Northeastern University - Seattle

CS6650 Building Scalable Distributed Systems

Professor Ian Gorton

Building Scalable Distributed Systems

Week 8 – Asynchronous Systems

Learning Objectives

- Describe the advantes and disadvantages of asynchrouns systems
- Compare and contrast message queues and publish-subscribe
- Explain the key features of RabbitMQ
- Describe how clustering and mirroring work with messaging

Outline

- Asynchronous systems basics
- Example: RabbitMQ
- Messaging patterns
- Clustering and mirroring

Asynchronous Systems Basics

Background

- So far we've examined communications such as:
  - Sockets
  - RPC
  - HTTP
- These work great!
  - Simple to program, eg Thrift, protocol buffers
  - Fast comms
- But lead to tight coupling:
  - Client needs to know who server is
  - What happens if the server is not available?

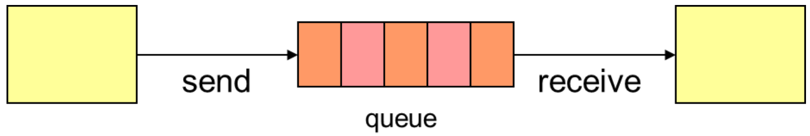
Middleware

- Middleware is the plumbing or wiring of many applications
- Provides applications with fundamental services for distributed computing
- Insulates applications from underlying platform (OS, DBMS, etc) APIs
- Lots of middleware exists
  - Different purposes
  - Different vendors
  - Different standards and proprietary technologies
- Focus of today – Message Oriented Middleware (MOM)

Messaging - MOM

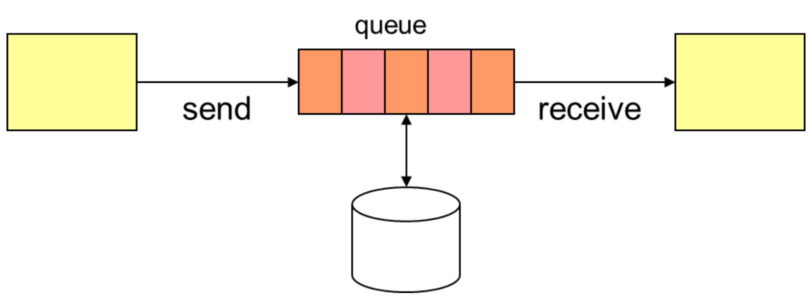
- Message Oriented Middleware (MOM) provides:
    - Asynchronous communications between processes, applications and systems
    - Send-and-forget - Delivering messages despite failures
    - Transactional Messaging - Deliver all messages in a transaction, or none
    - Persistence - Messages can be logged at the server and hence survive server failure

Messaging Primitives



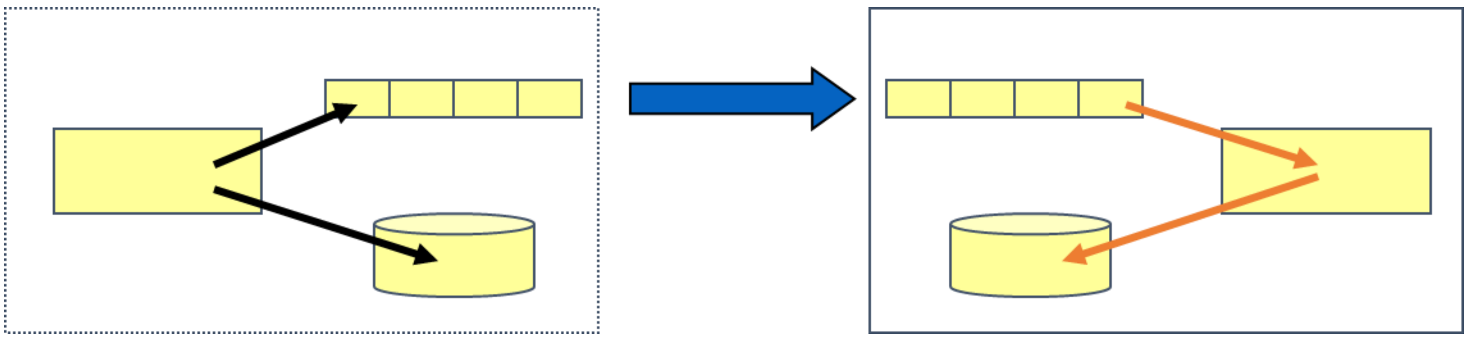
- Send (queue, message)
  - Put message onto queue
- Receive (queue, message)
  - Get message from queue
- No dependency on state of receiving application on message send

Messaging Primitives: Persistence



- Receipt of message at queue implies message is written to disk log
- Removal of message from queue deletes message from disk log
- Trade-off performance versus reliability

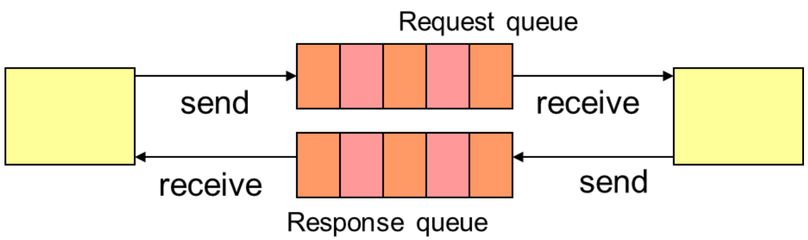
Messaging Primitives: Transactions



```
Begin transaction
...
update database record
put message on queue
...
commit transaction
```

