TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Communication in Distributed Systems
Message Oriented Middleware

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2025/26

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Message-oriented communication

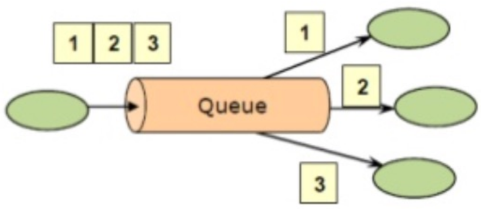
- RPC provides distribution transparency over socket programming
- But still coupling between interacting entities
  - Temporal: caller waits for a reply
  - Spatial: shared data
  - Functionality and communication are tightly coupled
- How to improve decoupling and flexibility?
- Message-oriented communication
  - Transient
    - Berkeley sockets, Message Passing Interface (MPI)
  - Persistent
    - **Message Oriented Middleware (MOM)**

Message-oriented middleware

- Communication middleware that supports sending and receiving messages in a persistent way
  - Provides intermediate-term storage for messages
- Loose coupling among system/app components
  - Supports temporal and spatial decoupling
  - Can also support synchronization decoupling
- Goals: increase performance, scalability, and reliability
  - Commonly used in serverless and microservice architectures
- Communication patterns:
  - **Message queue**
  - **Publish-subscribe** (pub/sub)
- Related systems:
  - **Message queue system** (MQS)
  - **Pub/sub system**

Message queue pattern

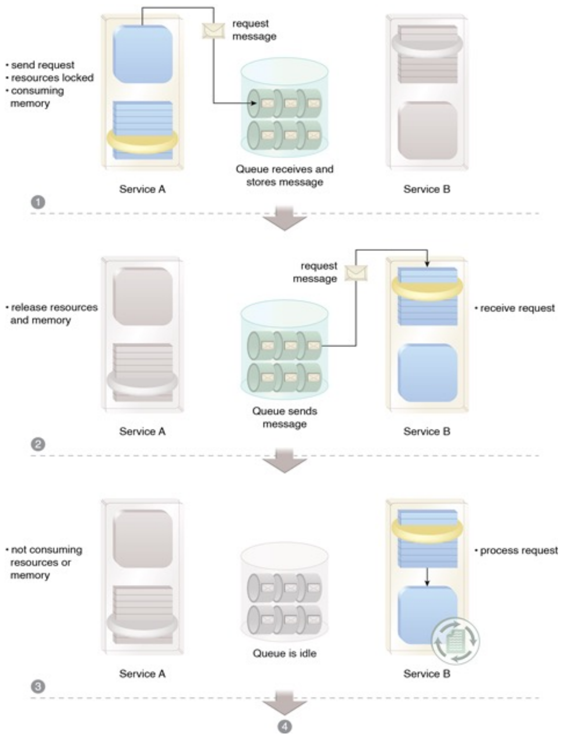
- Messages sent to a queue are stored until retrieved by a consumer
- Multiple producers can send messages to the queue
- Multiple consumers can receive messages from the queue
- Communication is one-to-one: each message is delivered to a single consumer



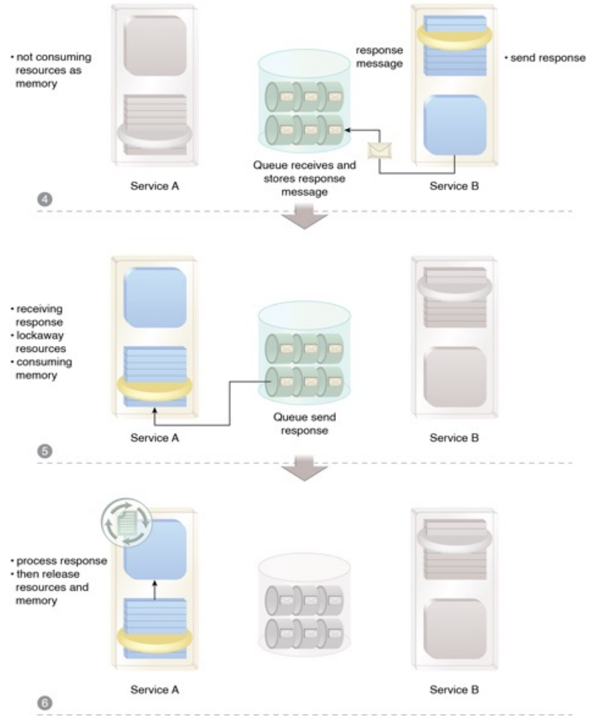
- Use cases:
  - Task scheduling, load balancing, logging or tracing

Message queue pattern

A sends a message to B



B processes the message and sends a response back to A

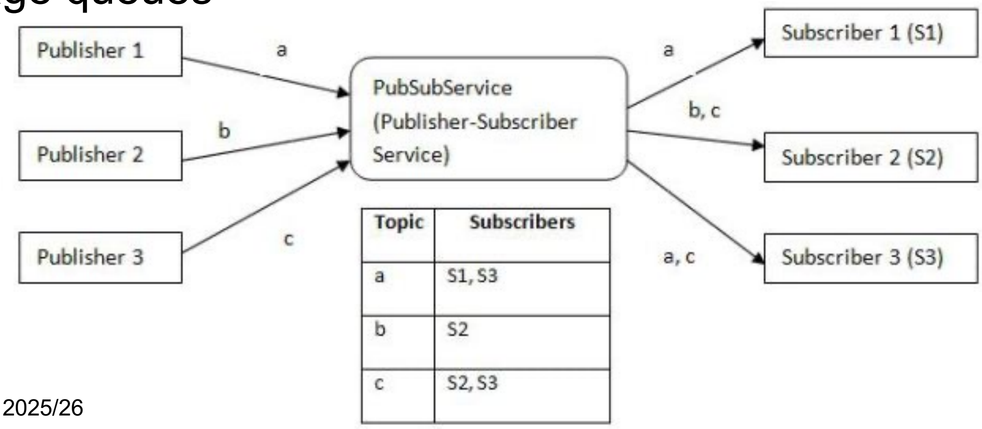


Message queue API

- Typical calls in MQS:
  - **put**: non-blocking send
    - Inserts a message into the queue immediately
  - **get**: blocking receive
    - Waits until the queue is non-empty and retrieve a message
    - Variant: can search for a specific message in the queue
  - **poll**: non-blocking receive
    - Checks the queue and retrieves a message if available
    - Never blocks
  - **notify**: non-blocking receive
    - Installs a callback handler that is automatically invoked when a message arrives in the queue

Publish/subscribe pattern

- Producers (publishers) send asynchronous messages, such as event notifications
- Consumers (subscribers) express interest in specific topics by subscribing
- Decoupling in time and space: publishers and subscribers do not need to interact directly
- Supports one-to-many communication, unlike message queues



Publish/subscribe pattern

- Multiple consumers can subscribe to a topic, with or without filters
- An *event dispatcher* collects subscriptions and route messages to all matching subscribers
  - For scalability, the dispatcher is typically distributed
- High degree of decoupling among components
  - Components can be easily added or removed: ideal for dynamic environments
- Use cases:
  - Event notification, multicasting
- Comparison with message queue

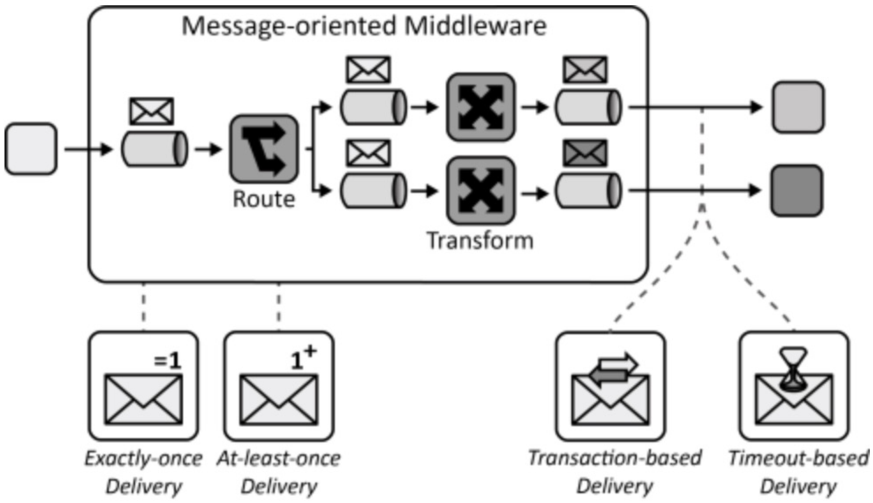
Feature	Message queue	Pub/sub
Delivery	One-to-one	One-to-many
Consumers	Single consumer per message	Multiple consumers per message

Publish/subscribe API

- Typical calls in a pub/sub system:
  - `publish(event)`: called by a publisher to publish an event
    - Events can be any data type and may include meta-data
  - `subscribe(filter_expr, notify_cb, expiry) → sub_handle`: called by a subscriber to subscribe to events
    - Inputs: filter expression, reference to a notification callback, and subscription expiry time
    - Returns a subscription handle
  - `notify_cb(sub_handle, event)`: called by pub/sub system to deliver to a matching event to subscribers
  - `unsubscribe(sub_handle)`: called by the subscriber to remove the subscription

MOM functionalities

- MOM simplifies distributed communication by handling:
  - Addressing: locating the recipient(s) for messages
  - Routing: delivering messages along appropriate paths
  - Availability of application components (or applications)
  - Message format transformations

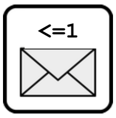


https://www.cloudcomputingpatterns.org/message_oriented_middleware

MOM functionalities

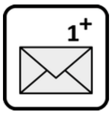
- Let's analyze
  - Delivery semantics
  - Delivery model
  - Message routing
  - Message transformations

