RabbitMQ 4.0 Masterclass: The Definitive Guide

Ashish

RabbitMQ

docker

GO

Ashish

RabbitMQ™

- Fundamentals
- Advanced Concepts
- High Performance Messaging
- Best Practices

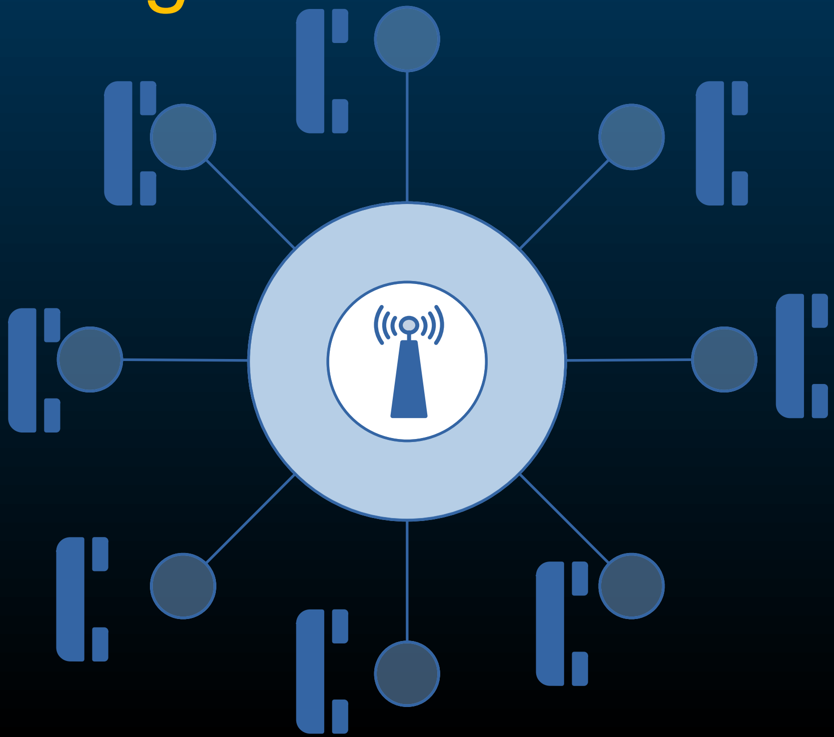
Ashish

RabbitMQ™

- Erlang: general purpose, concurrent, functional programming language.
- Concurrency
- Fault Tolerance
- Distributed
- Hot Code Upgrades

RabbitMQ™

- Joe Armstrong, Robert Virding and Mike Williams
- Telecommunications
- Messaging Systems
- Financial Systems
- E-commerce
- IoT (Internet Of Things)



Ashish

Significance of Erlang in RabbitMQ

- Concurrency and Scalability
- Fault Tolerance and Reliability
- Distributed Systems
- Hot Code Upgrades
- Erlang Ecosystem

Ashish

RabbitMQ™

Developed by Rabbit Technologies

The problem before RabbitMQ

- Tightly coupled systems
- Asynchronous Communication
- Reliability Issues

Ashish

RabbitMQ™

Similar Technologies/Tools before RabbitMQ

- JMS (Java Message Service)
- ActiveMQ
- ZeroMQ
- IBM MQ (formerly WebSphere MQ)

RabbitMQ™

- Decouples Components
- Provides Asynchronous Messaging
- Enhances Scalability
- Ensures Fault Tolerance
- Supports Multiple Protocols

Ashish

RabbitMQ™

- Not a Framework ❌
- Not a Library ❌
- A Tool (a powerful tool) ✅

Ashish

- Handles Asynchronous Communication
- Enable Loose Coupling
- Support Scalability
- Ensure Fault Tolerance
- Integrate with Other Systems

Ashish

RabbitMQ™

What problems does it solve?

- Message Queuing
- Message Routing
- Guaranteed Message Delivery
- Load Balancing
- Real Time Communication

RabbitMQ™

A Tool for Modern Architectures

- Microservices
- Event Driven Applications
- Real-time Systems

RabbitMQ™

Key function of a Message Broker

- Message Queuing
- Message Routing
- Message Transformation
- Handling Message Delivery

RabbitMQ™

Decoupling

Asynchronous Communication

Scalability

Fault Tolerance

Ashish

RabbitMQ Concepts

- **Producer:** an app/service that sends messages to RabbitMQ
- **Consumer:** an app/service that receives and processes messages from RabbitMQ
- **Queue:** where RabbitMQ stores messages.
- **Exchange:** routing mechanism. Types: Direct, Fanout, Topic, Headers
- **Binding:** link between and and a queue
- **Routing Key:** a string that helps the exchange determine where to send a message
  - For direct exchanges, the routing key is compared to the binding key of queues
  - For topic exchanges, the routing key acts as a pattern-matching string