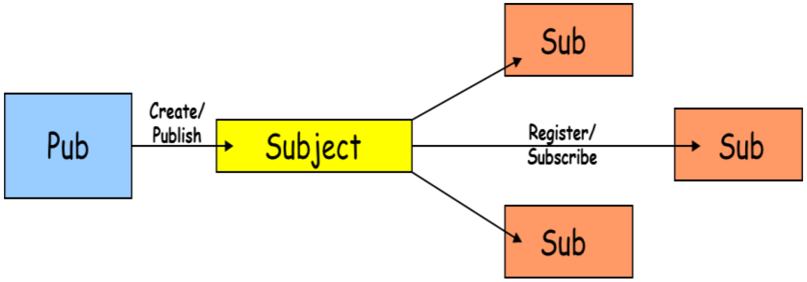
```
Begin transaction
...
get message from queue
update database record
...
commit transaction
```

Messaging Primitives: Transactions

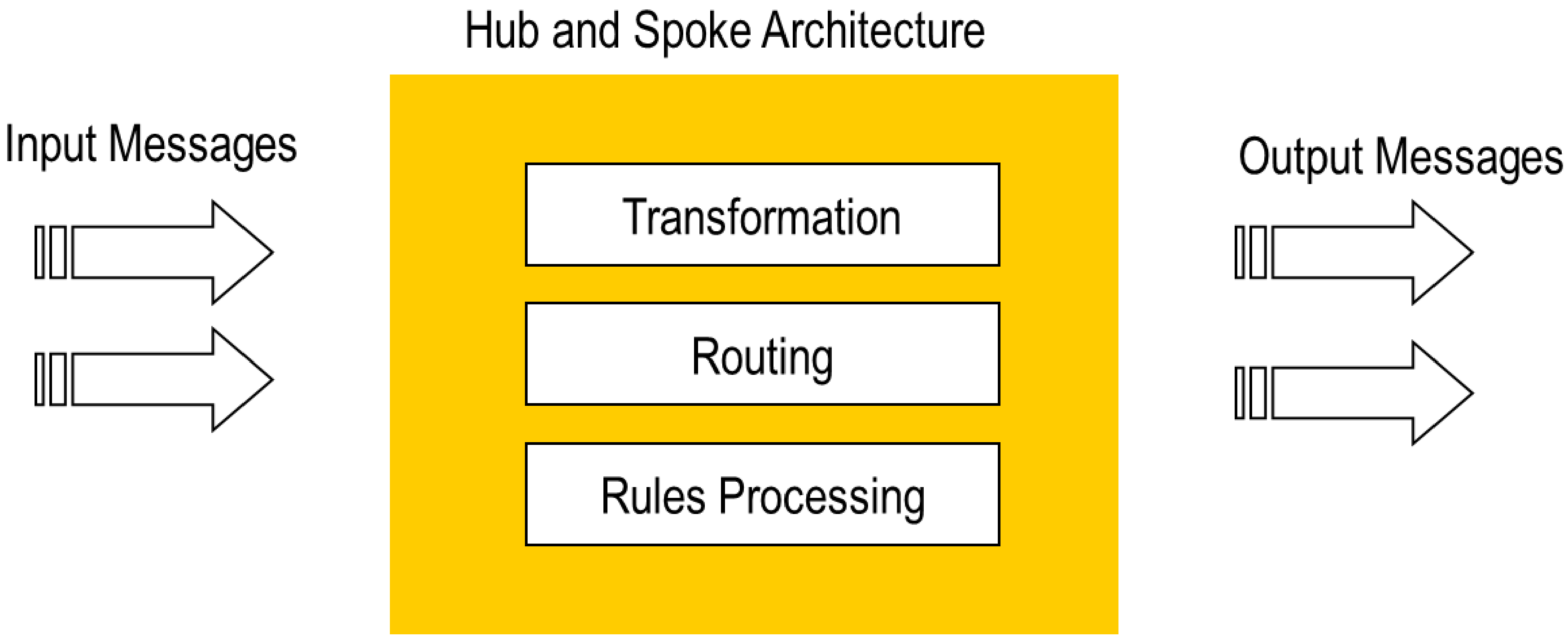


- Sender and receiver do *not* share a transaction
  - Rollback on receiver does not affect the sender
- 'Synchronous' operations are not atomic (Request/response is 3 transactions not 1)
  - Put to request queue
  - Get from request queue, put to response queue
  - Get from response queue

Messaging Primitives: Publish-Subscribe



- Provide 1-to-N, N-to-1, and N-to-N communications
- Messages are ‘published’ to logical *subjects* or *topics*
- Subscribers receive all messages from subjects they subscribe to



Message Brokers

(A big step towards an Enterprise Service Bus (ESB))

