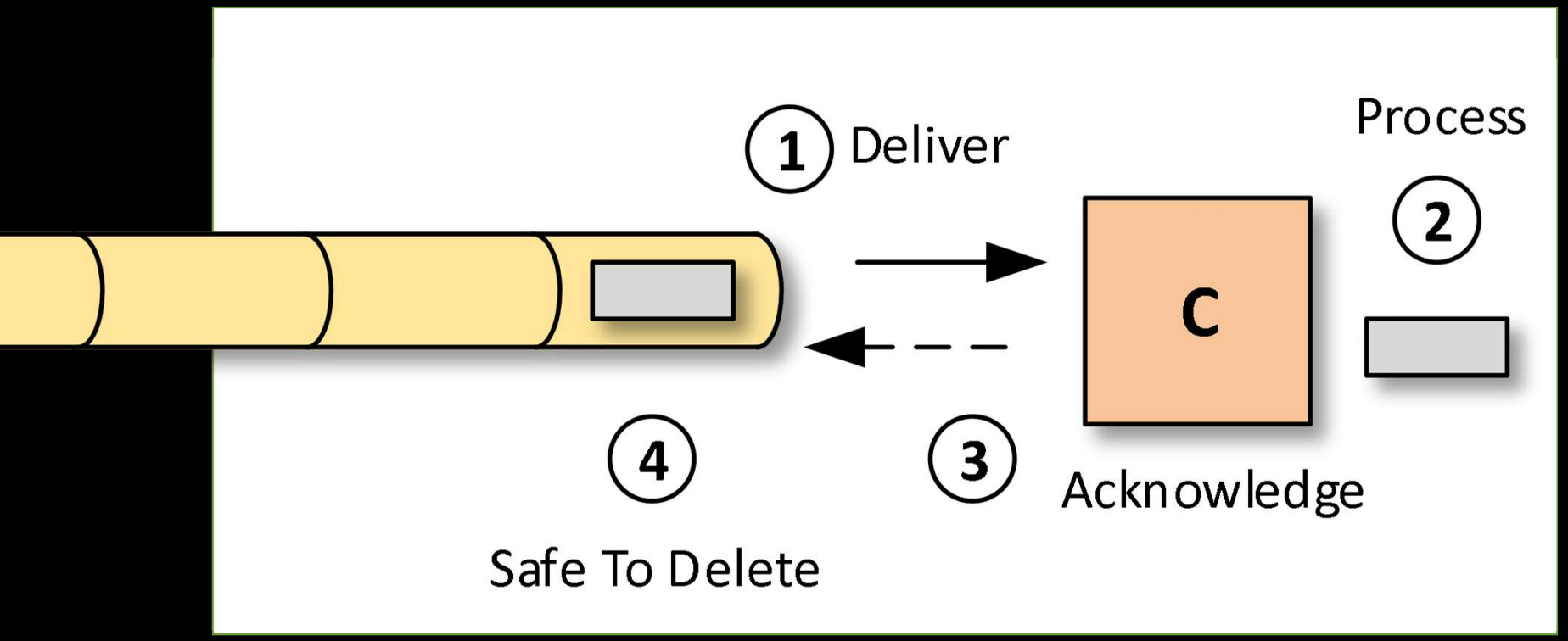
AMQP Client

- User
  - Authentication
  - Server side authorization
  - security on exchanges / queues
- Connection
  - Heartbeat
- Channel (aka Model)
  - Multiple per connection
  - Should use 1 per thread
  - Model has Actions (Publish/Get/Consume)

Transfer of Responsibility (Publisher)



Transfer of Responsibility (Consumer)



Message Publishing

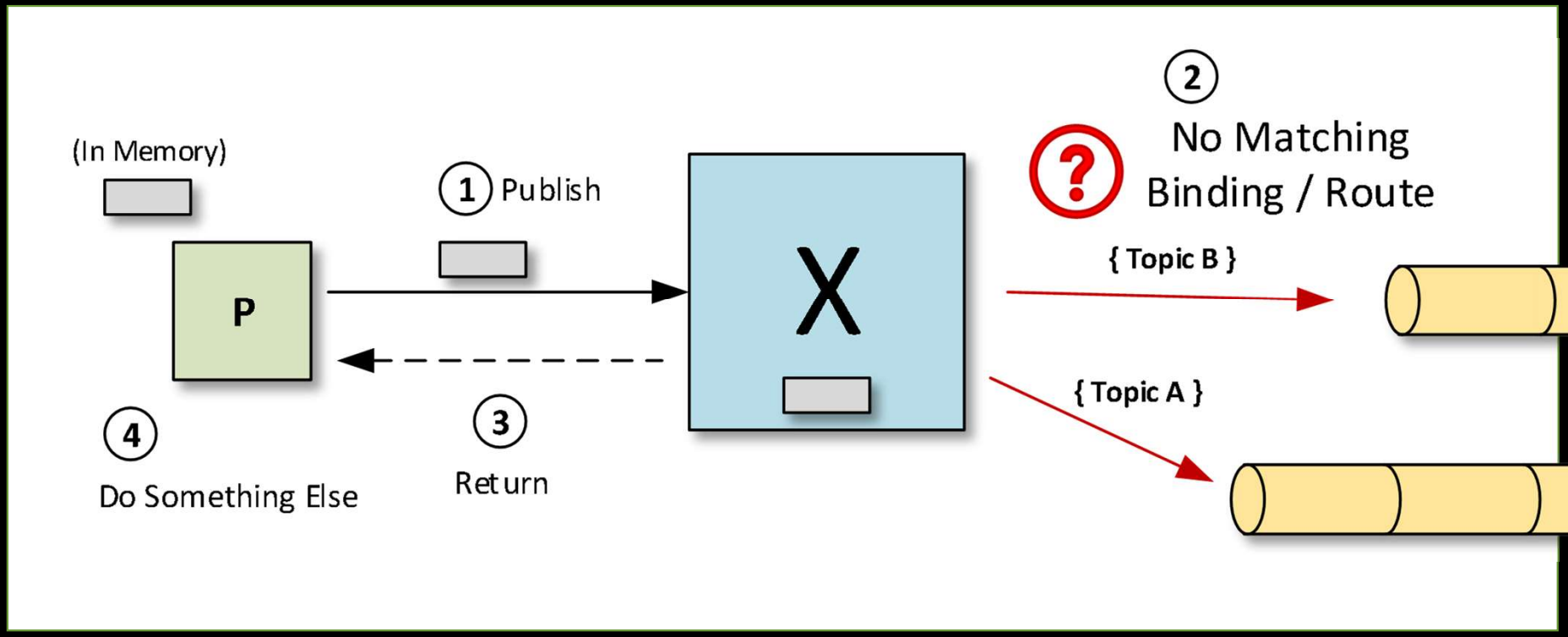
- Byte[] – Payload
  - Up to application to decide what you put in and how
  - Unless using a framework (SpringAMQP)
- Routing Key / Topic
  - If unspecified, “” is used