RabbitMQ Concepts

- Message: it consists of two parts:
  - Body: The actual data or content of the message
  - Headers/Properties: Metadata that can be attached to the message
- Acknowledgement (Ack)
- Virtual Hosts (vhosts): a logical separation within the RabbitMQ broker
- Clustering: allowing you to distribute the workload across multiple servers.
- Management Plugin: RabbitMQ provides a management plugin that allows you to manage and monitor RabbitMQ through a web-based interface

Ashish

AMQP and Other Protocols

- AMQP (Advanced Message Queuing Protocol)
  - open standard for messaging and is the primary protocol used by RabbitMQ
  - defines the structure of messages, and the interaction between message brokers, producers, and consumers in a messaging system
  - Key Features
    - Reliable Delivery
    - Queuing
    - Routing
    - Security
    - Flow Control

- Why is AMQP Important?
  - Interoperability
  - Reliability and Durability
  - Flexibility
  - Standardized Messaging

AMQP and Other Protocols

- MQTT (Message Queuing Telemetry Transport)
- STOMP (Simple Text Oriented Messaging Protocol)
- HTTP (Hypertext Transfer Protocol)

Feature	AMQP	MQTT	STOMP	HTTP
Message Delivery Guarantees	Guaranteed delivery, persistent messages, acknowledgements	Basic delivery guarantees	Basic delivery guarantees	No guarantees for message delivery
Routing Flexibility	Complex routing with exchanges, routing keys, and bindings	Simple pub/sub model, no complex routing	Simple pub/sub model, no complex routing	No native routing, just request/response
Security	Advanced security features (authentication, encryption)	Limited security (depends on implementation)	Limited security (depends on implementation)	Limited security (depends on HTTPS setup)
Use Case	Large-scale, mission-critical, enterprise systems	IoT, low bandwidth environments	Simple messaging for web apps	Web-based applications, microservices
Protocol Complexity	More complex, feature-rich	Very simple, lightweight	Simple text-based protocol	Very simple, used for HTTP requests

AMQP and Other Protocols

- Why is AMQP better?
  - Advanced Routing
  - Reliability
  - Extensibility
- Why does RabbitMQ use AMQP?
  - Interoperability
  - Message Reliability
  - Scalability
  - Routing Flexibility

Ashish

Understanding the AMQP Message

- An AMQP message in RabbitMQ consists of two main parts
  - Message Properties – Metadata about the message.
  - Message Body – The actual data that is transmitted.
- The Structure of an AMQP Message
  - Message Properties
    - Delivery Mode
    - Message ID
    - Timestamp
    - Priority
    - Content-Type
    - Content-Encoding
    - Correlation ID
    - Reply-to
    - Expiration
    - User-Defined Headers
  - Message Body- can contain
    - Text
    - Binary Data
    - Serialized Objects

Understanding the AMQP Message

How is AMQP Message used in RabbitMQ?

- Message Routing
- Message Filtering
- Message Acknowledgement
- Dead-Lettering

Ashish

Understanding the AMQP Message

How do message properties impact message delivery?

- Persistence
- Priority
- TTL (Time-to-Live)
- Correlation ID

AMQP Message and Consumer Interaction

- Process the Message
- Acknowledge the Message
- Error Handling

Ashish

Advantages of RabbitMQ

Reliability and Durability
- Message Persistence
- Acknowledgments
- Dead Letter Queues

Scalability
- Horizontal Scaling
- Sharding
- Load Balancing

Flexibility in Routing Messages
- Exchanges
- Multiple Queues

Advantages of RabbitMQ

Rich Management Interface

- Real-Time Monitoring
- Message Tracking
- User Management

Strong Community and Ecosystem

- Extensive Documentation
- Third-Party Plugins
- Active Support

Language Support

- Java, Python, Ruby, Go, Node.js, .NET, and more

Ashish

Shortcomings of RabbitMQ

- Performance Overhead
- Complexity in Clustering
- Limited Support for Long-Running Tasks
- Scaling and Throughput Limitations
- Single Point of Failure
- Limited Features for Stream Processing