Messaging Implementations

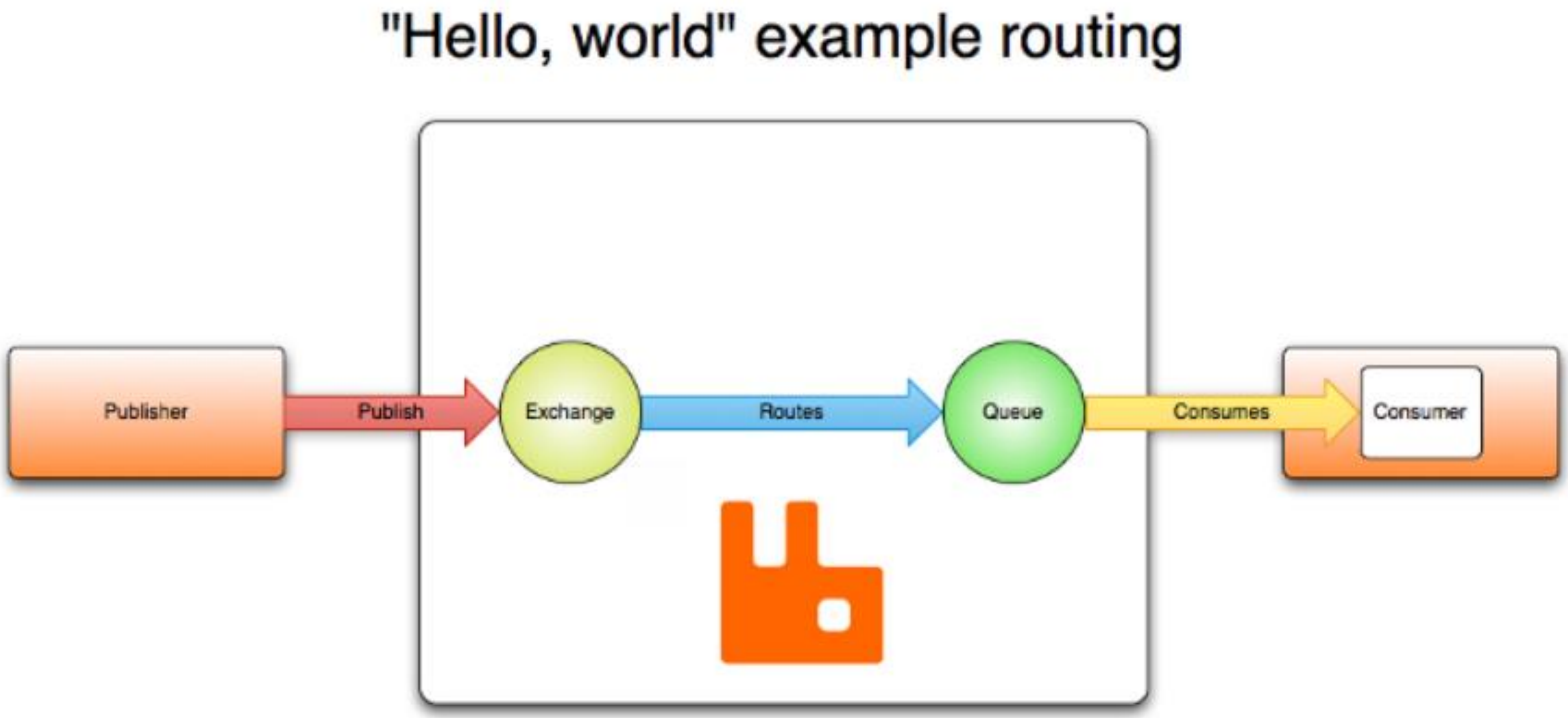
- Many! Many!! Many!!!
- Each has different:
  - Features
  - Strengths
  - Weaknesses
- In general there are two extreme choices:
  - Smart endpoints/dumb pipes
  - Dumb endpoints/smart pipes
- Which scales best?

Example: RabbitMQ

- Widely used, open source broker that implements AMQP protocol
- messages are published to exchanges
- Exchanges distribute messages to queues using rules called bindings.
- the broker can deliver messages to consumers subscribed to queues
- Optionally consumers fetch/pull messages from queues on demand