MOM delivery semantics



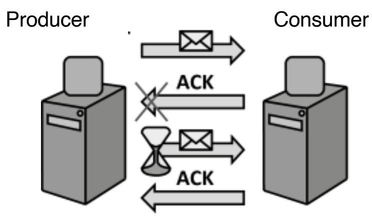
At-most-once delivery

- Each message is delivered no more than once
- Messages may be lost, but are never redelivered



At-least-once delivery

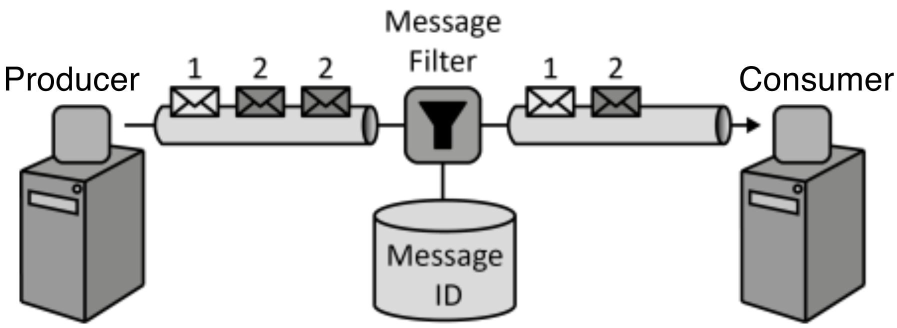
- Messages are never lost, but they may be delivered multiple times
  - Since messages can be duplicated, the consumer-side application must be able to handle duplicate processing (or be sure that re-processing the same message does not cause errors)
- Mechanism*:*
  - Consumer sends ack for each message; MOM resends message if ack is missing



MOM delivery semantics

Exactly-once delivery 

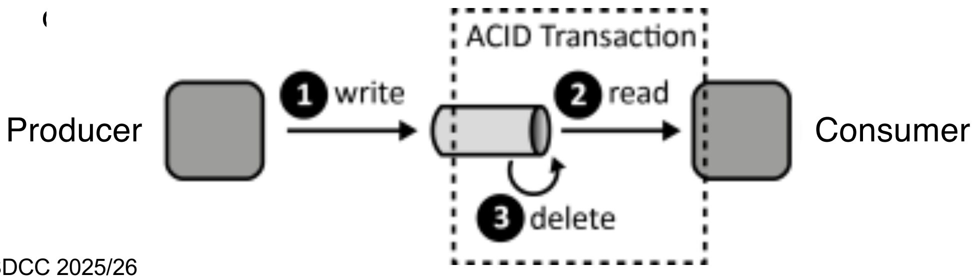
- Message is delivered exactly once to consumer
- Mechanisms
  – Deduplication logic: MOM filters message duplicates
    - Each message has a unique ID
  – Idempotent consumers: can safely process the same message multiple times
  – Messages survive MOM failures



MOM delivery semantics

Transaction-based delivery 

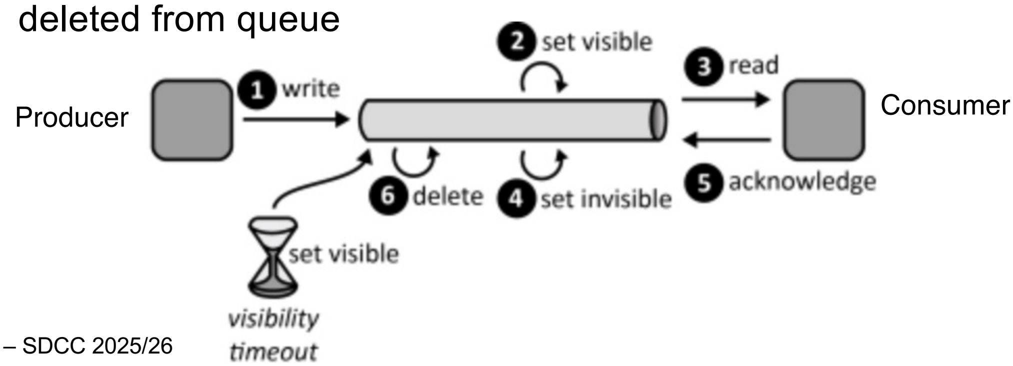
- Guarantees messages are deleted from the queue only after they have been successfully received and processed by the consumer
- Mechanism
  – MOM and consumer participate in a transaction: read and delete operations are part of the transaction, ensuring ACID properties
    - Commit: if successful, the message is permanently deleted
    - Rollback: if the transaction fails, the message remains in the



MOM delivery semantics

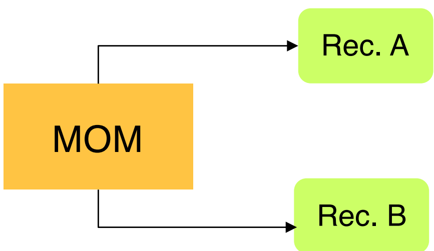
Timeout-based delivery

- Ensures messages are delivered at least once while preventing duplicate consumption
- Mechanism
  - Message is not deleted immediately from the queue, but marked as invisible until *visibility timeout* expires
  - Invisible message cannot be read by another consumer
  - After consumer's ack of message receipt, message is deleted from queue

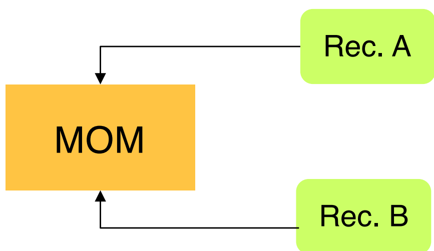


MOM delivery model

- How messages are retrieved by subscribers or consumers
- Options: push vs. pull delivery
- Push: MOM actively notifies the receiver(s) when a message is available
- Pull: receiver(s) periodically polls MOM for new messages



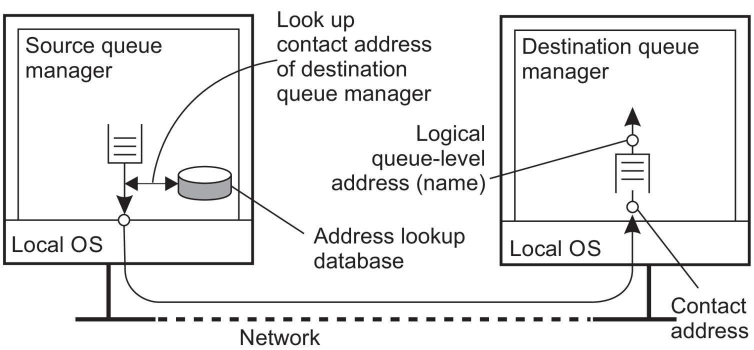
Push delivery



Pull delivery

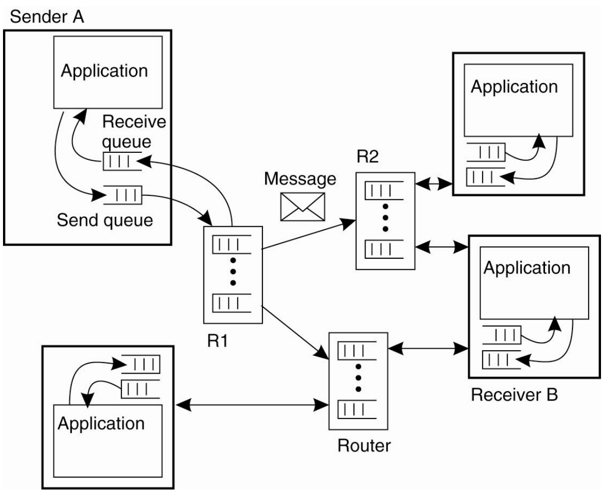
Message routing in MOM

- Queues are managed by queue managers (QMs)
  – Applications can only put messages into a local queue
  – Applications can only retrieve a message from a local queue
- QMs are responsible for routing messages between local queues and remote queues
  – Act as message-queuing relays, communicating with distributed applications and other QMs
  – Form an overlay network
  – Some QMs may operate only as routers, without storing messages



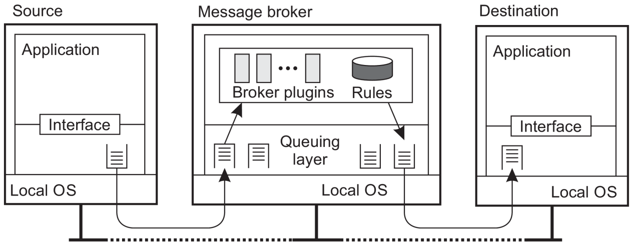
Message routing: overlay network

- Overlay network for message routing
  – Each QM maintains a routing table to map queue names to their location
- Overlay network maintenance
  – Static overlays: routing tables are set up and managed manually
    - Easier to configure but less flexible
  – Dynamic overlays: routing tables are updated at runtime
    - More complex but supports scalable and adaptive routing



Application heterogeneity: message broker

- Problem: new and existing apps rarely share a common data format, how to integrate them into a single, coherent system?
- Solution: ***message broker***
  - Handles application heterogeneity in MOM
- Responsibilities
  - Message transformation: uses a repository of conversion rules to transform message types
  - Content-based routing: routes messages based on content or type
- Scalability and reliability: usually implemented in a distributed manner



