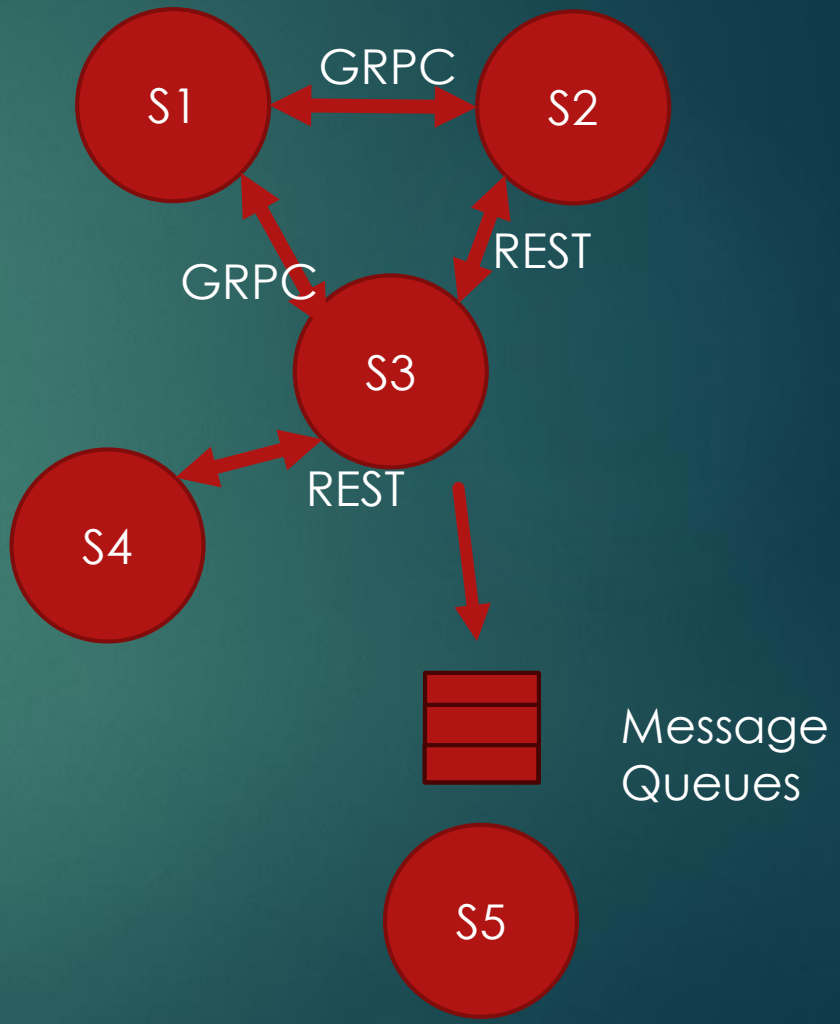
Message Queues

K V SUBRAMANIAM

Recall: SOA principle

- Build everything as a service
- Define an interface
  - Services call each other using the interface
  - Use REST/RPC for sync communication
- Asynchronous communication
  - Producers/consumers on a message queues



What are message queues

Data structure/service

Stores messages till they are processed by a consumer

Decouples producers (senders) from consumers (receivers)

Key Concepts

Producers	• Sends message to the queue
Queue	• Holds messages temporarily
Consumer	• Retrieves and processes messages
Broker	• Mangers queues and routing
Message	• Unit of data being transferred

Why use Message Queues

Decoupling

- Producers and consumers don't need to know about each other

Scalability

- Consumers can scale independently.

Reliability

- Messages can be persisted until processed

Load Balancing

- Multiple consumers can share workload

Asynchronous Processing

- Producers can continue without waiting for consume

Standard protocols

AMQP	MQTT	JMS
• Advanced Message Queuing Protocol Open standard Reliable message deliver, Queueing, Routing	• Message Queuing Telemetry Transport Lightweight publish/subscribe protocol Low bandwidth/high latency or unreliable networks (e.g IOT) Small Code Footprint	• Java Messaging Service • Java API specification • Point to point and pub/sub

AMQP – what it specifies

Message Format

- Binary message format.
- Properties (metadata –e,g: expiration, priority), Header and Body(can be in any format)