RabbitMQ
Open Source Enterprise Messaging

Examples: RabbitMQ



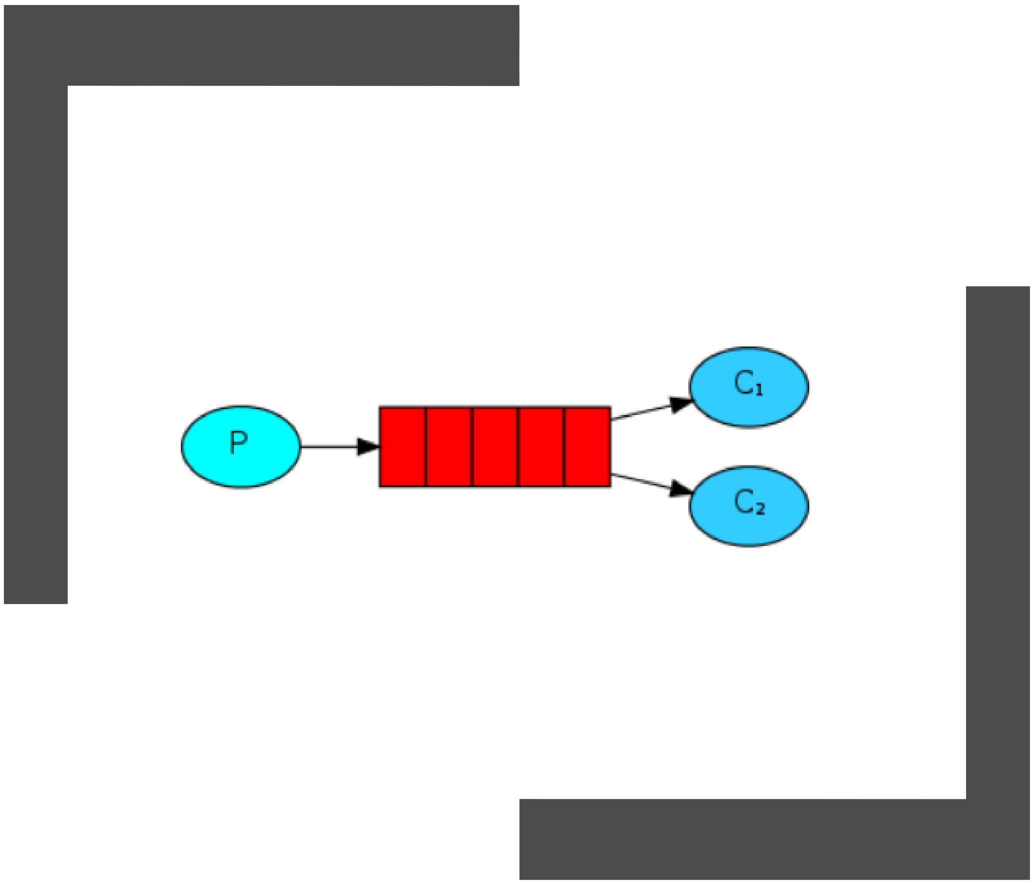
Producer

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

public class Send {

    private final static String QUEUE_NAME = "hello";

    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        try (Connection connection = factory.newConnection();
             Channel channel = connection.createChannel()) {
            channel.queueDeclare(QUEUE_NAME, false, false, false, null);
            String message = "Hello World!";
            channel.basicPublish("", QUEUE_NAME, null, message.getBytes("UTF-8"));
            System.out.println(" [x] Sent '" + message + "'");
        }
    }
}
```

Consumer

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;

public class Recv {

    private final static String QUEUE_NAME = "hello";

    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(), "UTF-8");
            System.out.println(" [x] Received '" + message + "'");
        };
        channel.basicConsume(QUEUE_NAME, true, deliverCallback, consumerTag -> { });
    }
}
```

Multiple Consumers

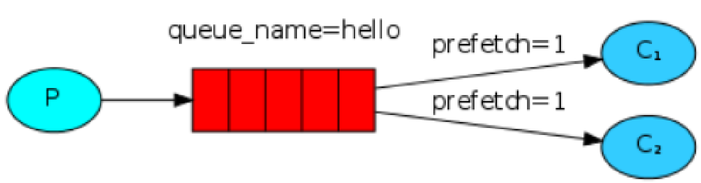


- Messages distributed in a round robin fashion to consumers
- Each message sent to exactly one consumer
- Consumers send message ack when processing complete
- Unack-ed messages redistributed when a consumer dies

Consumer – Manual Acknowledge

```
channel.basicQos(1); // accept only one unack-ed message at a time

DeliverCallback deliverCallback = (consumerTag, delivery) -> {
  String message = new String(delivery.getBody(), "UTF-8");

  System.out.println(" [x] Received '" + message + "'");
  try {
    ProcessMessage(message);
  } finally {
    System.out.println(" [x] Done");
    channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
  }
};
boolean autoAck = false;
channel.basicConsume(TASK_QUEUE_NAME, autoAck, deliverCallback,
consumerTag -> { });
```

Message Distribution



- Limits number of messages sent to a single consumer
- If consumer is busy, messages are sent to next consumer with capacity

Persistent/Durable Messaging

- If RabbitMQ broker fails, it loses:
  - All queues on the server
  - All messages in the queues
- For safety, we need to persist the broker configuration and messages

Persistent/Durable Messaging

```
// in producer and consumer
boolean durable = true;
channel.queueDeclare("task_queue", durable, false, false, null);
```

```
// and ….
import com.rabbitmq.client.MessageProperties;

channel.basicPublish("", "task_queue",
        MessageProperties.PERSISTENT_TEXT_PLAIN,
        message.getBytes());
```

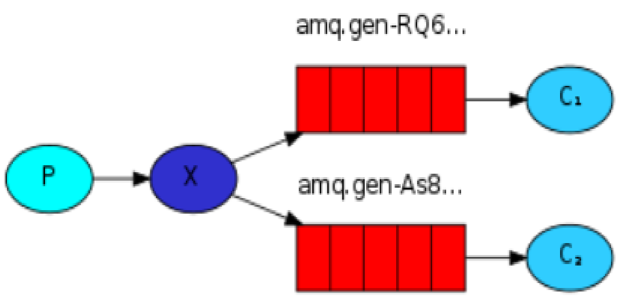
Publish-Subscribe

- Every message sent to all subscribers
- Utilize RabbitMQ exchanges
- Producers send messages to exchanges
- Exchanges distribute messages
  - One queue
  - Multiple queues
  - No queues
- Determined by *exchange type*
  - Default exchange is “”
- For pub-sub, we use a *fanout* exchange
  - Also *direct, topic, headers*

Publish-Subscribe

- Fanout exchange sends a message to all queues it knows of:
- In publisher and subscriber:
  - channel.exchangeDeclare("logs", "fanout");
- In publisher:
  - channel.basicPublish("logs", "", null, message.getBytes());
- In subscriber:
  - String queueName = channel.queueDeclare().getQueue();
  - channel.queueBind(queueName, EXCHANGE_NAME, "");