- Try not to think about the queue when publishing
- It is possible that no one could be listening for your message

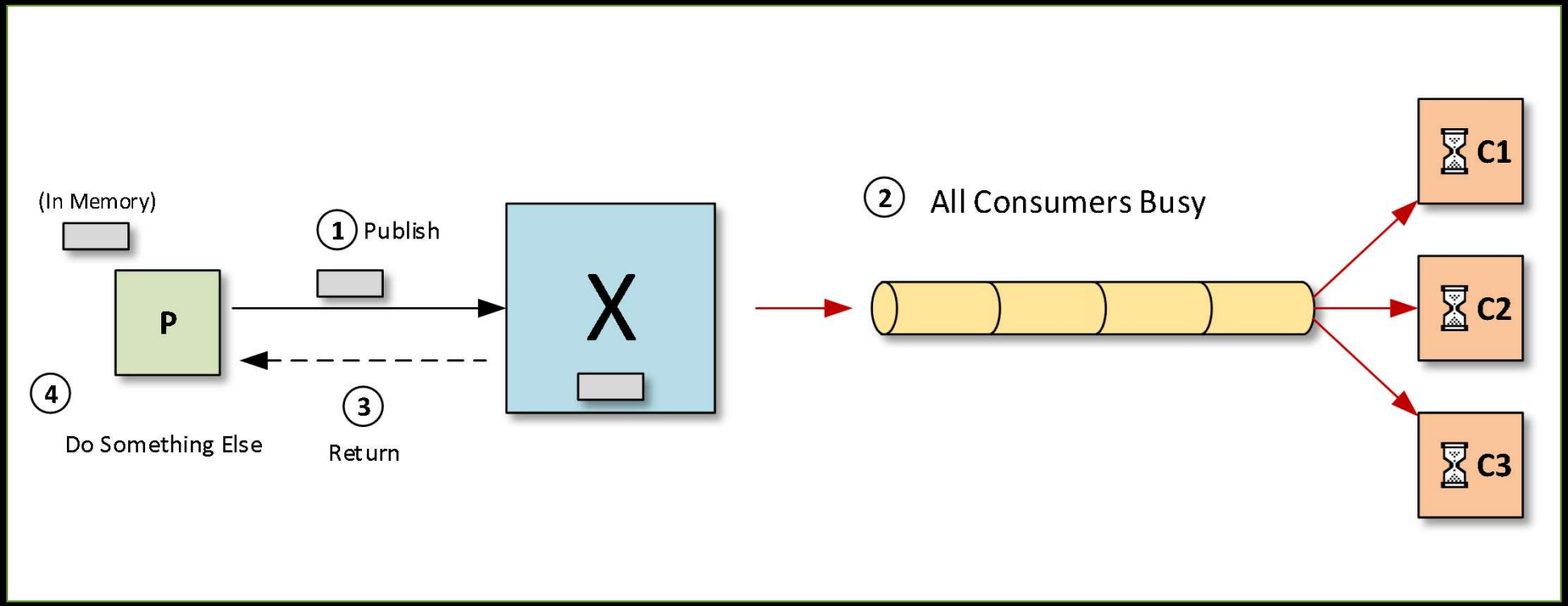
Message Publishing - Flags

- Flags
  - Immediate – requires an idle consumer
  - Mandatory – requires a valid queue to hold the message
- Delivery Mode
  - Persistent – message will be stored to disk (if possible)
  - Transient – message will only be in memory (faster)