Exchange Types

Routing Mechanisms

- Keys and bindings

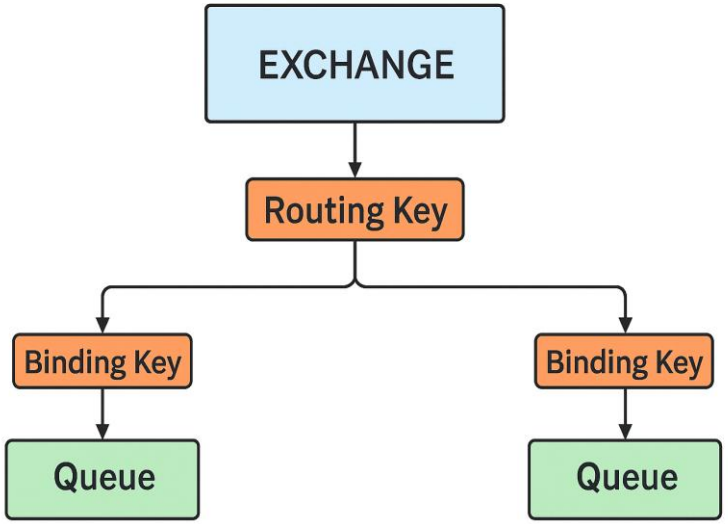
Delivery Guarantees

- At least once, at most once

Publisher consumer interactions

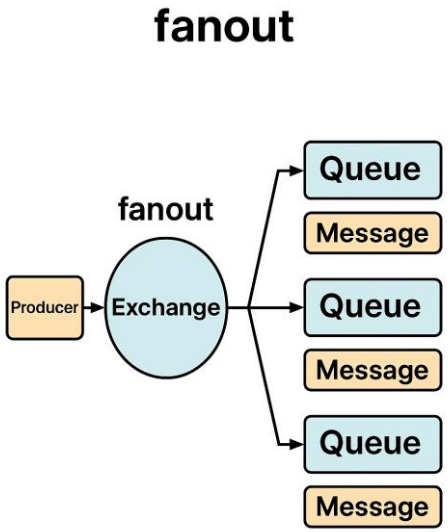
Routing Mechanism

Routing	Routing Keys • Message Attribute set by the publisher • Used by exchange to match messages to queues based on bindings • Format
Binding	Binding Keys • Associated with a queue • Acts as filter or pattern that determines which message should be routed to the queue • Exchange type determines how this is interpreted.



Exchange Types

Direct	• Routes message to queues • Based on exact match between routing key and queues binding key
Fanout	• Broadcast messages to all queues bound to exchange • Ignore routing keys
Topic	• Routes message to queues based on pattern matching • Routing keys using wildcards
Headers	• Use message headers instead of routing keys for routing decisions