Publish-Subscribe

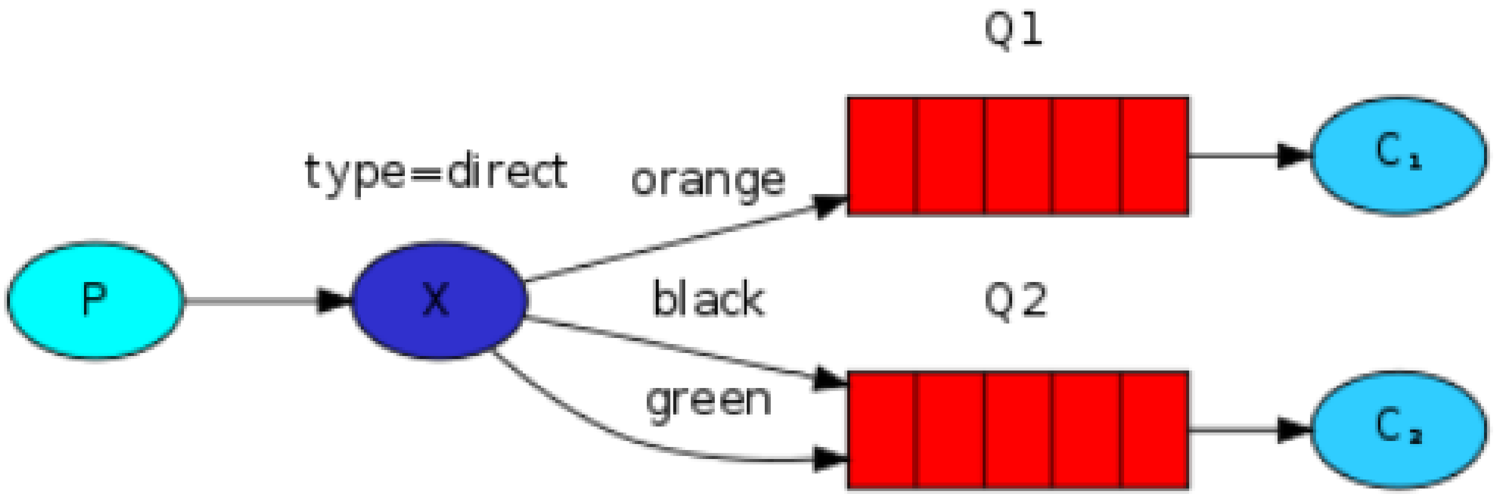


- Subscriber creates and binds a temporary queue per subscriber to the exchange
- Temp Q is:
  - non-durable
  - autodelete
  - generated queue name
- When a new subscriber connects to the exchange, it only sees newly published messages

Message Filtering

- Send messages to queues based on attributes/content
  - Associate Binding key with a queue in subscriber
- Use a direct exchange
  - message goes to the queues whose binding key exactly matches the message routing key
  - Message discarded if no match
- A queue can specify more than one binding key

Message Filtering



Message Filtering

- In publisher

  channel.exchangeDeclare(EXCHANGE_NAME, "direct");

  channel.basicPublish(EXCHANGE_NAME, ***“black”***, null, message.getBytes());

- In one subscriber

  channel.exchangeDeclare(EXCHANGE_NAME, "direct");

  String queueName = channel.queueDeclare().getQueue();

  channel.queueBind(queueName, EXCHANGE_NAME, ***“black”***);

  channel.queueBind(queueName, EXCHANGE_NAME, ***“green”***);

- In other subscriber

  channel.exchangeDeclare(EXCHANGE_NAME, "direct");

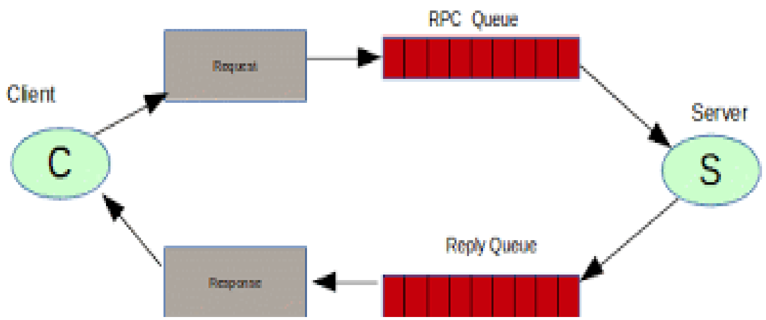
  String queueName = channel.queueDeclare().getQueue();

  channel.queueBind(queueName, EXCHANGE_NAME, ***“orange”***);

Multi-criteria Filtering

- Filtering based on multiple criteria
- Topic exchanges
  - Khoury.staff.announce
  - Khoury.student.announce
  - Khoury.student.party
  - #.announce
  - Khoury.*.announce
  - Khoury.student.*