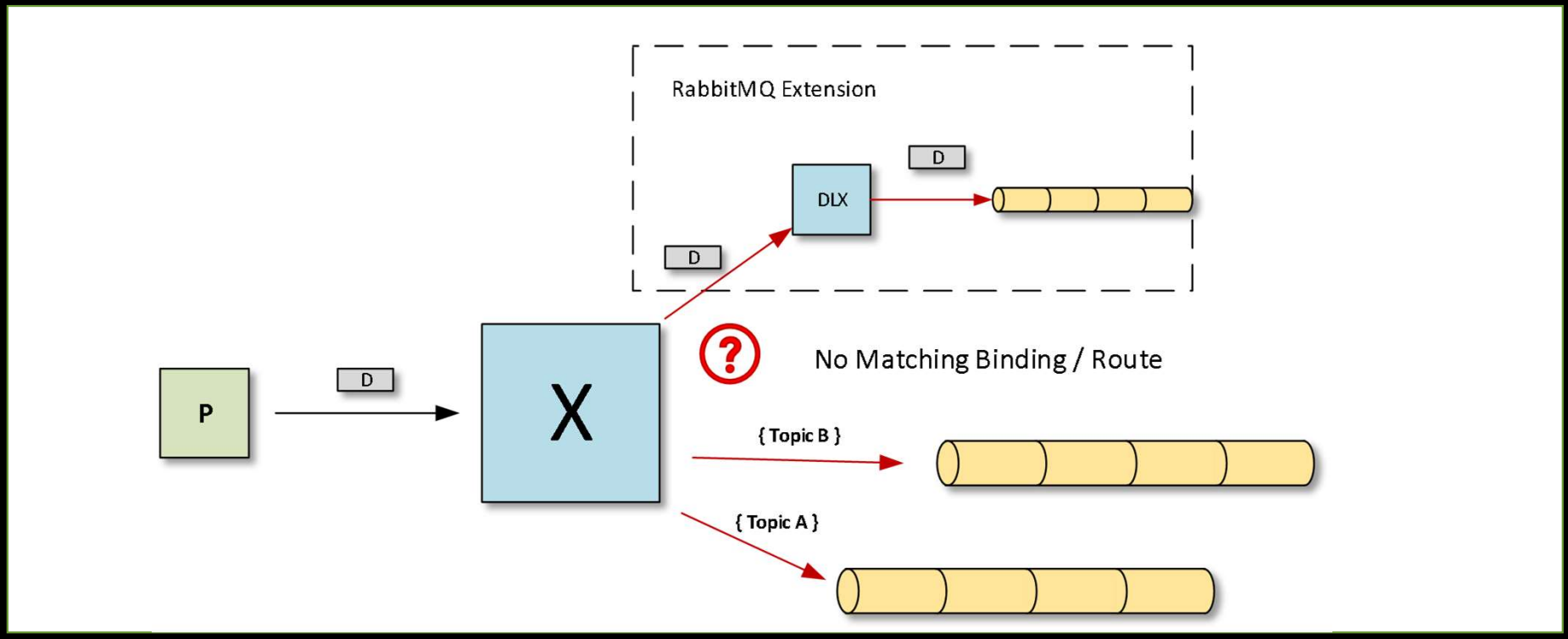
Mandatory Mode



Mandatory Mode



Dead Letter Exchange



Message Publishing – Headers / Properties

- Headers
  - Well known properties
  - Custom user data
- Used by RabbitMQ
  - user_id – use for authentication if enabled
  - expiration - TTL for message
- priority - not enforced by AMQP
- reply_to - should be set if publisher needs a reply

Message Publishing – Headers / Properties

- timestamp
- Type Specifiers
  - content_type
  - content_encoding
  - type
- Identifiers
  - app_id
  - cluster_id
  - correlation_id
  - message_id – client specified / server generates sequence number per consumer which is different

Payload – Byte[]

- If you are using the RabbitMQ client by itself, you will need a pattern for interpreting the message payload
  - Serializing the messages
  - Binary / Xml Serialization
  - Custom Payloads
  - Versioning

Routing Key / Topic

- Aside from actual Payload, this is the single most important message property
- Determines where message is sent based on bindings
- Much more efficient than sending messages that a client can't process

- I prefer 1 topic per message type and perhaps include a version #
  - Application.MessageTypeA.V1
  - Application.MessageTypeB.V1

- (Alternatively could use a header – ContentType + Content Encoding)

Publish Modes

- Standard Mode – no guarantees
- Transaction Mode (Tx)
  - Not the same as ACID / Database transaction
  - Reduces throughput by ~250 times
  - Its there because its in the AMQP standard, but not recommended by RabbitMQ
- Confirmation Mode
  - Serves the same purpose as Tx Mode, just faster (its an RMQ extension)
  - Broker will send a confirmation to the publisher indicating it success 'owns' the messages
  - Can be configured to group together Acks (ie if received N then all before have been received - Exactly like TCP Ack's)

Publish Callbacks (Client Events)

- Basic.Ack – Message Confirmed
  - Client is no longer responsible for it
- Basic.Return – Message could not be confirmed:
  - Immediate Flag
  - Mandatory Flag
- Basic.Nack
  - Rare, problem with broker and client needs to resend

AMQP Concepts

- Connection -
- Messages -
- Publishing -
- Exchange
  - Bindings
- Queue
  - Bindings
  - Consumer

Exchanges

- These are where you publish
- Even if you don’t specify one, it is going to the ‘default exchange’
- Types
  - Fanout – sends out all messages
  - Direct - sends out specific messages
  - Topic – uses topic patterns
  - Header – inspects headers
- Durability

Fanout & Direct - Exchange

- Faster – But you may not care
- Fanout
  - Can not specify topic
  - you get everything
- Direct
  - Must specify every topic you want
  - Potentially a lot of maintenance

Topic Exchanges

- Very Flexibile!
- Allows wild cards in bindings
  - # = all topics
  - Namespace.Class.*
  - Namespace.*.Method
- Can represent Fanout and Direct
  - binding with '#' = fanout
  - binding with 'Topic' = direct
- Because of its more generality, I always use topic

Header Exchange

- May be useful if you have a lot of custom headers
- Only inspects the string properties of header
- Harder to check multiple headers with complex conditions
- Can not fully represent a topic exchange
- Slowest

Bindings – Exchange -> Exchange

- Exchanges can be bound to each other (RabbitMQ)
  - Same rules for binding queues apply to exchanges
- These allow for federation / scaling of messages publication
  - Can be used for distributed setups

Bindings – Exchange -> Queue

- Without bindings, no messages go into a queue
- Depending on the exchange type, different options exist
  - Routing key
    - (Topic / Direct – implied # for fanout)
  - Header

Durability

- Remember Message Delivery Mode? – Transient or Persistent
- A persistent message will only be saved to disk if the Exchange / Queue they pass through are durable
- If the message is transient it will not be stored even if the Exchange / Queue is durable they are (for speed)

Queues

- A client Gets / Consumes messages from a Queue
- Application named or broker named (random)
- Shared vs Exclusive
- Durability

Message Retrieval

- Now that my message is in the queue, How do I get it out?
- You need to know the queue name
- Get Operation
  - Takes a single message
- Consumer (RECOMMENDED)
  - Put the client channel into consumer mode

Acknowledgements

- Used to indicate message processed successfully by client
- AutoAck is used, broker assumes success when it has been delivered to subscriber
- Outstanding Unacked messages tracked by broker
  - Used in QoS
- Can Ack the highest number and perform a batch Ack
- Message will not be redelivered if it is rejected

AMQP Concepts (Recap)

- Connection
- Messages
- Publishing
- Exchange
  - Bindings
- Queue
  - Bindings
  - Consumer

Broker Infrastructure

- Now you understand the purpose of Exchange, Bindings, Queues
- When to define them?
  - Clients can create them on demand, but how will another client know about it?
  - Where will the client know where to publish?
- Static
  - Parts of the infrastructure should always exist and no client should create them (think of a database table)
  - These are the ‘starting points’ for your client to either broadcast its existence to the world, or start publishing to.
  - Manage these object them as you would table schemas and roll out changes carefully
- Dynamic
  - Some parts can be declared at runtime
  - Implicitly dynamic (publishers)

Declaration

- Declaring Exchanges and Queues
  - Always succeeds
  - Idempotent operation
    - Except when configuration is different
  - Some feel its ok to always ‘declare’, but I think could get into problems
- Passive Declaration
  - Will fail if it doesn’t exist
- Declaring returns meta data
    - Message Count
    - Consumer Count
    - Randomized Name (if left blank)