Ashish

RabbitMQ v/s Kafka

- **Message Delivery**
  - RMQ uses message queues
  - Kafka uses topic-based pub-sub semantics
- **Use Case**
  - RMQ is best for real-time message brokering
  - Kafka is primarily designed for high throughput, distributed streaming
- **Throughput**
  - RMQ throughput is lower compared to Kafka
  - Kafka is built for high throughput
- **Message Durability**
  - RMQ: messages can be made persistent
  - Kafka retains messages for a defined period
- **Scaling**
  - RMQ can scale horizontally
  - Kafka is designed for horizontal scaling
- **Protocol Support**
  - RMQ implements AMQP but also supports others
  - Kafka uses its own Kafka protocol

Ashish

RabbitMQ v/s Redpanda

- Performance
  - RMQ is known for it's reliability and flexibility
  - Redpanda is designed to use modern hardware more efficiently.
- Compatibility with Kafka
  - RMQ is not compatible with Kafka's protocol, but does support other protocols
  - Redpanda is Kafka-compatible
- Operational Simplicity
  - RMQ provides a rich management interface
  - Redpanda is designed to be simpler to manage compared to Kafka
- Storage
  - RMQ relies on disk-based storage for persistent messages
  - Redpanda uses an optimized log-structured storage model

RabbitMQ v/s ActiveMQ

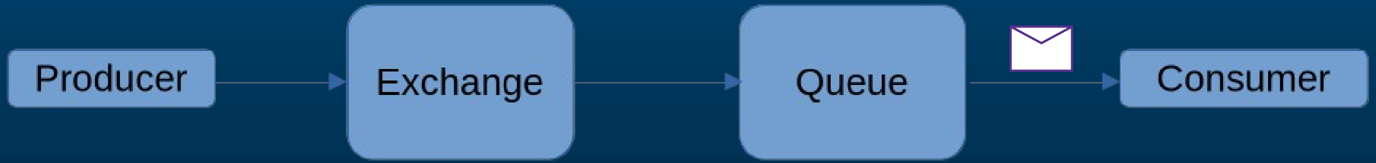
- ## Protocol Support
  - RMQ primarily supports AMQP and can also support STOMP, MQTT and other protocols
  - ActiveMQ supports JMS, AMQP, STOMP, MQTT and other protocols, it is more Java centric

- ## Performance
  - RMQ excels in task queuing and job dispatch scenarios
  - ActiveMQ provides high throughput similar to RMQ but is often considered to have more complex setup and configuration

- ## Clustering and Scalability
  - RMQ supports federation and sharding for horizontal scaling
  - ActiveMQ: scaling is often considered more complicated than RMQ’s federation

Ashish

Queues



- Asynchronous Messaging
- Decoupling
- Reliability and Durability
- Load Balancing and Scalability
- Guaranteed Delivery

Queues

Why do we need to understand Queues before RabbitMQ?

- Messaging Model
- Optimizing Queue Design
- Understanding Flow Control
- Fault Tolerance

Queues

FIFO Principal

- First In, First Out
- Types of queues
  - Simple Queue
  - Circular Queue
  - Priority Queue
  - Double-Ended Queue (Deque)

1. Message Flow Control
2. Concurrency/Parallelism
3. Fault Tolerance

Queues

In RabbitMQ, every queue is typically represented by an Erlang process.

- Each queue in RabbitMQ is managed by a dedicated Erlang process
- Messages are not represented as separate Erlang processes. Instead, they are data stored in the queue's process
- RabbitMQ's use of Erlang processes for queues allows it to handle many queues concurrently, each with its own independent state and processing logic.

Ashish

Producers-Consumers

Characteristics of a Producer

- Sends Message
- Does Not Process Message
- Does Not Need to Know Consumers

Ashish

Producers-Consumers

Characteristics of a Consumer

- Receives and Processes Messages
- Acknowledges Message Receipt
- Does Not Need to Know Producers

Producers-Consumers

Producer-Consumer Interaction

- Producer sends a message
- Message enters queue
- Consumer receives the message
- Consumer acknowledges the message
- Message is removed from the queue

Ashish

Producers-Consumers

Advantages of Producer-Consumer Model

- Decoupling of Components
- Asynchronous Processing
- Fault Tolerance
- Scalability
- Load Balancing

Ashish

Network Layers

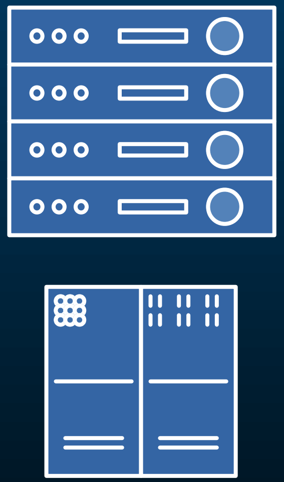
The OSI (Open Systems Interconnection) Model