MOM frameworks

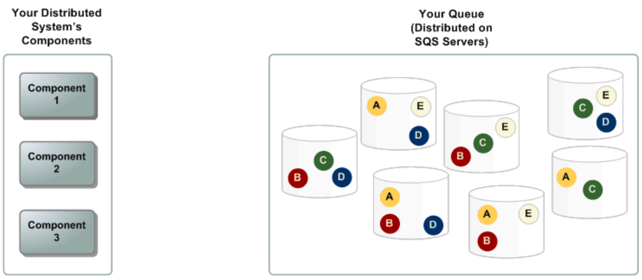
- Main MOM systems and libraries
  - Apache ActiveMQ https://activemq.apache.org
  - **Apache Kafka**
  - Apache Pulsar https://pulsar.apache.org
  - IBM MQ https://www.ibm.com/products/mq
  - NATS https://nats.io
  - **RabbitMQ**
  - Redis https://redis.io/
  - ZeroMQ https://zeromq.org
- Note: distinction between queue-based and pub/sub patterns is sometimes blurred
  - Some frameworks support both patterns, e.g., NATS, Pulsar and Kafka (with specific configurations)
  - Others only one, e.g., Redis is pub/sub only, ZeroMQ is queue only

MOM frameworks

- Cloud-based MOM services
  - **Amazon Simple Queue Service (SQS)**
  - Amazon Simple Notification Service (SNS)
  - CloudAMQP: RabbitMQ as a Service
  - Google Cloud Pub/Sub
  - Microsoft Azure Service Bus
- Note
  - Cloud MOMs abstract infrastructure management, allowing developers to focus on messaging patterns
  - Many support both queue and pub/sub patterns

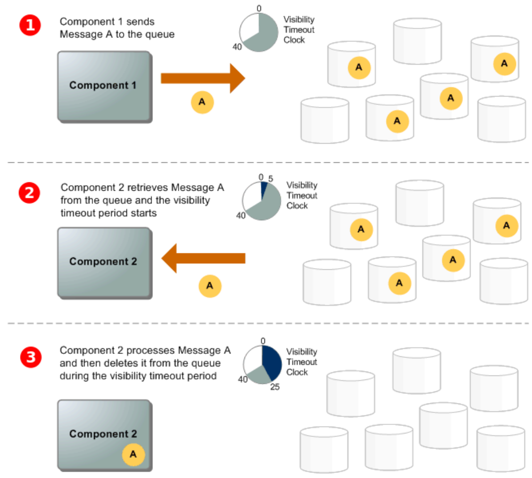
Amazon Simple Queue Service (SQS)

- Reliable, scalable cloud-based message queue service
  - Designed to decouple application components, allowing them to run independently and asynchronously
  - Supports integration across different languages and technologies
- Architecture
  - Message queues fully managed by AWS
  - Queues are distributed across multiple SQS servers
  - Replication within a region: messages stored on multiple servers for high availability
  - Pull delivery model: consumers poll the queue to retrieve messages



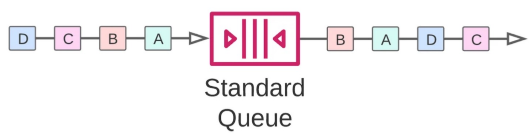
Amazon SQS: Message handling

- Consumer must delete message after processing
  - Queue = temporary holding location, not permanent storage
  - Message retention period: configurable (max 14 days)
- Implements timeout-based delivery
  - When a consumer retrieves a message, it remains in the queue but becomes invisible to others (visibility timeout)
  - If processing succeeds, the consumer deletes the message
  - If processing fails, the timeout expires and the message becomes visible again



Amazon SQS: Polling and queue types

- Consumers poll the queue to receive messages
  - *Short* polling: queries only a subset of SQS servers (may return empty)
  - *Long* polling: queries all servers, wait for messages (reduces empty responses)
- Queue types: standard or FIFO
- Standard queue (default)
  - Best-effort ordering: messages may arrive out-of-order
  - Duplicates possible



- FIFO queue
  - Guarantees in-order delivery (sent=processed order)
  - No duplicates
  - ✗ Lower throughput

Amazon SQS: API

- `CreateQueue`, `ListQueues`, `DeleteQueue`
  – Create a new queue, list all queues, delete an existing queue
- `SendMessage`
  – Add a message to queue (message size up to 256 KB)
  – For larger payloads: store on S3 and send an S3 reference instead
- `ReceiveMessage`
  – Retrieve one or more messages
  – Retrieve up to 10 messages: can specify the maximum number of messages to fetch (from 1 to 10)
  – No filtering before delivery
  – Post-delivery filtering
    - If a consumer wants to filter messages, it must inspect message body or attributes after receiving the messages and decide whether to process or discard them

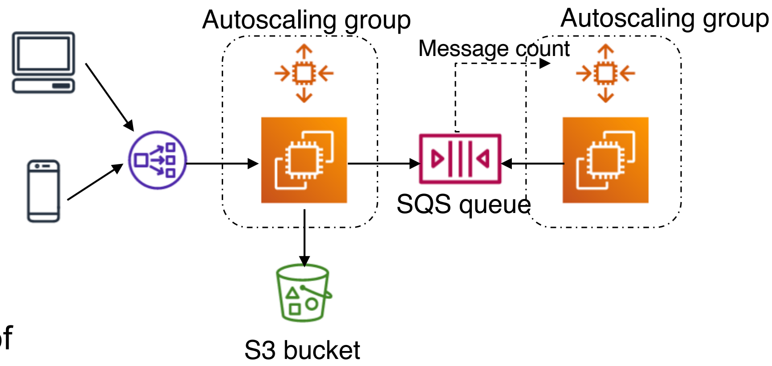
https://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/Welcome.html

Amazon SQS: API

- `DeleteMessage`
  – Remove a specified message from the queue after processing
- `ChangeMessageVisibility`
  – Change the visibility timeout of a specific message in the queue
  – A message remains in the queue until explicitly deleted by the consumer
  – Default visibility timeout: 30 sec.
- `SetQueueAttributes`, `GetQueueAttributes`
  – Modify queue attributes (e.g., message retention), retrieve current queue settings

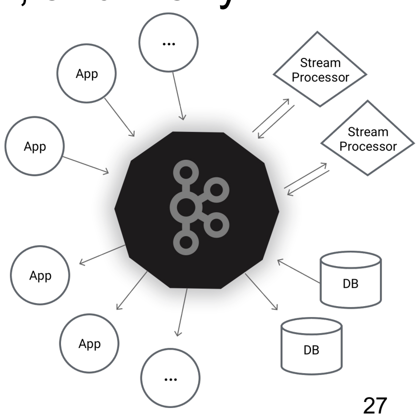
Amazon SQS: example

- Photo processing service using SQS
  - Goal: use SQS to decouple the front-end and back-end components, ensuring load balancing and fault tolerance
  - Front-end uploads a photo to S3 and sends to back-end an SQS message containing the S3 link
  - A pool of EC2 instances (back-end) retrieves messages from the queue
    - Each instances gets a photo, processes it (e.g., resize) and delete the message after processing
    - If processing fails, the message becomes visible again after the visibility timeout, allowing another instance to retry
    - The back-end pool can be scaled horizontally based on the number of messages in the queue

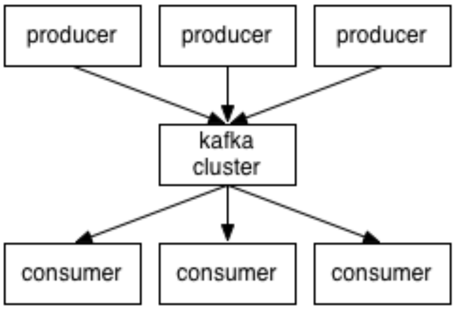


Apache Kafka

- Open-source, distributed event streaming platform
  https://kafka.apache.org/
- Designed to
  - Publish and subscribe to streams of events
  - Store event streams durably and reliably
  - Process event streams
- Started by LinkedIn in 2010, now Apache project
- Used at scale by LinkedIn, Netflix, Uber, and many others
- Written in Java and Scala
- Horizontally scalable and fault-tolerant
- High-throughput ingestion
  - Billions of events



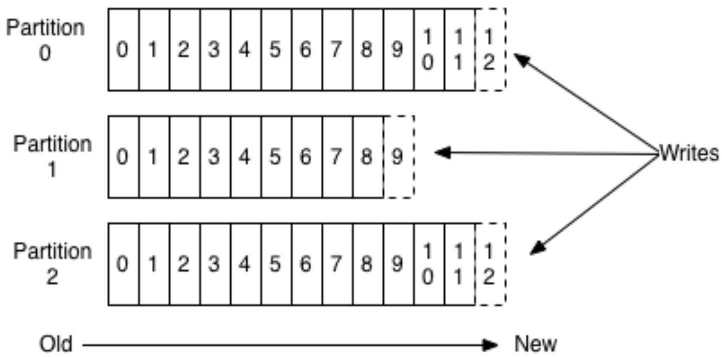
Kafka at a glance



- **Kafka topics**: Kafka stores streams of events (aka *messages* or *records*) in categories known as topics
- **Kafka cluster**: composed of servers called brokers, which can span multiple data centers or cloud regions
  - Brokers receive and store events
- **Producers**: publish (write) events to a Kafka topic
- **Consumers**: subscribe to Kafka topics, read published events and process them
  - A topic can have 0, 1, or many subscribing consumers

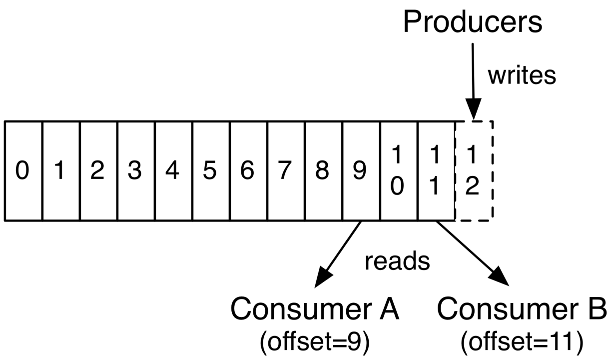
Kafka: topics and partitions

- Topic: category to which events are published
- For each topic, Kafka cluster maintains a **partitioned log**
- Log (data structure): *append-only, totally ordered* sequence of events
- Partitioned log: each topic is split into a pre-defined number of partitions
  - Partition: the unit of parallelism for a topic