Infrastructure

- Static
  - Exchanges
  - Work Queues
- Dynamic
  - Publishers
  - Observers

Advanced Topics

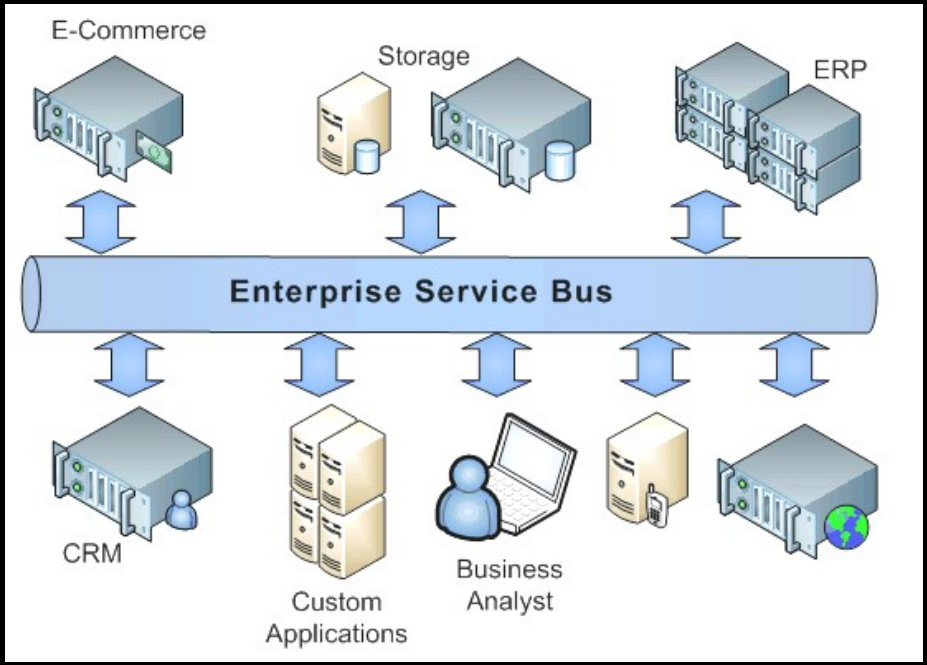
- Message Expiration / Auto Delete
- Error Handling
- Usage Patterns

Message Patterns

- 1 - Publisher – N Subscribers
  - Handled through configuration transparent to publisher
- Work Queue / Load Balanced
  - Need to process ‘every’ messages
  - as fast as possible
  - Can be done in parallel
  - Order not important
- Observer
  - Just want to be able to ‘peek’ be notified of what is going
  - No business processes based off the events
  - Logging is a simple example

Enterprise Service Bus

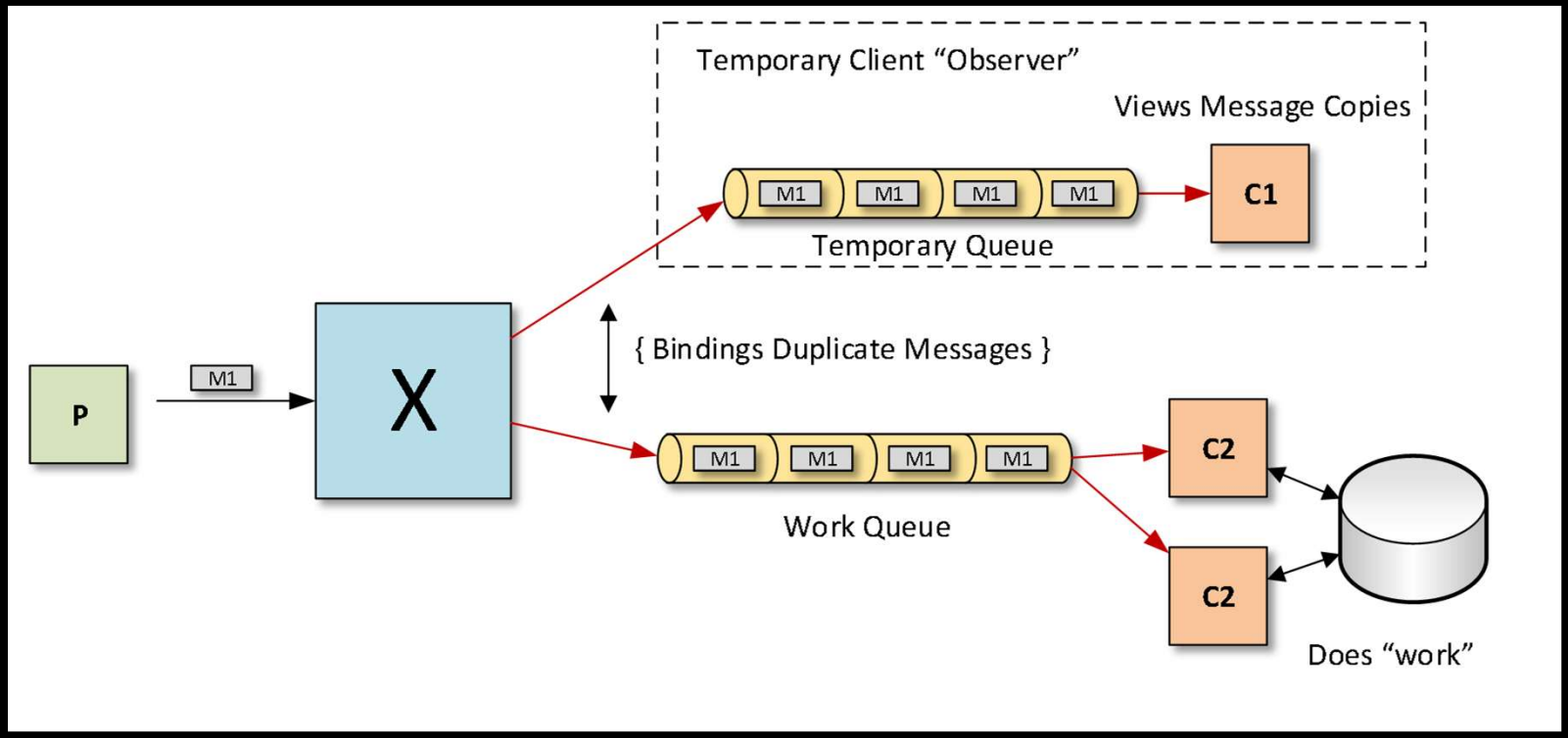
Common Pattern Allow for Many Pub / Subs