RabbitMQ RPC – Send and Receive

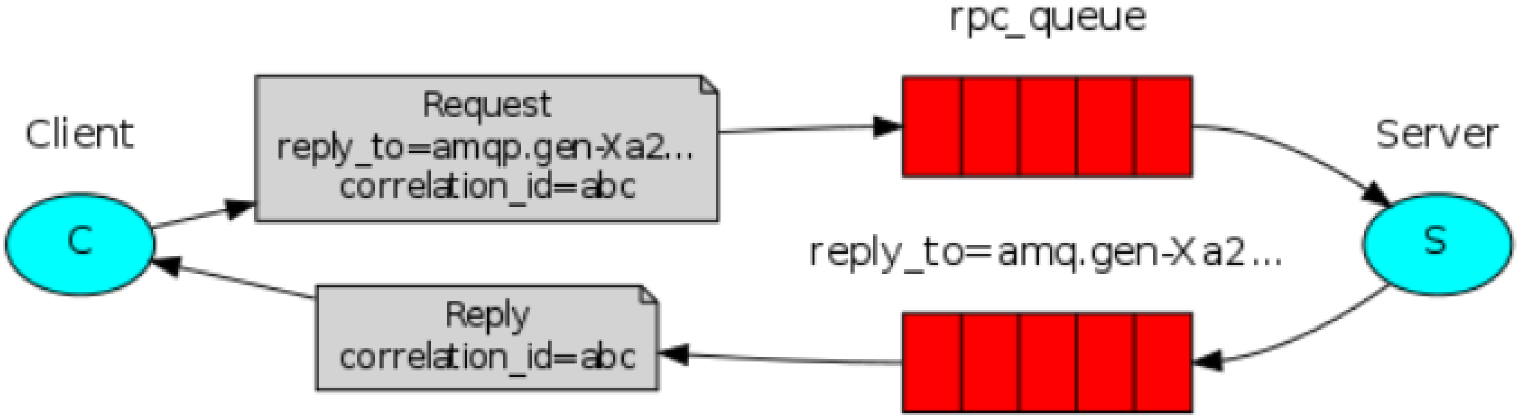


- Emulate a synchronous call with queues
- Client sends request via a queue and specifies a *callback queue*
- Server processes the message and sends result to callback queue

Correlation IDs

- How does client know which request a response is related to?
    - All in same response queue
    - Might not be in the same order as sent
- Solution is a Correlation ID
    - Unique value for every request
    - Sent in response message
    - Enable matching response to request

Correlating Requests and Responses



RabbitMQ RPC

- Client sends message with two properties:
  - replyTo: anonymous exclusive queue created just for the request,
  - correlationId: unique value for every request.
- The request is sent to an rpc_queue queue.
- The RPC worker (aka: server) waits for requests on rpc_queue. On Arrival:
  - processes request
  - sends a message with the result back to the Client, using the queue from the replyTo field, includes correlationId
- Client waits on the reply queue. On arrival:
  - checks the correlationId property matches the value from the request
  - Prcoesses the response

RabbitMQ RPC Client

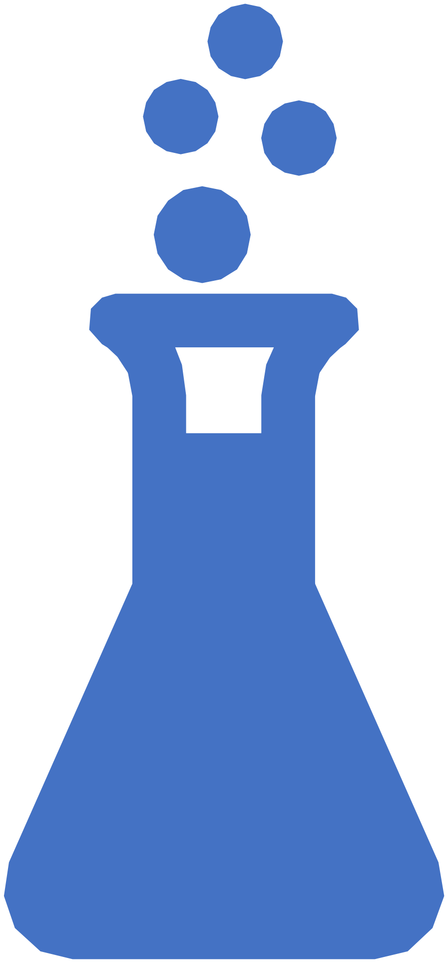
- Server and client examples – check ‘em out ☺

```
// construct message with necessary properties
String replyQueueName = channel.queueDeclare().getQueue();
    AMQP.BasicProperties props = new AMQP.BasicProperties
        .Builder()
        .correlationId(corrId)
        .replyTo(replyQueueName)
        .build();

channel.basicPublish("", requestQueueName, props, message.getBytes("UTF-8"));
```