- Application Layer (Layer 7)
- Presentation Layer (Layer 6)
- Session Layer (Layer 5)
- Transport Layer (Layer 4)
- Network Layer (Layer 3)
- Data Link Layer (Layer 2)
- Physical Layer (Layer 1)

Network Layers

The TCP / IP Model

- Application Layer (Layer 4)
- Transport Layer (Layer 3)
- Internet Layer (Layer 2)
- Link Layer (Layer 1)



RabbitMQ Architecture

- Exchanges
- Queues
- Bindings
- Producers
- Consumers
- Channels
- Virtual Hosts
- Connections

RabbitMQ Architecture

- Exchanges
  - Direct Exchange
  - Fanout Exchange
  - Topic Exchange
  - Headers Exchange

RabbitMQ Architecture

- Queues
  - Queue Durability
  - Queue Persistence
  - Exclusive Queues

Why Docker?

- Isolation and Environment Control
- Consistency Across Machines
- Quick Setup and Tear Down
- Clustering and Multi-node Setup
- Pre-configured Environments
- Scaling and Advanced Configurations
- Resource Efficiency

Introduction to Docker

- Docker Images
- Docker Containers
- Docker Engine
- Docker Hub and Registries
- Docker Volumes
- Docker Networks: Bridge, Host, Overlay

RabbitMQ Ports

Port	Purpose	Use Case	Protocol
5672	AMQP (default)	Communication for producers and consumers	AMQP 0.9.1 / 1.0
15672	HTTP Management UI	Web interface for RabbitMQ management	HTTP
15692	Prometheus HTTP Exporter	Exposing RabbitMQ metrics to Prometheus	HTTP / Prometheus
25672	Clustering/Inter-node comm	RMQ node-to-node clustering comm	Erlang distribution
5671	AMQP over TLS/SSL	Secure AMQP communication over TLS	AMQP over TLS/SSL
4369	EPMD (Erlang Port Mapper Daemon)	Discovery and communication between nodes	Erlang
4368	Stream Broker	Streaming messages in RabbitMQ	AMQP / HTTP
9100	STOMP over Websocket	STOMP protocol over Websockets	STOMP/Websocket
5552	Stream	Stream	Stream
5673	AMQP for another node or client	Alternative AMQP connection port for instances	AMQP

Ashish

RabbitMQ Acknowledgment Modes

Ack Modes

- Nack (Negative Acknowledgment)
- Automatic Acknowledgment
- Reject Requeue True
- Reject Requeue False

Ashish

RabbitMQ Acknowledgment Modes

Ack Mode	Description	When to Use
Nack Message Requeue True	Negative acknowledgment, message requeued for retry	Use for transient issues where retrying makes sense
Automatic Ack	Message automatically acknowledged when delivered to consumer	Use for stateless or non-critical message processing
Reject Requeue True	Message rejected but requeued for retry	Use when message needs reprocessing after failure
Reject Requeue False	Message rejected and discarded (not requeued)	Use for permanently invalid or useless messages

Ashish

Classic Queues

How Classic Queues Work

- Single Master Architecture
- Delivery Mechanism

Key Features

- Lightweight and Fast
- Support for Durability
- Flexible Routing
- Single Node Dependency

Quorum Queues

Why were Quorum Queues introduced?

- Split-Brain Issues
- Unreliable Failovers
- Data Loss
- Performance Degradation

How they work

- Replication Model
- Consensus & Fault Tolerance
- Durability and Persistence

Fault Tolerance

Consistency & Automatic Failover

Key Features

Predictable Scaling

Ideal for Critical Workloads

Quorum Queues

Use Cases

- Financial Transactions
- Order Processing
- Audit Logs and Event Tracking
- IoT and Telemetry

Trade-offs and Consideration

- Higher Disk and Memory Usage
- Increased Latency

Quorum Queues

Feature	Classic Queue	Quorum Queue
Replication	Optional (Mirrored*d)	Always (Raft-based)
Consistency	Weak (possible data loss)	Strong (data safety)
Performance	Faster	Slower due to replication
Failover	Manual (or Mirrored*d)	Automatic leader election
Best For	Speed, simple workloads	High durability, critical data

*d = deprecated

Stream

A log-structured data structure where messages are persisted in an append-only fashion and can be replayed by multiple consumers.