Observer

- Client wants to see current events (doesn't care about past events)
- Queue - used by one consumer
  - Temporary – Name assigned by Broker
  - Exclusive – Don't want anyone else taking messages
  - Non-Durable – who cares if the messages are lost?
  - Auto ack mode -there is no real work
  - Auto-Delete – Remove the queue when client disconnects
- A new one will be created next time client
- Messages dropped when client is not listening

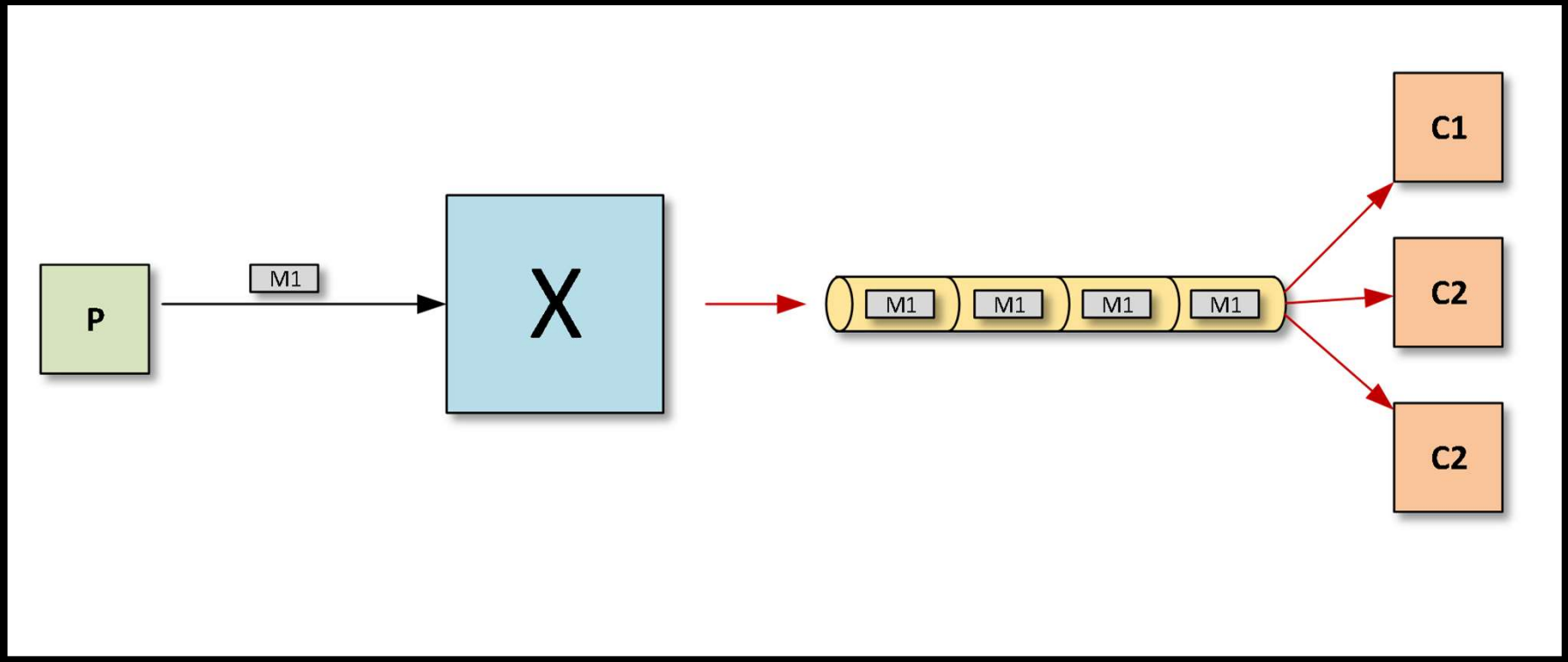
Observer


Temporary Client "Observer"
Views Message Copies
M1
M1
M1
M1
C1
Temporary Queue
{ Bindings Duplicate Messages }
P
M1
X
M1
M1
M1
M1
C2
C2
Work Queue
Does "work"