RabbitMQ RPC Server

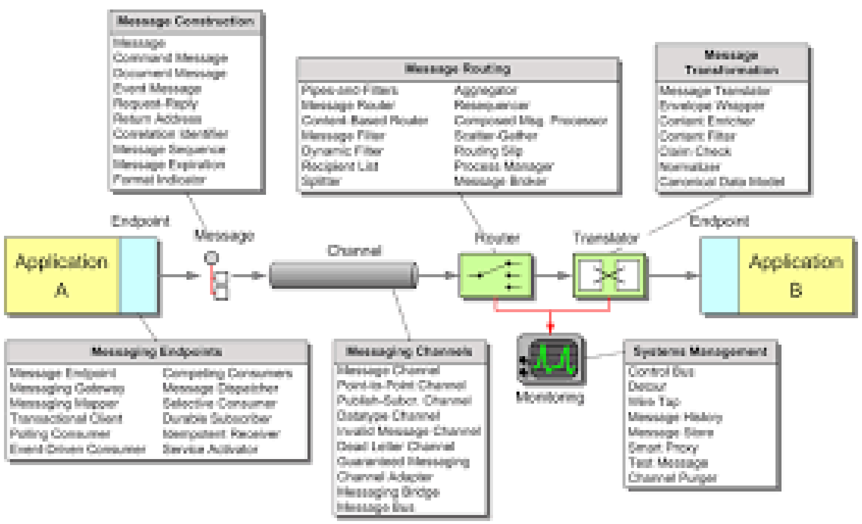
```
// on message arrival
AMQP.BasicProperties replyProps = new AMQP.BasicProperties
                .Builder()
               .correlationId(delivery.getProperties().getCorrelationId())
                .build();


// process the message and create response

channel.basicPublish("", delivery.getProperties().getReplyTo(), replyProps,
      response.getBytes("UTF-8"));
channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
```

Lab

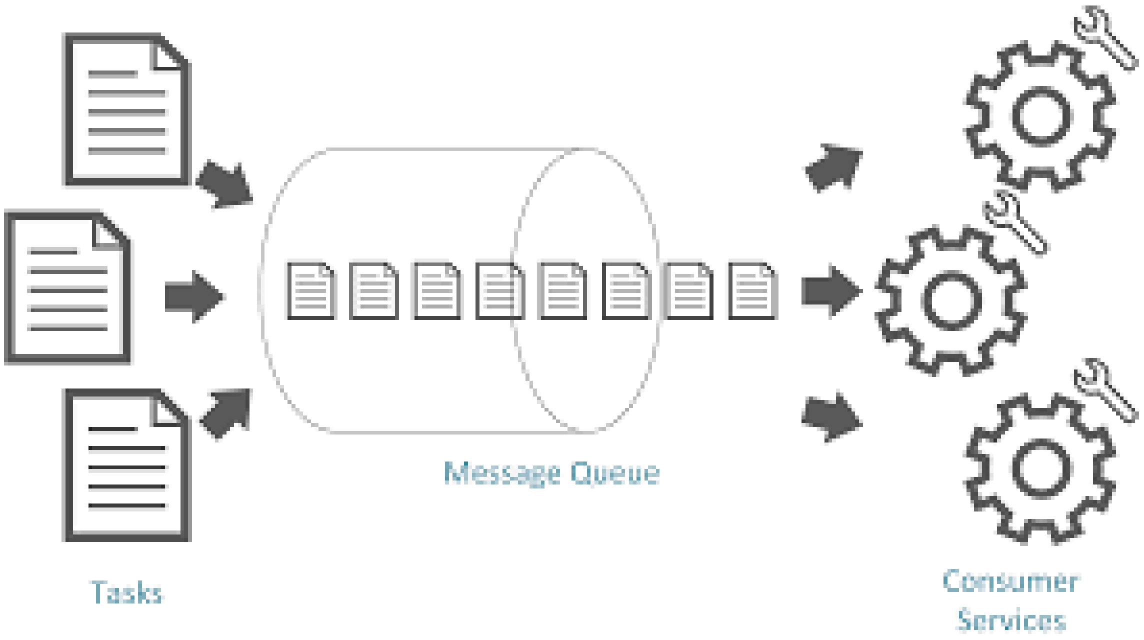


Messaging Patterns

Messaging Patterns Catalog



- Catalog of ~65 patterns at [Enterprise Integration Patterns]
- Comprehensive
- Some of these important for scalability



Competing Consumers Pattern

Advantages

- **Scalability**: consumers can be increased or decreased on the fly.
- **Availability**: If consumers are unresponsive or overloaded, the queue can still store messages
- **Guaranteed Delivery**: At least once
- **Failover**: If a consumer fails mid-task, the message is returned to the queue, to be picked up by another consumer.

Implications

Message Ordering: no guarantee of the order in which messages are processed

Poison Messages: how do we handle malformed messages that cause crashes/exceptions?

Idempotence: consumers must implement idempotent processing

At least once processing

- In RabbitMQ
  - A consuming application should acknowledge a messages when all processing completed.
  - Once ack received, the broker is free to mark the message for deletion from the queue
- If an acknowledgement is not received
  - RMQ [detects connection/node failures]
  - Delivers message to another worker
  - Marks message with *redelivered* flag
- Ergo, processing must be idempotent

Message Invisibility Window

- Messages hidden in queue until processing confirmed
- In some systems, (eg RMQ), messages deleted by broker when acknowledgement received
- In others, (eg AWS SQS), consumer must delete message within invisibility window time
  - Default 30 seconds
- If not deleted, message becomes visible for processing again
  - Ergo idempotence