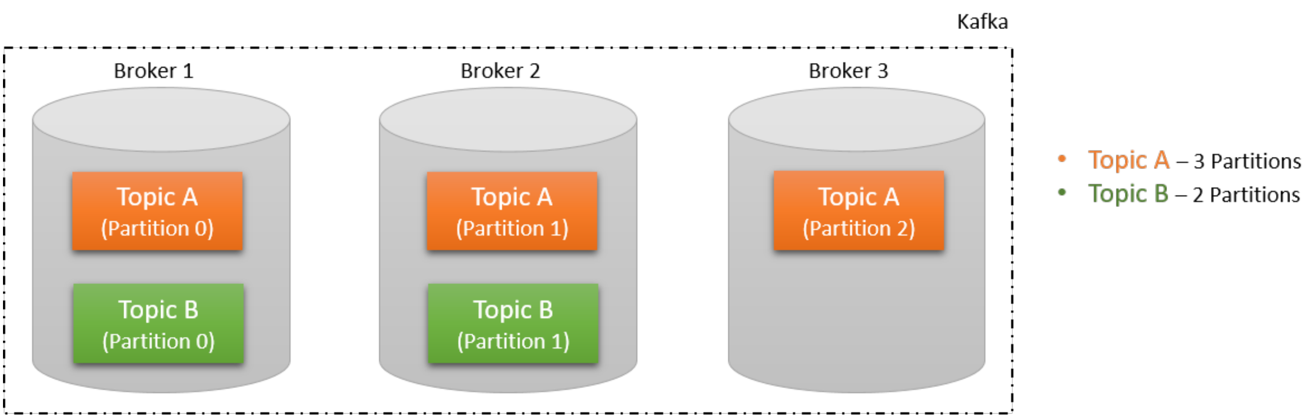
Kafka: partitions and parallelism

- Parallelism and scalability come from having multiple partitions per topic
- Producers publish events to topic partitions
- Consumers read events from the topic
- Each partition is:
  - A numbered, ordered, immutable sequence of records
  - Appended to, never modified (immutable records)
  - Similar to a commit log
- Every record has a monotonically increasing **offset**
  - Uniquely identifies each record within a partition



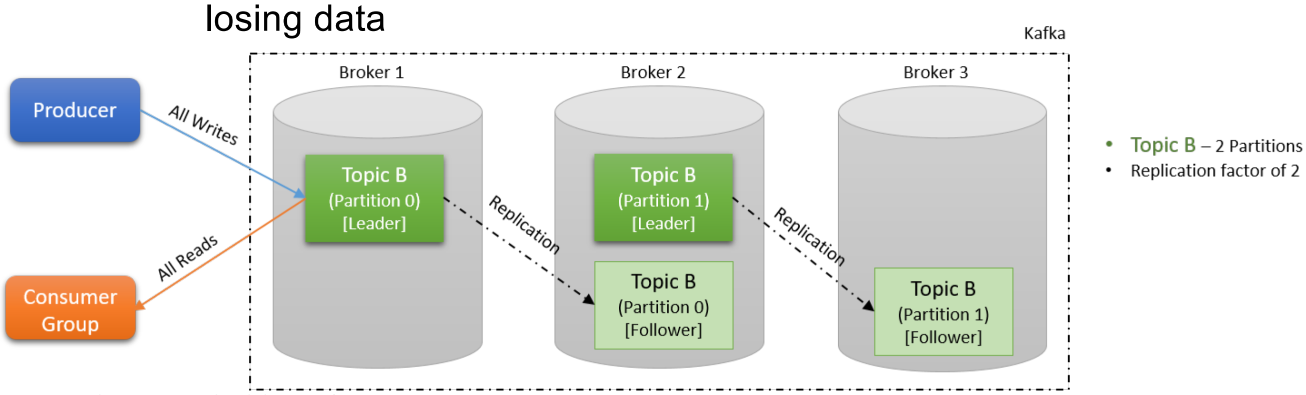
Kafka: partitions and design choices

- Improve scalability: topic partitions are distributed across multiple brokers
  - ✓ I/O throughput increases through parallel reads and writes
    - Multiple producers can write concurrently to different partitions
    - Multiple consumers can read concurrently from different partitions



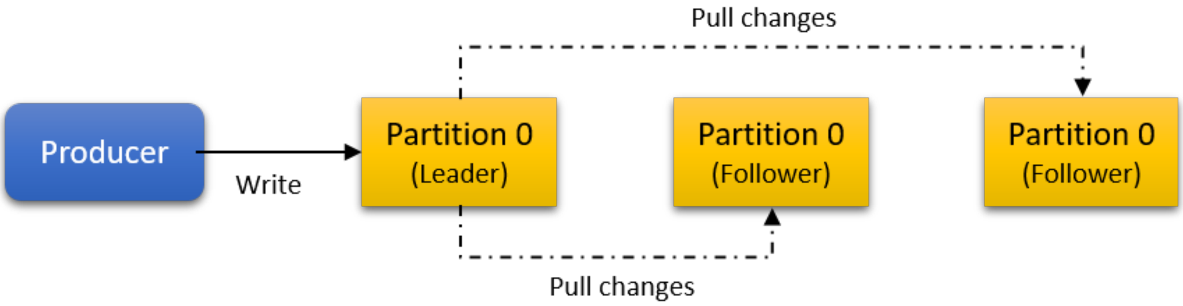
Kafka: partitions and design choices

- Improve fault tolerance: partitions can be *replicated* across brokers
  - Each partition has:
    - 1 leader: handles all writes and reads
    - 0 or more followers: replicate the leader data
- *Replication-factor* = total number of replicas (including leader)
  - Replication-factor = *N* → up to *N*-1 brokers can fail without losing data



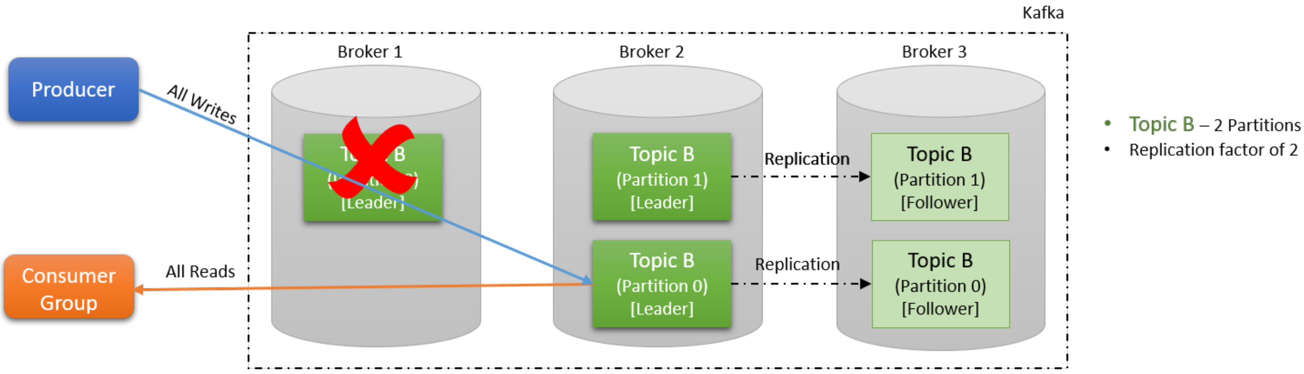
Kafka: partitions and design choices

- Simplify data consistency management: only leader handles all reads and writes
  - Producers write to the leader, consumers read from the leader
  - Followers replicate the leader and serve as backups
  - Followers can be:
    - in-sync: fully updated with the leader
    - out-of-sync: lagging behind the leader



Kafka: partitions and design choices

- Share responsibility and balance load: each broker acts as leader for some partitions and follower for others
- Coordination across brokers is managed by Apache Zookeeper or KRaft
  - Handles leader election
  - Maintains cluster metadata

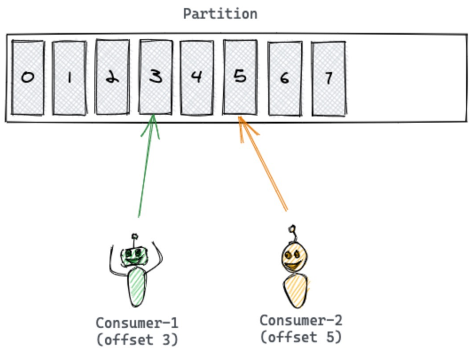


Kafka: producers

- Producers = data sources
- Publish events to topics of their choice
  - Send events directly to the broker that is leader for the target partition (no routing tier)
- Responsible for assigning events to partitions using:
  - Key-based partitioning
    - Event contains a partition key (e.g., order ID, user ID)
    - Partition is chosen by hashing the key
    - Example: if `key=user_id`, all events for the same user go to the same partition
  - Round-robin partitioning
    - Default if no key is specified
  - Custom partitioning
    - A partitioner can be provided to implement a custom partitioning logic
- Multiple producers can write concurrently to the same partition

Kafka: consumers

- Kafka uses a pull-based delivery model for consumers
  https://kafka.apache.org/documentation.html#design_pull
- A consumer track its progress using offset
  – Offset identifies which events have been consumed
- Multiple consumers can be read the same partition, each reading at different offsets