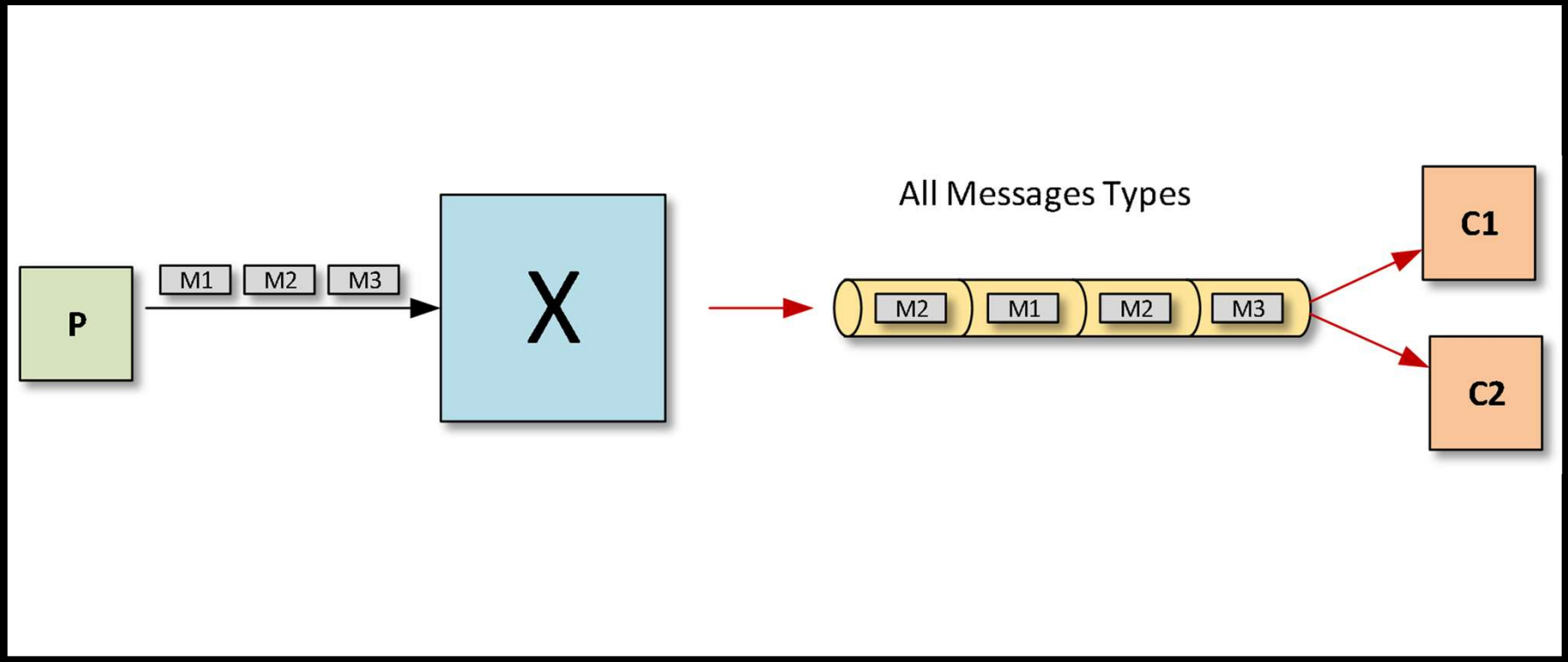
Work Queue

- Queue - used by multiple consumer
  - Permanent Name – Defined by application
  - Shared – needs to be for load balance
  - Durable – who cares if the messages are lost?
  - Manual Ack - only when successfully processed
  - No-Auto-Delete – Remove the queue when client disconnects
- Messages will persist between client connects and disconnects regardless if anyone is listening

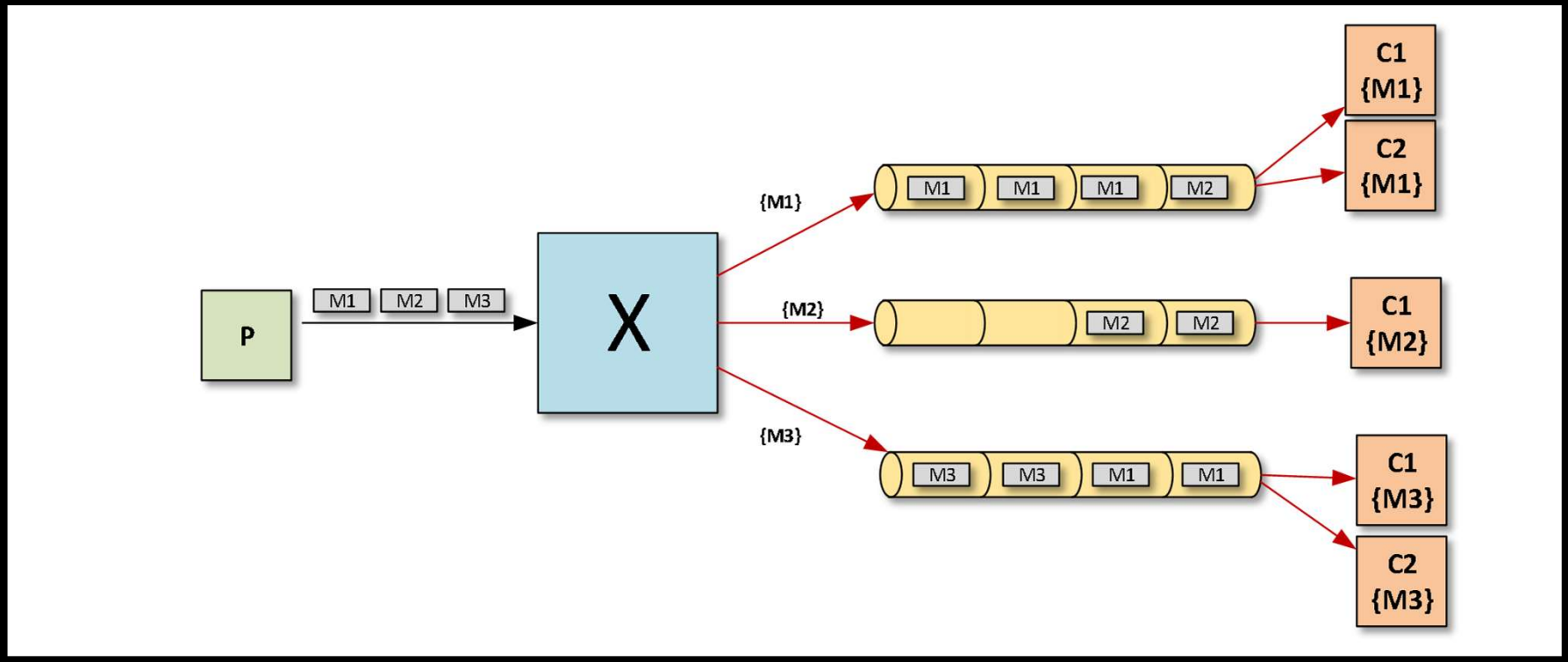
Work Queue (Load Balancing)



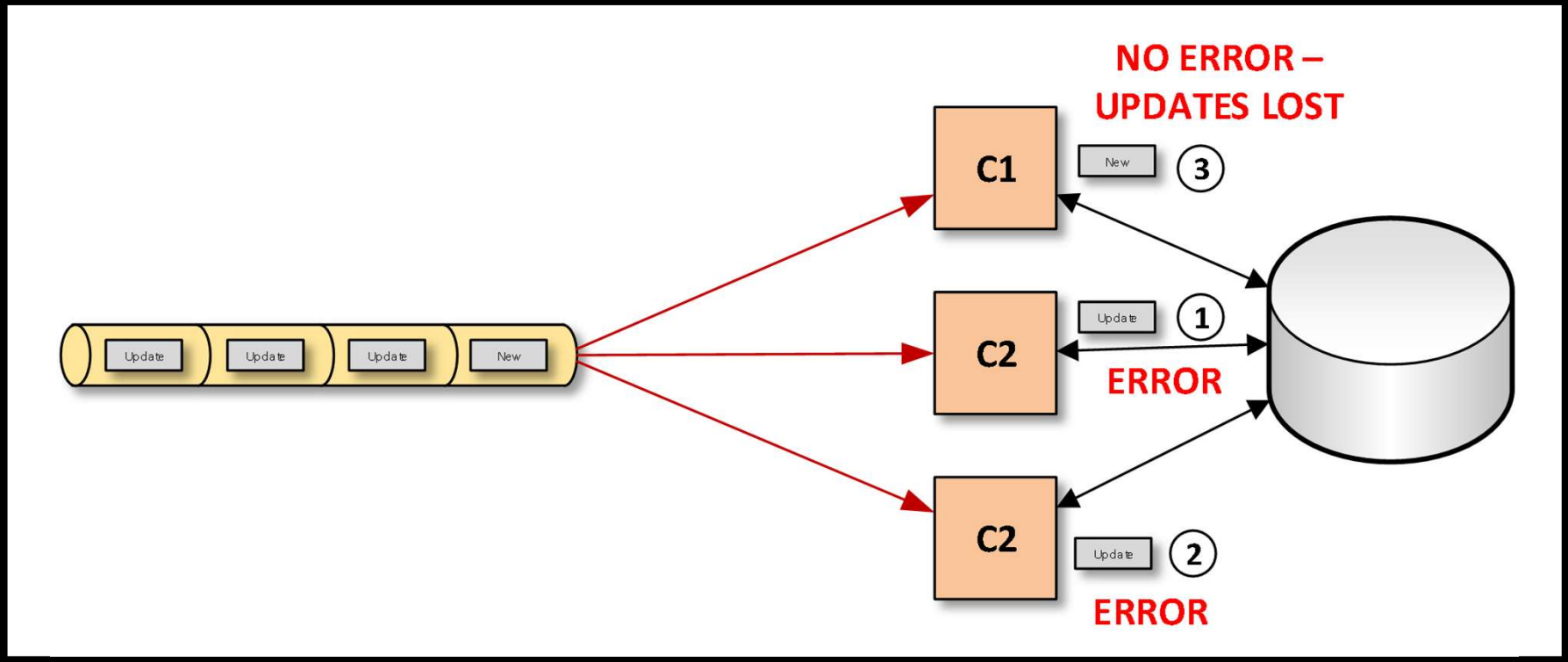
Work Queue (Anti-Pattern)