Key Characteristics

- Durable and Persistent
- High Throughput
- Multiple Consumers
- Event Replay

Stream

Feature	Streams	Classic Queues	Quorum Queues
Message Storage	Log-based, message stay for a set time	FIFO, messages are removed once consumed	FIFO, replicated across nodes
Performance	High-throughput	Medium	Medium
Consumer Model	Multiple consumers can read the same message at different times	Message is deleted after acknowledgment	Message is deleted after acknowledgment
Replication	Not Raft based, but supports Mirroring	None (single node)	Uses Raft-based replication
Ordering	Preserves strict ordering	Maintains FIFO order	Maintains FIFO order
Use Case	Event streaming, analytics, time series data	Traditional messaging	High availability, fault tolerance

Ashish

Exchanges

The purpose of an exchange is to **decide how messages should be routed to one or more queues**.

Types of Exchanges in RabbitMQ

- Direct Exchange
- Fanout Exchange
- Topic Exchange
- Headers Exchange
- X-Local-Random Exchange

Ashish

Exchanges

1. Producers
2. Routing Key
3. Binding
4. Routing Logic

Ashish

Exchanges

- Flexibility
- Decoupling
- Scalability
- Custom Routing Logic
- Routing Control

Ashish

Exchanges

Exchange-Queue Relationship

- Binding
- Durability
- Routing Key

Best Practices

- Use the right exchange type for your use case
- Avoid unnecessary complexity
- Leverage headers exchanges for complex routing
- Monitor exchange performance

Ashish

Messaging Patterns

Messaging patterns are predefined, reusable ways of structuring how messages are sent, received, and processed between different components in a system.

Why Are They Called "Patterns"?

The word **"pattern"** refers to a **recurring and proven solution** to a common problem in message-based communication.

Messaging Patterns

Aspect	Design Patterns (Code)	Messaging Patterns
Definition	Common solutions for structuring code	Common solutions for message flow
Used In	Object-Oriented Programming (OOP), Software Design	Message Queues, Event-Driven Systems
Examples	Singleton, Factory, Observer	Publish-Subscribe, Request-Reply
Purpose	Make code more reusable and maintainable	Make message delivery reliable and scalable

Ashish

Messaging Patterns

Simple Queue Pattern
Work/Task Queue Pattern
Pub-Sub (Fanout) Pattern
Routing Pattern
Dead Letter Pattern
Delayed Messaging Pattern
Request-Reply Pattern

Docker Network

Network Types in Docker

- Bridge Network
- Host Network
- Overlay Network
- None Network

Docker Network Drivers

- bridge
- host
- overlay
- macvlan
- none

Clustering

Node: a running RabbitMQ instance

- RAM Node
- Disk Node

How Clustering Works

- Start multiple RabbitMQ instances
- Join nodes into a cluster
- Message distribution
- Failover handling

Common Cluster Configurations

- Single Cluster
- Fedrated Cluster

Plugins

Why Use Plugins

- Extend Functionality
- Customization
- Seamless Integration
- Community Contributions

Types of Plugins

- Protocol Plugins: AMQP, MQTT, STOMP, HTTP
- Management Plugins: RMQ Management Plugin, Prometheus Plugin
- Authentication and Authorization Plugins: LDAP, OAuth2
- Federation and Clustering Plugins: Federation, Shovel
- Storage and File System Plugins: RMQ File-Based Storage Plugin
- Other Plugins: DLX Plugin, Throttle Plugin

Ashish

Shovel

A Message Replication Mechanism

Transfer messages between different RabbitMQ brokers or different virtual hosts within the same broker

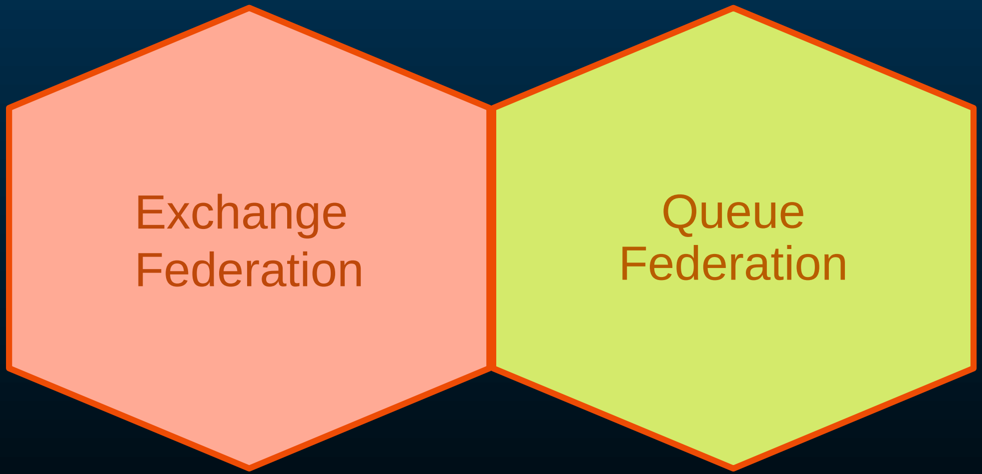
Key Concepts

- Source: The RabbitMQ server or virtual host from where messages are being sent.
- Destination: The RabbitMQ server or virtual host where the messages are transferred to.