Kafka: consumers

- Why pull?
- Push model
  – Broker actively pushes events to consumers
  ✗ Broker must manage
    - Different consumers with their needs and capabilities
    - Transmission rate control
    - Timing decisions: whether to send a message immediately or accumulate more data to send later
- Pull model
  – Consumers control event retrieval from broker
  – Consumers maintain an offset to identify the next event to read
  ✓ More scalable and flexible
    - Less burden on brokers
    - Consumers can pull events when ready, adjusting the rate
  ✗ If no events are available, consumers may end up busy waiting for events to arrive

Kafka: consumers

- How can consumers read in a fault-tolerant way?
  - After a consumer reads events, it stores its committed offset in a special Kafka topic called `__consumer_offsets`
  - In case of crash, the consumer can recover by reading events from the committed offset
  - By default, `auto-commit` is enabled

Kafka: message retention and storage

- Kafka brokers store messages reliably on disk
- Messages are stored in log segments (files on disk)
- Unlike other message queue and pub/sub systems, Kafka does not delete messages after delivery, but retains messages based on size or time
- Issue: how to free up disk space?
  - Topics are configured with retention time (how long events should be stored)
  - Upon expiry, events are marked for deletion
  - Alternatively, retention can be specified in bytes
- On each broker, messages are flushed from the broker's in-memory buffer to the disk
  - A log flush policy controls when messages are written to disk (time interval and number of messages)

Hands-on Kafka

- Preliminary steps
  - Download and install Kafka
    https://kafka.apache.org/downloads
    - Configure Kafka properties in `server.properties`
      - E.g., `listeners` which are network interfaces on Kafka broker: `listeners=PLAINTEXT://localhost:9092`
      - E.g., log flush policy
  - Start Kafka environment
    - Kafka can use KRaft or ZooKeeper (we will use KRaft)
      - Alternatively, start ZooKeeper first
    - Start Kafka broker (default port: 9092)

```
$ bin/kafka-server-start config/kafka.properties
```


Hands-on Kafka

- Let's use Kafka CLI to create a topic, publish and consume events to/from topic, and delete it
- Create a topic named test with 1 partition and no replication (`replication-factor=1`)

```
$ kafka-topics --create --bootstrap-server localhost:9092
  --replication-factor 1 --partitions 1 --topic test
```

  - Bootstrap server: an initial broker used by the producer to connect to the Kafka cluster
  - It provides the producer with metadata about the cluster, including a list of all the brokers
  - Multiple bootstrap servers for fault tolerance and load balancing
  - Once the producer discovers the partition leader, it can start sending messages

Hands-on Kafka

- Produce messages into the topic

```
$ kafka-console-producer --bootstrap-server localhost:9092 \
  --topic test
>msg 1
>msg 2
```

- Consume messages from the beginning of the topic

```
$ kafka-console-consumer --bootstrap-server localhost:9092 \
  --topic test --from-beginning
```

- Consume messages from a given offset (e.g., 2) of a specific partition (e.g., partition 0) of the topic

```
$ kafka-console-consumer --bootstrap-server localhost:9092 \
  --topic test --offset 2 --partition 0
```

- Produce messages with key

```
$ kafka-console-producer --bootstrap-server localhost:9092 \
  --topic test --property parse.key=true --property key.separator=:
>course:sdcc
>university:Tor Vergata
```


Hands-on Kafka

- List available topics

```
$ kafka-topics --list --bootstrap-server localhost:9092
```

- Delete a topic

```
$ kafka-topics --delete --bootstrap-server localhost:9092 \
  --topic test
```

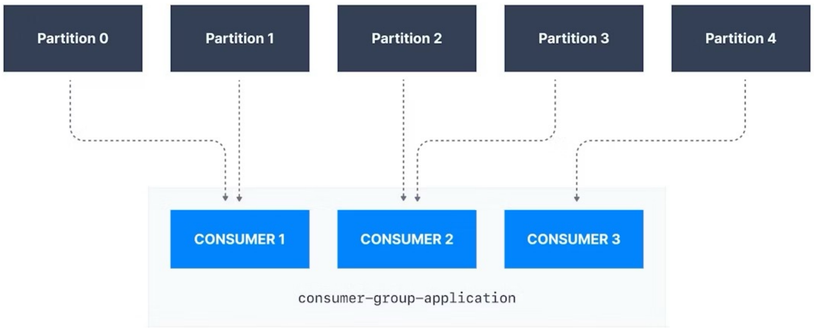
- Stop Kafka

```
$ kafka-server-stop
```

  – If Zookeper: `$ zookeeper-server-stop`

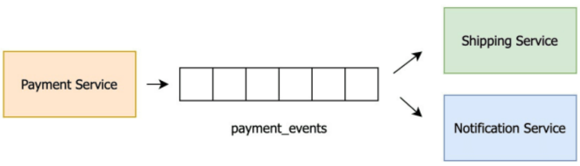
Kafka: consumer group

- Consumer Group: set of consumers that cooperate to consume data from a topic and share a group ID
  – A Consumer Group represents a *logical* subscriber
  – Topic partitions are divided among consumers in the group
    - Reassigned when consumers join/leave the group
  – Each event is delivered to *only one* consumer within the group
  – Each group maintains its own offset per topic partition

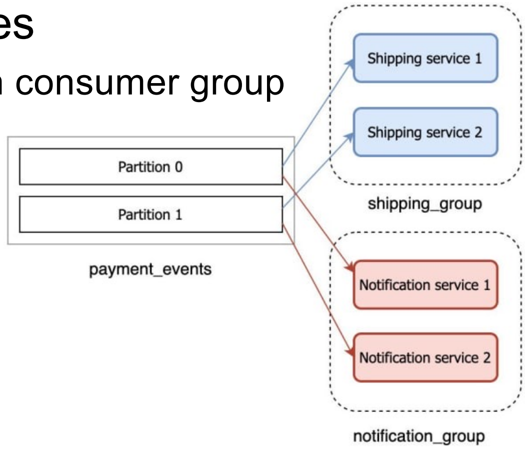


Kafka: consumer group

- How can many consumers read the same events?
  – By using different group IDs: consumers in different groups can independently read the same events from the same topic
- Example: microservices communicating through Kafka



- When we scale microservices
  – Each microservice has its own consumer group



Hands-on Kafka: consumer group

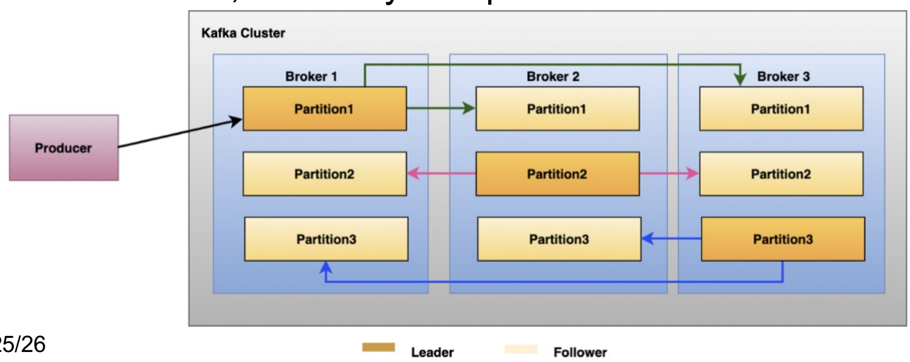
- Launch a consumer in a consumer group named `my-group`

```
$ kafka-console-consumer \
  --bootstrap-server localhost:9092 \
  --topic test \
  --group my-group
```

- Open two more terminal windows and run two additional consumers in `my-group`
  - Each consumer will be assigned to a different topic partition
- Produce messages in the topic and see how they are distributed among the consumers
  - Each consumer will read the messages from its assigned partition

Kafka: replication

- Topic partitions are replicates across brokers for fault tolerance
  - Leader-follower model: followers replicate the leader log
  - Replication is asynchronous: periodically, followers pull messages from the leader in order, using offsets
  - In-sync replicas (ISR): the leader and any followers that are updated with the leader's log
  - Durability (no message loss) is guaranteed as long as at least one in-sync replica is available
  - If the leader crashes, one in-sync replica is elected as new leader



Kafka: event ordering guarantees

- Events written by a producer to a topic partition are appended in the order they are sent

  message: m1 m2 m3 m4 m5

  offset: 10 11 12 13 15

- Consumer reads events in the same order they are stored in the partition

  m1 → m2 → m3 → m4 → m5

- Kafka provides ordering guarantees, but *only within a partition*
  - Total order of events within each partition (*per-partition ordering*)
- Kafka does not preserve order across partitions of the same topic
- In practice, per-partition ordering combined with key-based partitioning is sufficient for most applications

Kafka: write guarantees

- Controlled by `acks` and `min.insync.replicas`
- `acks` (producer setting): how many in-sync replicas must acknowledge a write before the producer considers it successful
  - `acks=0`: no ack (fire and forget)
  - `acks=1`: wait for leader only
  - `acks=all`: wait for all ISR replicas
- `min.insync.replicas` (topic/cluster setting): minimum number of ISR replicas that must acknowledge a write for it to succeed
- Strong durability: `acks=all + min.insync.replicas>=2`
  - Example: `replication-factor=3`
    - `min.insync.replicas=1`: only leader
    - `min.insync.replicas=2`: leader + 1 follower (recommended)
    - `min.insync.replicas=3`: leader + 2 followers (highest durability)

Kafka: read and visibility guarantees

- Message durability: depends on `acks` + ISR size/status
  – A write is kept only if enough in-sync replicas acknowledge it (as required by `acks` and `min.insync.replicas`)
- Message availability: consumers can read messages as soon as they are appended to the leader's log
  – Replication to followers is asynchronous
  – Exception: transactional consumers (`read_committed`)
- A write is successful only if:
  1. The leader appends the message, <u>and</u>
  2. The acknowledgment requirements (`acks` and `min.insync.replicas`) are satisfied