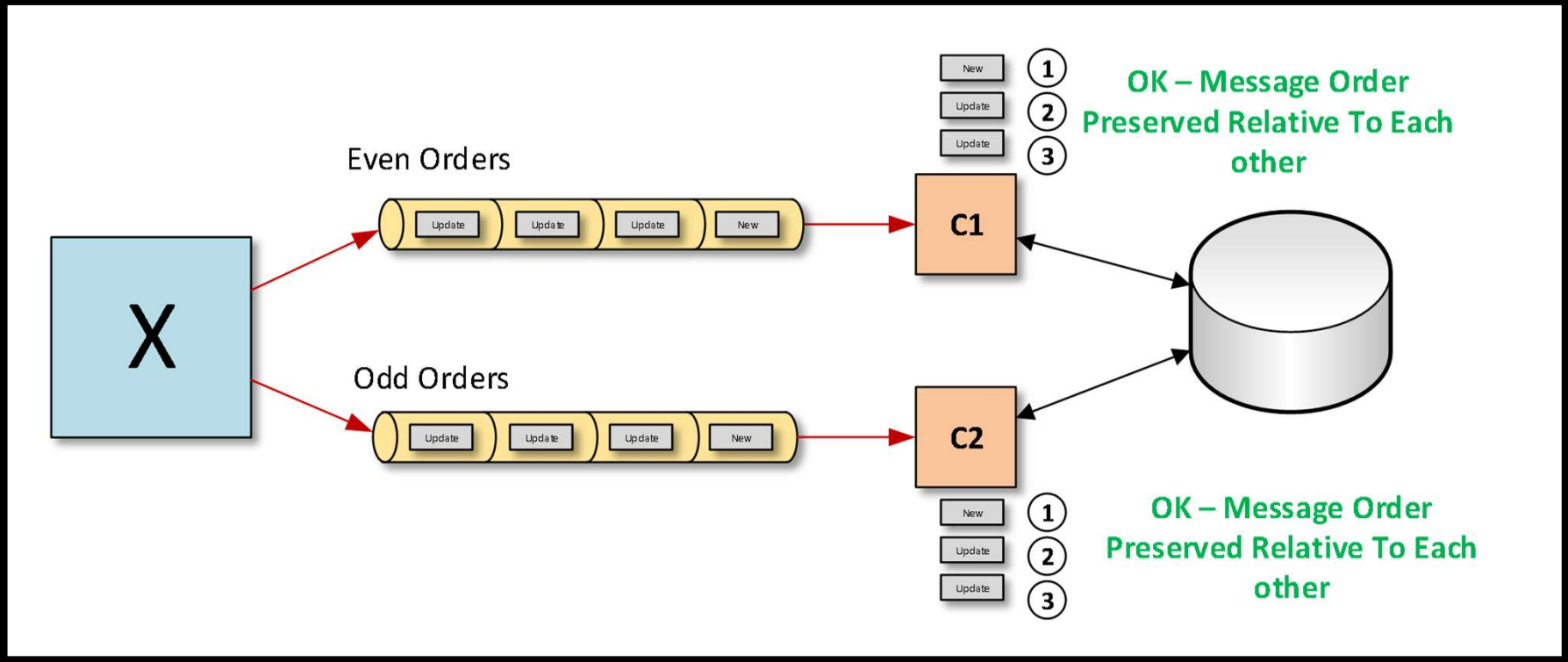
Work Queue (Anti-Pattern) - Solution



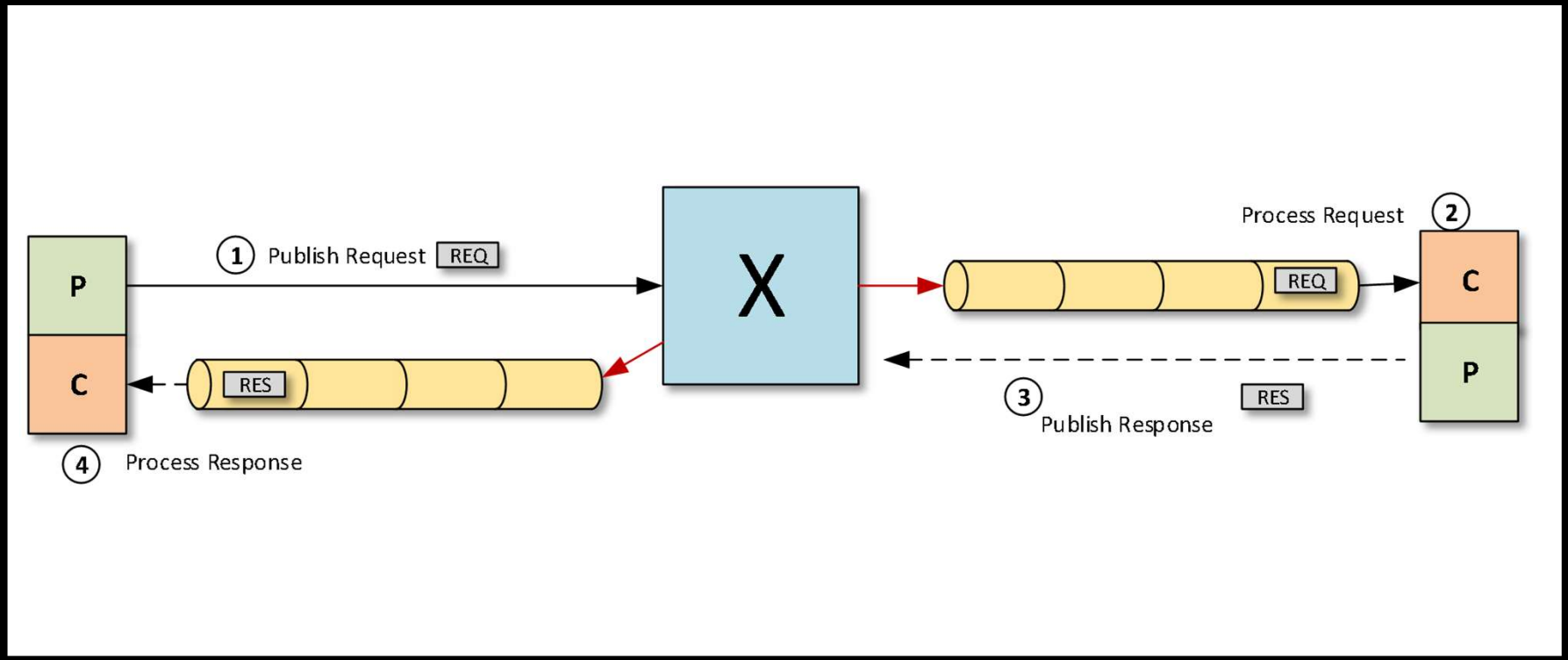
Work Queue (Anti-Pattern)
Message Order Dependency



Work Queue (Anti-Pattern) - Solution



Request - Response



Time To Live / Auto Delete

- Messages, Queues, Exchanges can all be cleaned up automatically
  - Useful or dynamically created resources not static infrastructure
- Message TTL
  - Can be set per Message
  - Can be set per Queue
  - Message cannot exceed queue's TTL
- Auto Delete
  - Queues can auto-delete after last client disconnects
  - Exchanges can auto-delete after last queue is deleted / unbound
  - Only takes after at least 1 connected

Other Client Events (Connection / Channel)

- I suggest subscribing to ALL events on the channel and connection
- Flow Control – Can kick in if the high/low watermarks are hit for memory
- Basic.RecoverOk
- ModelShutdown
- CallbackException

Error Handling

- When using acknowledgements
  - Bad messages can hold up the queue and/or create infinite loops
  - Must ack it with a 'Reject'
- Basic.Reject(Requeue=False)
  - Message is dropped
- Basic.Reject(Requeue=True)
  - put back in the queue(in the same spot) and it will be reprocessed again
  - If the same error happens again, it will reprocess this message indefinitely
- There is a 'Redelivery' flag but No Redelivery Count

No Redelivery Count

- The redelivery flag will be set regardless if the a client rejects it explicitly or it was dropped due to disconnection
- Some errors are transient (remote server busy) and might not occur the 2nd time
- You can not make this determination based on a single flag. A count of redelivery attempts could help your error handling logic determine when its time to ‘Reject’

Pattern for Error Handling

```
try {
        ProcessMessage
} catch(TransientException) {

        If (msg.Redelivery = false) {
                Reject(msg, requeue = true)  // Perhaps the error wont occur next time?
        } else {
                Reject(msg, requeue = false) // Possibly 2nd time
        }
} catch(FatalException) {
        Reject(msg, requeue = false)
}
```

Dead Letter Exchange

- Configured per queue
- Tells queue where to resend message if it has been ‘Rejected’ by a client
- Expired Messages also sent
- This is much better safer than acking and republishing in client code as it is all handled and guaranteed by the broker