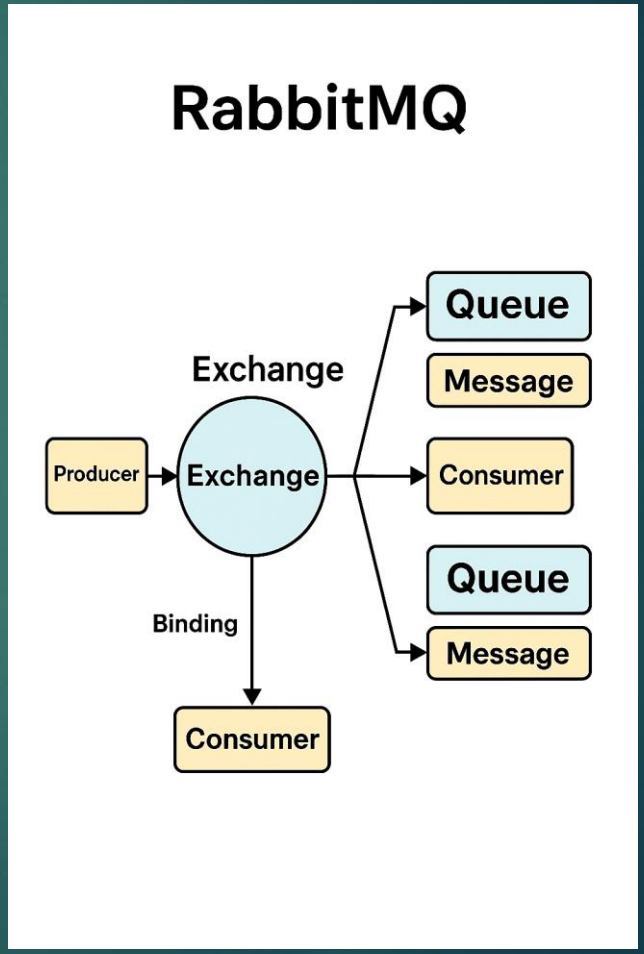


Delivery Guarantees

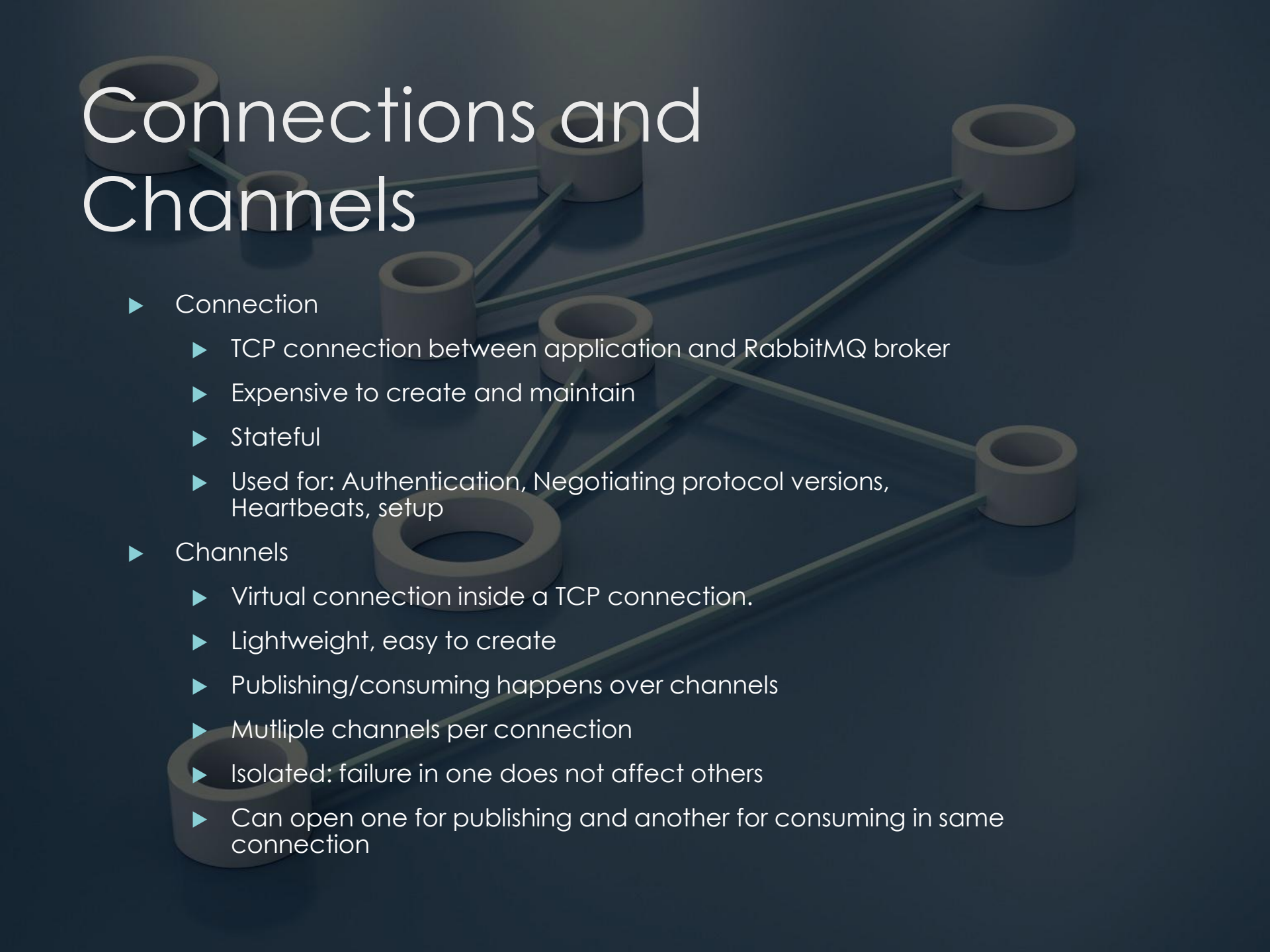
At Most Once	•Message be lost •Never Duplicated •No acknowledgement required
At least Once	•Guaranteed to be delivered •Can be duplicated •Requires Acknowledgement
Exactly One	•Ensures deliver once and only once •Requires transctional support •Less common in practice

Publisher Consumer interactions

- **Publisher Flow:**
  - Connect to broker.
  - Declare exchange (optional).
  - Publish message with routing key and optional headers.
  - Optionally wait for **publisher confirm**.
- **Consumer Flow:**
  - Connect to broker.
  - Declare queue and bind to exchange.
  - Start consuming messages.
  - Send **ack** after processing (or **nack/reject** if failed).



Connections and Channels



- Connection
  - TCP connection between application and RabbitMQ broker
  - Expensive to create and maintain
  - Stateful
  - Used for: Authentication, Negotiating protocol versions, Heartbeats, setup
- Channels
  - Virtual connection inside a TCP connection.
  - Lightweight, easy to create
  - Publishing/consuming happens over channels
  - Mutliple channels per connection
  - Isolated: failure in one does not affect others
  - Can open one for publishing and another for consuming in same connection

Examples of Message Queue Systems

RabbitMQ

- Implements AMQP, supports routing, reliability and plugins

Apache Kafka

- Distributed log based queue, great for high throughput streaming

Amazon SQS

- Fully cloud based queue

ZeroMQ

- Lightweight messaging library. Not a broker.

Google Pub/Sub

- Cloud managed, async messaging, event driven using topics

Azure Service Bus

- Equivalent to SQS. Supports queues and topics.

Lets try it out

Startup a docker container for RabbitMQ

```
docker run -d --hostname rabbit-host --name rabbitmq \
  -p 5672:5672 -p 15672:15672 \
  rabbitmq:3-management
```

5672 is the AMQP port to be used by clients

15672 – management UI at http://localhost:15672

Pip install pika (python library for AMQP)

Multiple send and receive files shared as a zip file.

Try out direct, topic and fanout routing using the corresponding send and receive files.

For Fanout, run multiple copies of receiver and then do a send.

Video Watch

- What we learnt from running thousands of production RabbitMQ clusters
  - https://www.youtube.com/watch?v=ez9kQEhHsnc