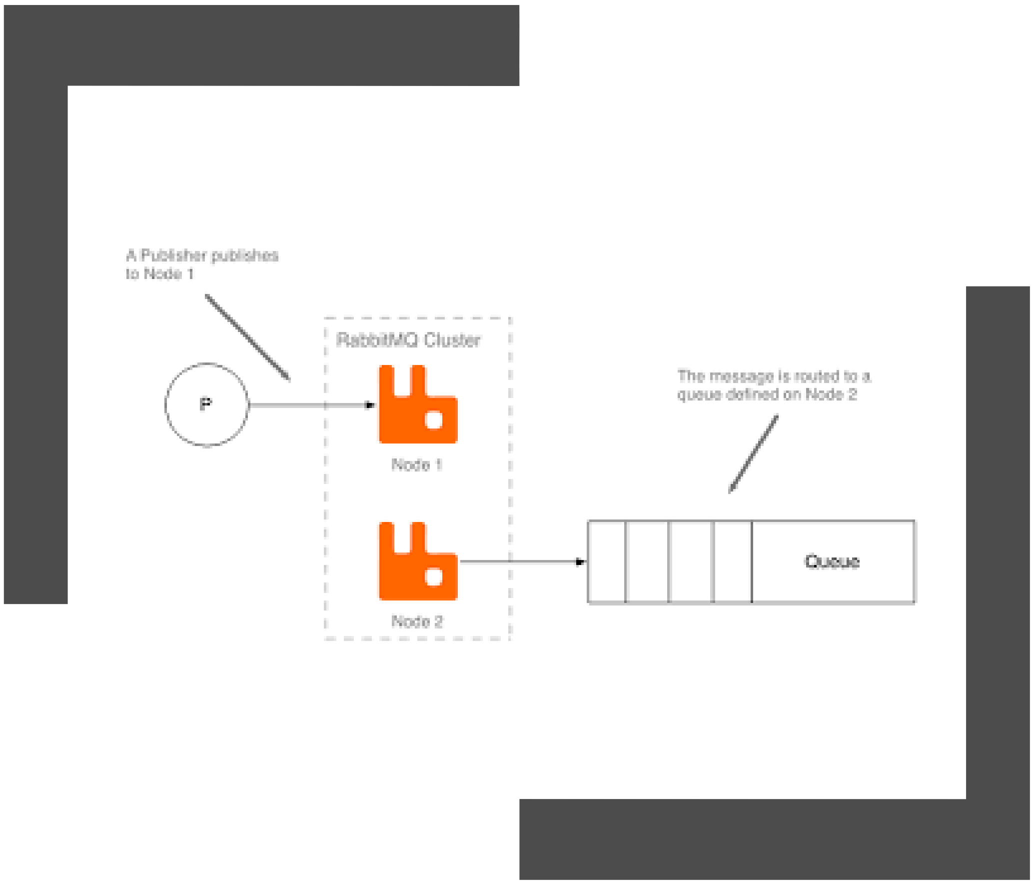
Poison Messages

- Corrupted messages, malicious content, programming bugs -> all may cause workers to fail
  - Poison messages
- Workers continue to fail on redelivery
  - Not good!!
- Somehow must detect these and stop them from being continually delivered
  - Deliver to *dead.letter* queue
  - Set an alarm
  - Diagnose problem

Detecting Poison Messages

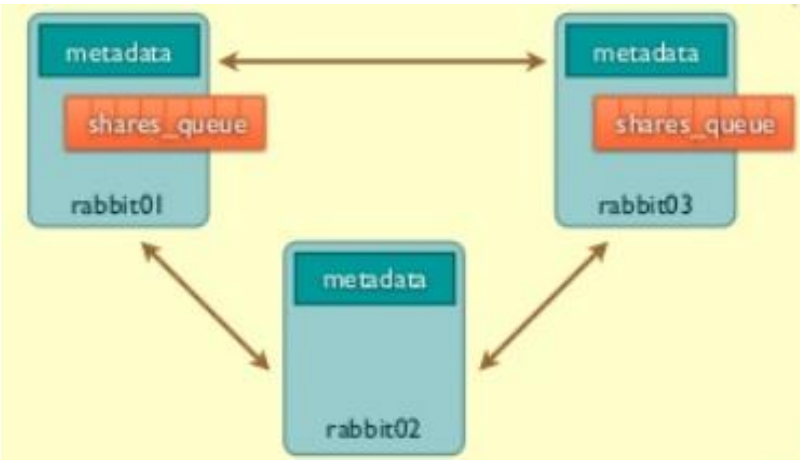
- In RMQ:
  - If a client dies while processing a message, message is requeued
  - Next time it is delivered a redelivered header value is set
  - Client may also send a ‘reject’ acknowledgement and specify that the message should not be requeued
  - Clients can send poison messages to a DLX
- In other systems (e.g, AWS SQS)
  - Message have a delivery count
  - If delivery count > n (3?), send to dead letter queue

Scaling Messaging



- Broker clustering in RMQ
- Brokers act as peers
  - Authenticate
  - replicate state
- Queues not replicated
- Clients connect to any broker and and requests are forwarded to queue origin

Queue Mirroring



- RMQ enables queues to be mirrored across cluster members
- Each queue has a master copy and is mirrored to other secondaries
  - Configurable batch sizes
- Producers connect to master
  - Ensures FIFO ordering
  - Mirrors for data safety
  - Consumers can read from master or mirrors

Master Failure

- If master fails, a mirror is chosen as new master
    - Most up to date mirror
    - Any non-replicated messages from master are lost
- Upon election, new master:
    - Makes all unacknowledged messages available on queue
    - Why?
- Clients must be aware of possible message redelivery

Summary

- Asynchronous systems can be built with messaging
- Point to Point
- Publish-Subscribe
- Many messaging platforms
- Messaging patterns
- Clustering and mirroring