Federation

Key Concepts

- Upstream Broker
- Downstream Broker
- Federation Links
- Federated Exchange
- Federated Queue
- Policy



Federation

Feature	Exchange Federation	Queue Federation
Pull or Push?	Pulls messages when needed	Pulls messages when needed
Scope	Works at the exchange level	Works at the queue level
Message Flow	Messages are routed to the federated exchange and then to queues	Messages are replicated directly into the federated queue
Best Use Case	Distributing messages to multiple brokers	Mirroring a queue from one broker to another

Feature	Federation	Shovel
Message Transfer	On demand (pull)	Continuous (push)
Configuration	Use policies	Requires explicit configuration
Use Case	Distributing messages between multiple brokers	Moving messages from one queue to another

Ashish

Federated Exchange

Why Use Federated Exchanges?

- Multi-Cluster Communication
- Global Message Distribution
- High Availability and Fault Tolerance
- Load Distribution
- Hybrid Cloud or Multi-Datacenter Setup

Key Features

- On-Demand Message Forwarding
- Loop Prevention
- Supports Multiple Upstreams
- Works with All Exchange Types

Ashish

Federated Exchange

- Upstream Exchange
- Downstream Exchange
- Federation Link
- Policies and Parameters

Policies

Few Policies

- Message TTL (Time to Live)
- Max Length
- Max Priority
- Federation

Use Cases

- TTL for Queues/Exchanges
- Dead Lettering
- Access Control
- Federation

Ashish

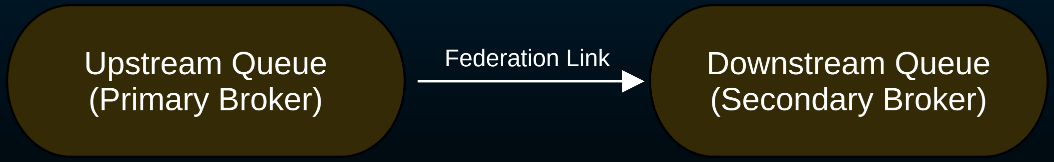
Federated Queues

Use Cases

- Geographically Distributed Systems
- Load Balancing Across Multiple RabbitMQ Clusters
- Disaster Recovery and High Availability
- Controlled Message Flow

How Federated Queues Work

- Upstream Broker
- Downstream Broker
- Federation Link
- Consumer Requests



Ashish

Federated Queues

Advantages

- Messages are only retrieved when consumers request them.
- Reduces network traffic compared to queue mirroring.
- Helps balance load dynamically across brokers.
- Ensures message availability across data centers.

Limitations

- Introduces latency since messages are fetched on demand.
- Requires a stable network connection between brokers.
- Does not support real-time mirroring of queues.

Ashish

RPC (Remote Procedure Call)

- Client sends a message to an RPC queue
- Server (worker) processes request, sends back response
- Client waits for response in temp queue