  – If reqs cannot be met, write is rejected (not visible to consumers)

Kafka: delivery semantics

- **At-most-once**: guarantees that events are never re-delivered but may be lost
  – Producer disables retries (i.e., `acks=0` on producer)

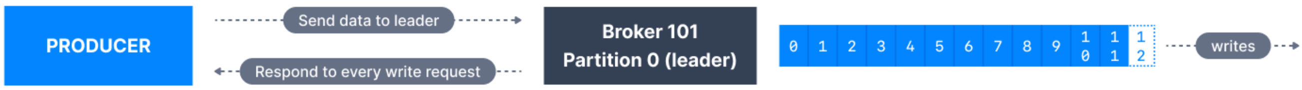
`acks=0`

  – Consumer commits its offset before processing event
    - What happens if the consumer crashes after committing but before processing?

https://kafka.apache.org/documentation/#semantics
https://docs.confluent.io/kafka/design/delivery-semantics.html

Kafka: delivery semantics

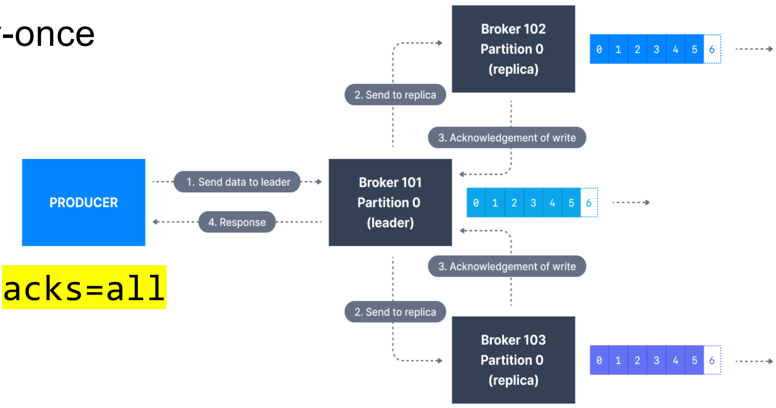
- **At-least-once** (default): guarantees no event loss, but events may be duplicated
  - Producer waits for ack only from the partition leader; if no ack, it retries
  - How? Set `acks=1` on producer
  - Consumer commits its offset after processing an event
    - What happens if the consumer crashes before committing?



`acks=1`

Kafka: delivery semantics

- **Exactly-once**: guarantees no event loss, no duplicates, but at the cost of higher latency and lower throughput
  - Producer: `acks=all`
    - Waits for ack from all in-sync partition replicas
  - Idempotent producer: detect duplicates caused by retries
    - Producer ID and sequence number are added to each event
  - Durability: `min.insync.replicas` >= 2
  - Transactions enabled: atomic writes across partitions + transactional offset commits + `read_committed` consumers
  - Not fully exactly-once



`acks=all`

Kafka: delivery semantics

- Take-away message: choose delivery semantics that fit your application
  - At-most-once: prioritize speed (e.g., telemetry, if some data loss is acceptable)
  - At-least-once: prioritize reliability (e.g., email notifications)
  - Exactly-once: prioritize correctness (e.g., financial transactions)

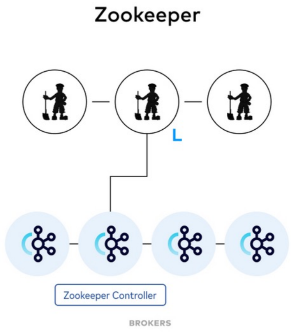
Kafka: trade-offs

- Consistency vs. availability trade-off
  - Kafka ensures strong consistency per partition: all in-sync replicas are up-to-date with the leader
  - If the leader fails and no ISR follower is available, the partition becomes unavailable

- Latency vs. durability trade-off
  - Producers control durability via `acks`
    - `acks=all`: highest durability, higher latency (waits for all ISRs)
    - `acks=1`: lower latency, but messages may be lost if leader crashes
    - `acks=0`: lowest latency, highest risk of loss
  - Lower latency increases the chance of data loss in case of failure

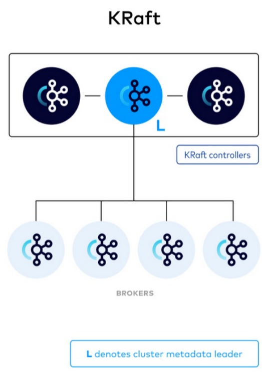
Kafka: from ZooKeeper to KRaft

- Zookeeper: distributed coordination service used for managing distributed systems https://zookeeper.apache.org/
  - Provides locking, leader election, monitoring
- Role of ZooKeeper in Kafka
  - Metadata management: list of brokers, configuration for topics and partitions
  - Leader election: to determine partition leader
  - Cluster health monitoring
- Cons
  - ✗ Operational complexity: separate Zookeeper cluster
  - ✗ Scalability: as Kafka cluster grows, Zookeeper becomes bottleneck
  - ✗ Single point of failure for coordination: if Zookeeper is unavailable, Kafka brokers may be unable to manage metadata, elect new leader, and know broker status



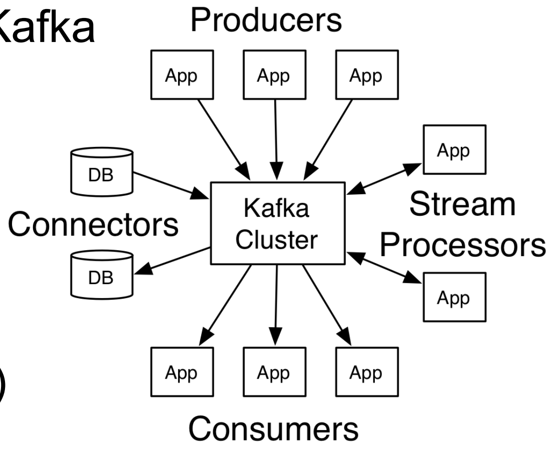
Kafka: from ZooKeeper to KRaft

- Apache Kafka Raft (KRaft)
  - Kafka-native consensus protocol that replaces Zookeeper for metadata management and leader election
  - Kafka itself stores metadata and partition leader information
  - Raft protocol coordinates metadata updates
  - KRaft is used for leader election
- Pros
  - ✓ Simplified architecture
  - ✓ Improved scalability
  - ✓ Better fault tolerance: metadata replicated to all brokers, making failover faster



Kafka: APIs

- A set of APIs for interacting with Kafka
- Producer API: allows clients to publish data to Kafka topics
- Consumer API: allows clients to consume data from Kafka topics
- Connect API: integrates external systems (databases, file systems) as data sources and sinks
  - Pre-built connectors: AWS S3, Lambda, MySQL, Postgres, …
- Admin API: manages clusters and resources (topics, partitions, brokers, etc.)
- Streams API: for real-time processing of streaming data



https://kafka.apache.org/documentation/#api

Kafka: client libraries

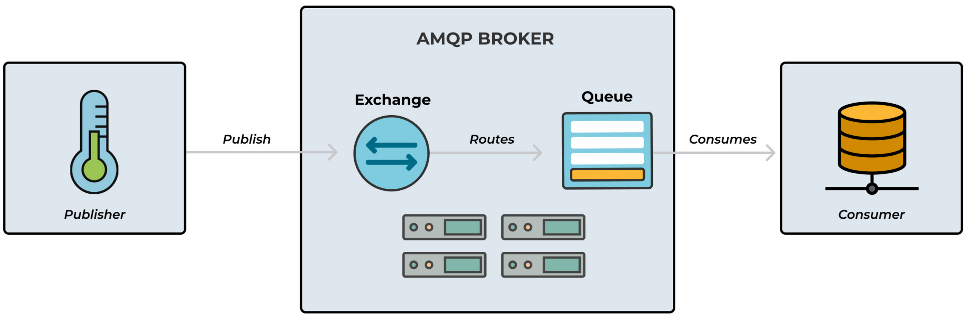
- Kafka officially provides an SDK for Java
  - Fully featured (supports all APIs except Admin), actively maintained
- Community-driven Kafka client libraries for other languages, including
  - Go
    - Confluent https://github.com/confluentinc/confluent-kafka-go
    - Kafka-go https://github.com/segmentio/kafka-go
  - Python
    - Confluent https://github.com/confluentinc/confluent-kafka-python

Interacting with MOM: messaging protocols

- Application-layer open standard protocols for MOM interaction
  - **AMQP** (Advanced Message Queueing Protocol)
    - Binary protocol
  - MQTT (Message Queue Telemetry Transport) https://mqtt.org
    - Binary, lightweight protocol, for low-bandwidth and low-power devices, commonly used in IoT applications
  - STOMP (Simple Text Oriented Messaging Protocol) https://stomp.github.io/
    - Simple, text-based protocol, commonly used in web applications and real-time communication
- Goals:
  - Platform- and vendor-agnostic
  - Interoperability: facilitates communication between different MOMs

Messaging protocols and IoT

- Widely used in Internet of Things (IoT)
  - Messaging protocols enable data exchange between sensors and services that process data



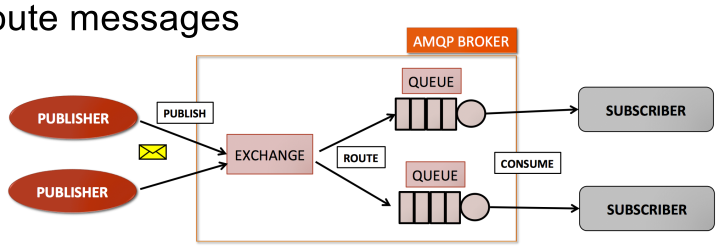
- Advantages on MOM in IoT
  - Decoupling: separate data producers from consumers
  - Resiliency: MOM provides temporary message storage
  - Traffic spikes handling: MOM can persist data and process it eventually