- Message sent unaltered to DLX.
  - Don’t forget to create a queue
- Optional Dead Letter Topic
  - Sets an additional property on the message

QoS Settings

- Global vs Local
  - Can be set per connection
  - Or set for all consumers
- Prefetch
  - Reduces the amount of network round trips
  - By default it is unlimited will send as many messages to a client as it has memory
  - Undesirable for round-robin / load balancing between long processes

More Messaging Anti-Patterns

- Basic.Get
- Ack / Republish Bad Messages
- Excessive RPC

Basic.Get

- Requires Polling / Busy Loop
- Wastes a thread
  - While(true) {
    - Var msg = channel.get();
    - Sleep(100);
  - }
- Only 1 Message at a time
- Doesn't Take advantage of AMQP 'Consumers'
  - Doesn't tie into QoS
  - Not Truly Asynchronous / Event Based
  - Always Use Consumers

Remote Procedure Calls

- It is somewhat cool that you can use RabbitMQ and wire up WCF or Java RPC calls
  - var message = client.Method(parameter);
- Use it with restraint. It can easily get out of hand
- Not obvious code being used is queue / dequeue messages
- Not really designing a message based application

More Advanced Topics

- Protocol / Message Versioning
- High Availability
- Distributed Systems

Multi-Version Protocols

- A large real-time system may be need to run with no down time
- Need to deploy updates and changes to messages definitions
- If using Topic or Header exchanges, this can be done as follows

High-Availability / Mirrored Clusters (Single Node)

- RabbitMQ supports built in clustering
- Building / Joining nodes to clusters will increase fault tolerance
- Messages will be can replicated to N out of M nodes in cluster
- Warning! – AMQP does not spec says nothing about clusters!
  - If the node a client connects to fails, it is up to the client to reconnect to another node!
  - Either let all clients know about all nodes
  - Or, use a hardware based solution (better for transparency)
- RabbitMQ Must be the same version for all nodes

Distributed RabbitMQ (Multiple Nodes)

- Can be used between versions of RabbitMQ

- Shovel Plugin
  - Copies ‘ALL’ messages from 1 exchange to a remote exchange
    - Creates a local queue

- Federation
  - Optimized to only send messages to remote exchanges for relevant queues

RabbitMQ over the internet Example

- Local RabbitMQ
- Federated
- Cloud RabbitMQ