Steps to implement RPC

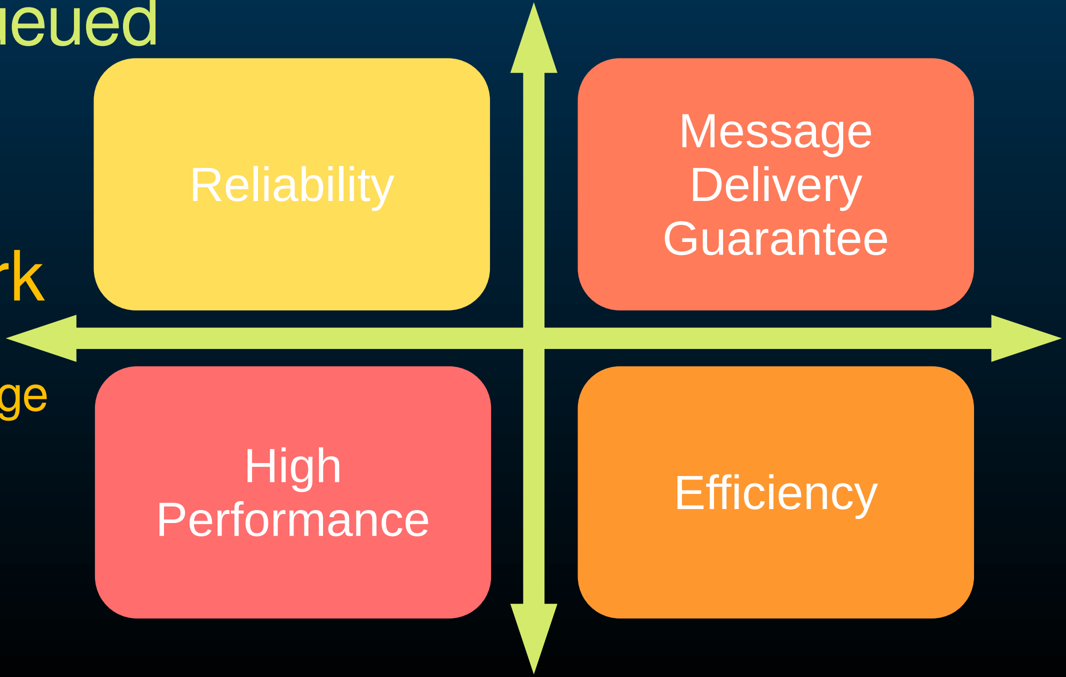
- Client sends a request with:
    - A reply-to queue (temporary queue to receive response)
    - A correlation ID (to match request and response)
- RPC Server (Worker) listens for requests
    - Computes the result and sends it to the *reply-to* queue
- Client receives the response and matches it with correlation ID

Publisher Confirms

Allows the publisher (the sender) to receive acknowledgment from RabbitMQ that a message was successfully received and queued

How Publisher Confirms Work

- Publisher sends a message
- RabbitMQ Acknowledges the Message
- Handling Failures



Publisher Confirms

When to use Publisher Confirms

- Message Delivery Reliability
- High-Volume Systems
- In Distributed Systems

Transactional Messaging

- Blocking Behavior
- Slower Performance
- Not Recommended for High-Throughput Systems
- Only One Transaction at a Time Per Channel

When to use Transaction-Messaging

- When data consistency is critical
- When you want to ensure atomic publishing of multiple messages
- In low-throughput systems where performance is not a concern
- When you want to avoid writing custom retry logic

Ashish

Comparison

Feature	Publisher Confirms	Transactional Messaging
Speed	Fast, Non-blocking	Slow, Blocking
Acknowledgment	Per message or batch confirmation	All or Nothing (Atomicity)
Reliability	High	Very High (but expensive)
Scalability	Good	Poor
Production Usage	Common	Rare

Ashish

Feature	RabbitMQ Streams	Kafka
What is it?	**Stream plugin** in RabbitMQ (added from v3.9+) to support **log-based, append-only** message storage & streaming.	**Native core model** - Kafka itself is designed from the ground up as a distributed, log-based streaming system.
Architecture	Adds **streaming capability** to RabbitMQ alongside traditional **queues & exchanges**.	Kafka is **only a log-based broker**; no traditional queues. It's always streaming/log-based.
Message Model	**Per-partition (Stream)** log, **ordered**, immutable sequence of messages. Consumer offset is stored manually or automatically.	**Partitioned logs** - each topic is split into partitions. Consumers track their own offset.
Protocol	Uses **AMQP 0-9-1** (for traditional queues) and a **binary stream protocol (TCP, port 5552 by default)** for Streams.	Uses **Kafka proprietary protocol** over TCP (port 9092 by default).
Persistence & Storage	**Disk-based storage.** Streams can hold **millions of messages** without impacting memory.	Kafka's entire architecture is **disk-backed**. Messages are always persisted on disk.
Ordering Guarantees	**Per-Stream ordering.** All consumers of a Stream see messages in order.	**Per-partition ordering.** Ordering is guaranteed within a partition, not across partitions.
Consumer Offset Management	Supports both **Automatic & Manual offset tracking**. Offset is stored **server-side**.	Consumers are responsible for managing offset, usually stored in **Kafka's internal topics** (e.g., `__consumer_offsets`).
Scalability	Streams can **scale within a cluster**, but **not as horizontally scalable as Kafka** (yet). Super Streams can help.	**Highly scalable.** Kafka clusters can handle thousands of partitions and consumers.
Fault Tolerance	RabbitMQ clustering & stream replication. Still maturing compared to Kafka's mature multi-broker replication.	**Very high fault tolerance.** Uses **replication factor** and leader election at partition level.
Use Case	Best when you need **both traditional queues and streaming in one broker**. Easy integration with existing RabbitMQ setup.	Designed for **high-throughput, real-time event streaming, big data pipelines.**
Consumer Model	**Pull-based.** Consumer requests data from a specific offset.	**Pull-based.** Consumers read from specific offsets.
Performance	**Very fast, lightweight.** 1 million+ messages per second possible.	**High throughput.** Handles millions of events per second in production-grade clusters.
Ecosystem	RabbitMQ Streams is **newer, limited client library support** but improving.	**Mature ecosystem.** Many client libraries, integrations, connectors, and monitoring tools.
Tooling	Uses **RabbitMQ Management UI**, CLI (`rabbitmqctl`), stream protocol libraries.	Kafka has a rich set of tools: **Kafka CLI, Kafka Connect, Confluent Control Center, etc.**

Monitoring

Performance Issues

- High Queue Lengths
- Slow Consumers
- Unacknowledged Messages
- Resource Utilization
- Dead Letter Queue

Best Practices for Profiling and Fixing Issues

- Regular Monitoring
- Optimize Queue Configuration
- Scale Consumers
- Manage Backpressue
- Use Dead Letter Queues (DLQ)
- Resource Allocation

Ashish

Monitoring

Queue Metrics

- rabbitmq_queue_messages
- rabbitmq_queue_messages_ready
- rabbitmq_queue_messages_unacked
- rabbitmq_queue_consumers

Connection & Channel Metrics

- rabbitmq_connections
- rabbitmq_channels
- rabbitmq_connection_open_channels

Node and Cluster Health

- rabbitmq_node_mem_used_bytes
- rabbitmq_node_disk_free_bytes
- rabbitmq_node_running (0 or 1)

Message Throughput

- rabbitmq_queue_messages_published_total
- rabbitmq_queue_messages_delivered_total
- rabbitmq_queue_messages_ack_total

Heartbeat & TCP Keepalive

Heartbeat

- Detect Connection Failures
- Keep Connection Alive
- Prevent Resource Wastage

TCP Keepalive

- Detecting Network Failures
- Long-lived Connections

Feature	Heartbeat	TCP Keepalive
Layer	Application Layer	Network Layer
Purpose	Ensures client-server application connection is alive	Ensures underlying TCP connection is alive
Frequency	Configured interval (e.g., every 30 seconds)	Typically 2 hours by default, configurable at OS level
Use Case	Detects application-level issues (e.g., client crash)	Detects network-level issues (e.g., server down)

AMQP 1.0

Feature	AMQP 0-9-1	AMQP 1.0
Protocol Model	Broker semantics with Exchanges, Queues, Bindings. AMQP 0.9. 1 defines the protocol between client and server as well as server entities such as exchanges, queues, and bindings.	General message transport protocol. AMQP 1.0 defines only the protocol between client and server.
Supports Classic, Quorum, Stream Queues	✅ Yes	❌ No (not exposed)
Supports Exchange types (direct, fanout, topic)	✅ Yes	❌ No (concept doesn't exist)
Advanced Features (DLX, TTL, Headers, etc.)	✅ Fully supported	❌ Not supported
Federation, Clustering, Streams	✅ Available	❌ Not visible
Purpose	High-level message broker with full feature set	Interoperability protocol only

Stream Filtering

- IoT Data Processing
- Real-Time Stock Market Data
- Personalized News Feeds
- Log & Event Monitoring (SIEM Systems)
- E-Commerce Order Processing

high-volume, categorized, or personalized data streams

Microservices

- Decentralized Services
- Independent Deployment
- Domain-Driven Design
- Technology Agnostic
- Resilience and Fault Tolerance

Advantages of Microservice

- Scalability
- Flexibility
- Faster Deployment
- Fault Isolation
- Easier Maintenance

Microservices

Challenges of Microservices

- Complexity
- Distributed Transactions
- Latency
- Service Discovery

Example

- User Service
- Product Service
- Order Service
- Payment Service
- Notification Service

Event Driven Architecture (EDA)

Events
Event Producers
Event Consumers
Event Brokers
Event Processing

Advantages of EDA

- Decoupling
- Asynchronous Communication
- Real-time Processing
- Flexibility

Event Driven Architecture (EDA)

Challenges of EDA

- Eventual Consistency
- Event Handling Complexity
- Distributed Transactions

Example of EDA

- User Service
- Payment Service
- Order Service
- VIP Service