AMQP: features

- Open standard messaging protocol, supported by industry
  - Version 1.0, approved in 2014 by ISO and IEC
  https://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf
- Application-level, binary protocol
  - Based on TCP with additional reliability mechanisms (delivery semantics)
- Programmable protocol
  - Entities and routing schemes are primarily defined by apps
- Designed to provide reliable, scalable and secure messaging
- Supports queue-based and pub/sub patterns
- Implementations
  - Apache ActiveMQ, RabbitMQ, Azure Event Hubs, Pika (Python client library), …

AMQP: architecture

- 3 key actors: publishers, subscribers, and brokers
  - Brokers manage and route messages
- AMQP entities (within broker): queues, exchanges and bindings



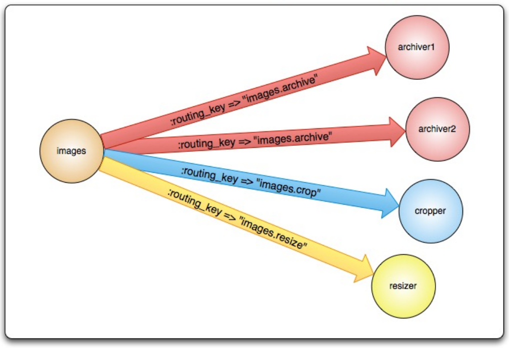
  - Queues: storage units that hold messages
  - Exchanges: routing entities that determined how messages are distributed to queues
    - Like post offices or mailboxes
  - Bindings: rules that define how messages are distributed to queues
  - Message delivery
    - Push delivery: brokers push messages to consumers subscribed to queues
    - Pull delivery: consumers pull messages from queues on demand

https://www.rabbitmq.com/tutorials/amqp-concepts

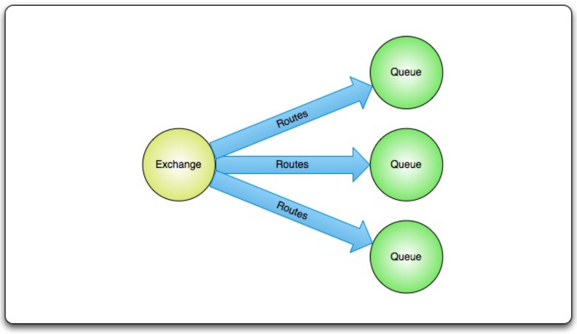
AMQP: routing

- Types of AMQP exchanges
  - Direct exchange: delivers messages to queues based on message routing key
    - Use case: routing to specific queues based on exact matching
  - Fanout exchange: delivers messages to all queues that are bound to the exchange
    - Use case: broadcasting to multiple queues

Direct exchange routing



Fanout exchange routing

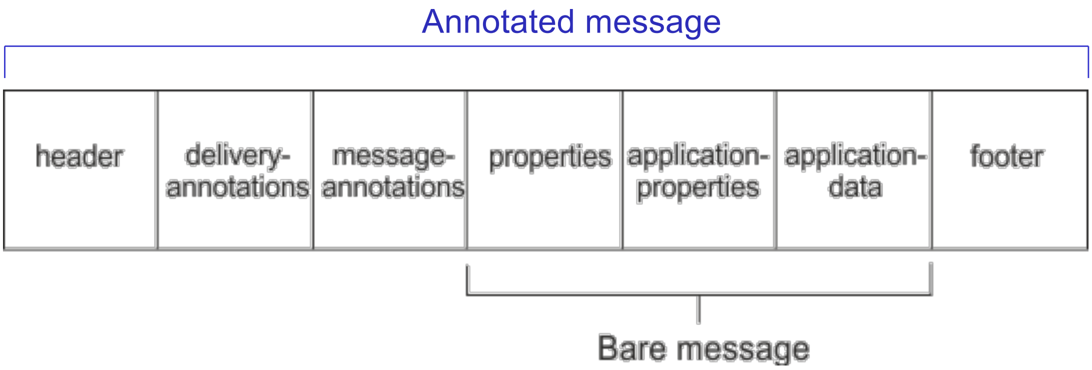


AMQP: routing

- Types of AMQP exchanges
  - Topic exchange: delivers messages to one or more queues based on topic matching
    - Routing key can include wildcards
    - Use case: commonly used to implement pub/sub patterns
    - Enables multicast routing, e.g., targeted notifications to users in different regions
    - Example: a message with routing key `"location.northAmerica.newYork"` will be delivered to queues bound with patterns like `"location.northAmerica.#"`, `"location.#"`, `"*.newYork"`
  - Headers exchange: delivers messages based on message headers
    - To route on multiple attributes that are expressed as message headers instead of routing key
    - Use case: complex multi-attribute matching

AMQP: messages

- AMQP defines two message types:
  - Bare messages: raw messages supplied by sender
  - Annotated messages: modified by intermediaries (e.g., brokers) during transit by adding headers and annotations
- Message header conveys delivery parameters
  - Including durability requirements, priority, time to live



RabbitMQ

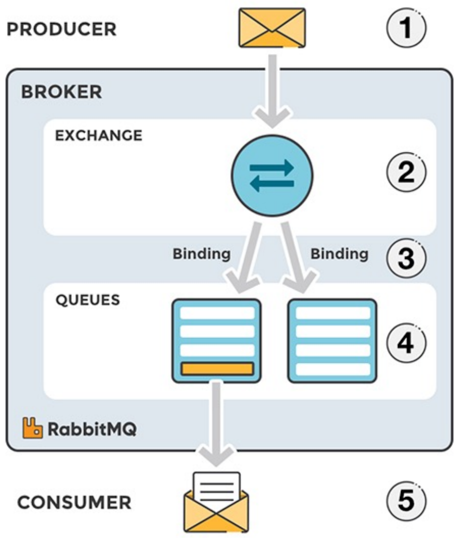
- Open-source message broker https://www.rabbitmq.com/

- Multiple messaging protocols supported
  - AMQP, MQTT, and STOMP
- Push delivery model (can also support pull)
- FIFO ordering guarantee at queue level
- Cross-platform
  - Can run on multiple operating systems and cloud environments
- Wide range of development tools (Java, Go, Python, …)

RabbitMQ: architecture

- Producers do not publish messages directly to queues
- Producer sends messages to an exchange, which routes messages to one or more queues with the help of bindings and routing keys
  - Binding: link between an exchange and a queue

Message flow in RabbitMQ



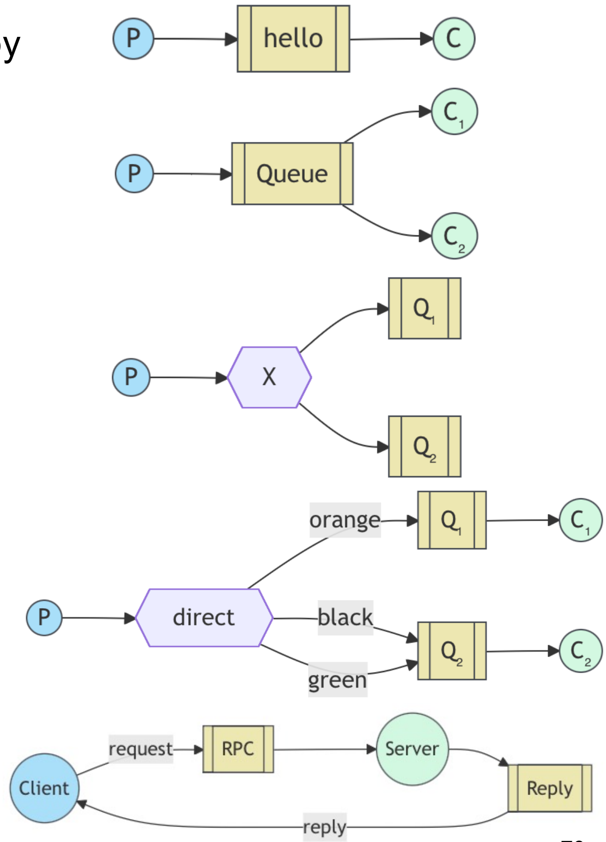
- RabbitMQ broker can be distributed, e.g., forming a cluster https://www.rabbitmq.com/distributed.html
  - Supports quorum queues: durable, replicated FIFO queues based on Raft consensus algorithm

RabbitMQ: delivery semantics

- At-most-once delivery by default
  - Consumer does not explicitly send acks (`autoAck = true` in the consumer settings); if the consumer fails before processing the message, the message will be lost
- At-least-once can be configured
  - Explicit message ack must be configured (`autoAck = false` in the consumer settings) and the consumer is required to send an explicit ack after successfully processing the message

RabbitMQ: use cases

1. Store and forward messages sent by a producer and received by a consumer (message queue)
2. Distribute tasks among multiple workers (work queue)
3. Deliver messages to many consumers (pub/sub) using a *message exchange*
4. Routing messages: producer sends messages to an *exchange,* that selects the queue based on binding rules and routing keys
5. Run a function on consumer and wait for result (RPC)



https://www.rabbitmq.com/tutorials

RabbitMQ and Go

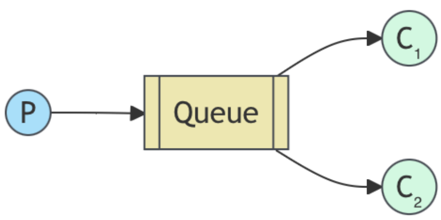
- Let's use RabbitMQ, Go and AMQP (messaging protocol) for:

Ex. 1: Message queue

https://www.rabbitmq.com/tutorials/tutorial-one-go

Ex. 2: Work queue

https://www.rabbitmq.com/tutorials/tutorial-two-go.html



Code on Teams/course website

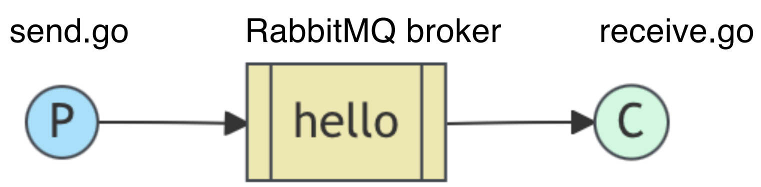
RabbitMQ and Go

- Preliminary steps:

1. Install RabbitMQ and start RabbitMQ server on localhost (port 5672) https://www.rabbitmq.com/download.html
   `$ rabbitmq-server`
   - RabbitMQ CLI tool: `rabbitmqctl`
     `$ rabbitmqctl stop`
     `$ rabbitmqctl status`
     Some useful commands for `rabbitmqctl`
     `list_channels`
     `list_consumers`
     `list_queues`
     `reset`
   - Also web UI for management and monitoring (port 15672)
2. Install Go AMQP client library
   `$ go get github.com/rabbitmq/amqp091-go`
   Details on `ampq`: https://pkg.go.dev/github.com/rabbitmq/amqp091-go

RabbitMQ and Go: example 1

1. Message queue pattern
   - Support single producer, single consumer or multiple producers, multiple consumers
   - Note that:
     - Message is delivered to only one consumer
     - Delivery is push-based



RabbitMQ and Go: example 2

2. Work queue pattern

– Version 1 (new_task_v1.go and worker_v1.go):
  - Multiple consumers: tasks are distributed among consumers in a round-robin fashion
  - Message loss scenario: if consumer crashes after the message is delivered by RabbitMQ but before completing the task, the message is lost

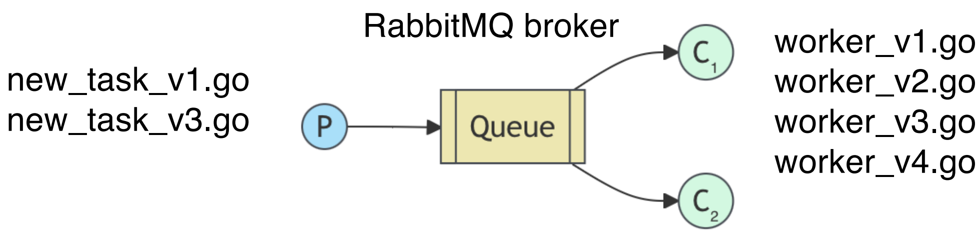
    `auto-ack=true`: message is considered to be successfully delivered as soon as it is sent to consumer ("fire-and-forget")

– Version 2 (new_task_v1.go and worker_v2.go):
  - Set `auto-ack=false` in `Consume` and add explicit ack in consumer to tell RabbitMQ that message has been processed and RabbitMQ can safely discard it
  - What happens when RabbitMQ is restarted while there are pending messages?
  - Which delivery semantics with explicit acks?

2. Work queue pattern

– Version 3 (new_task_v3.go and worker_v3.go):
  - Use a durable queue so messages are persisted to disk and survives RabbitMQ crash and restart
  - Queue declared with `durable=true` in `QueueDeclare`

– Version 4 (new_task_v3.go and worker_v4.go):
  - Improve task distribution by preventing a worker from receiving a new message while it still has unacknowledged messages
  - Achieved using channel prefetch (`Qos`) setting



Multicast communication

- **Multicast communication**: a group communication pattern where data is sent to *multiple* receivers (but not all) at once
  - Can be one-to-many or many-to-many
  - One-to-many multicast apps: video/audio resource distribution, file distribution
  - Many-to-many multicast apps: conferencing tools, multiplayer games, interactive distributed simulations
  - Broadcast: special case of multicast, where data is sent to *all* receivers
- Cannot be implemented via unicast replication (source sends indivudual copies to each receiver): lack of scalability
  - Solution: replicate data only when needed

Types of multicast

- How to realize multicast
  - **Network**-level multicast (IP-level)
    - Packet replication and routing managed by network routers
    - Uses IP Multicast
    - ✗ Limited in usage due to the need for specific router support and network infrastructure
  - **Application**-level multicast
    - Replication and routing managed by the hosts

Application-level multicast

- Basic idea:
  - Organize nodes into an overlay network
  - Use the overlay network to disseminate data
  - Can be structured or unstructured
- Structured application-level multicast
  - Explicit communication paths
  - How to build a structured overlay network?
    - Tree: one path between each pair of nodes (e.g., tree building based on Chord)
    - Mesh: multiple paths between each pair of nodes
- Unstructured application-level multicast
  - No predefined communication paths, typically relying on random connections between nodes

Unstructured application-level multicast

- How to realize unstructured application-level multicast
  - ✓ Flooding
    - Node *P* sends the multicast message *m* to all its neighbors
    - Each neighbor forwards *m* to all its neighbors (except to *P)* if it has not seen *m* before
  - ✓ Random walk
    - Node *P* sends multicast message *m* to a randomly chosen neighbor
    - The chosen neighbor forwards *m* to another randomly chosen neighbor
  - 👉 **Gossiping**

Gossip-based protocols

- Gossip-based protocols (or algorithms) are probabilistic (aka *epidemic* algorithms)
  - Gossiping effect: information spreads within a group just as it would be in real-life social interactions
  - Strongly related to epidemics, by which a disease is spread by infecting members of a group, which in turn can infect others
- Allow information dissemination in large-scale networks through random choice of successive receivers among those known to sender
  - Each node sends the message to a randomly chosen subset of nodes in the network
  - Each receiving node will forward the message to another randomly chosen subset, and so on

Origin of gossip-based protocols

- Proposed in 1987 by Demers et al. as a solution for data consistency in replicated databases with hundreds of servers
  - Assumption: no write conflicts (i.e., independent updates)
  - Updates are initially performed at one replica server
  - Each replica shares its updated state with only a few selected neighbors
  - Update propagation is *lazy*, i.e., not immediate
  - Eventually, each update should reach every replica

Demers et al., Epidemic Algorithms for Replicated Database Maintenance, PODC 1987 https://dl.acm.org/doi/pdf/10.1145/41840.41841

Why gossiping in large-scale DSs?

- Several attractive properties of gossip-based information dissemination for large-scale distributed systems
  - Simplicity of gossiping algorithms
  - No centralized control or management (and related bottleneck)
  - Scalability: each node sends only a limited number of messages, independently from system size
  - Reliability and robustness: thanks to message redundancy

Who uses gossiping? Examples

- AWS S3 uses a gossip protocol to rapidly disseminate information throughout its system. This allows S3 to quickly route around failed or unreachable servers, among other things
- Amazon's Dynamo uses gossiping for node failure detection
- BitTorrent uses a gossip-based information exchange
- Cassandra uses gossiping for group membership and node failure detection
- Gossip dissemination pattern
  https://martinfowler.com/articles/patterns-of-distributed-systems/gossip-dissemination.html

Strategies to spread updates

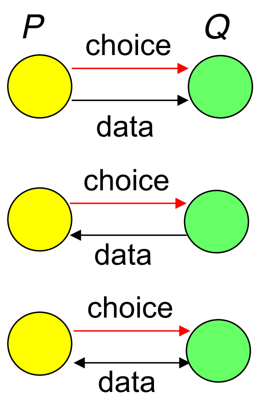
- Let's consider the two principal operations

1. Anti-entropy: each node periodically selects another node randomly and exchanges updates (i.e., state differences), with the goal of having identical states on both node afterwards

2. Rumor spreading: a node that has a new update (i.e., has been contaminated) periodically selects *F* (*F* >= 1) peers and sends them the update (contaminating them); a node that has received an update can stop further distributing it

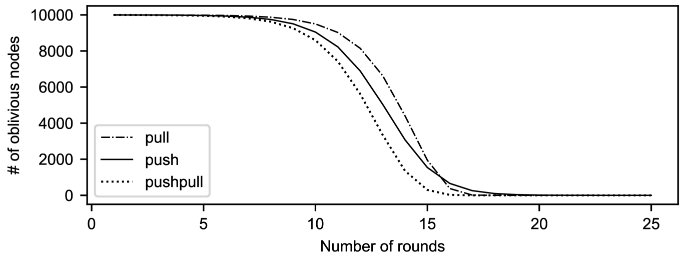
Anti-entropy

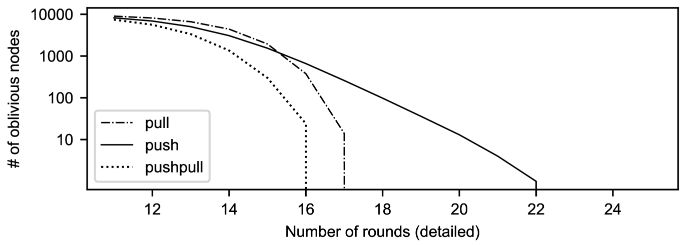
- Goal: increase node state similarity, thus decreasing "disorder" (the reason for the name!)
- Node *P* selects node *Q* randomly: how does P update *Q*?
- Different update strategies:
    1. Push: *P* only pushes its own updates to *Q*
    2. Pull: *P* only pulls new updates from *Q*
    3. Push-pull: *P* and *Q* send updates to each other, i.e., *P* and *Q* exchange updates



Anti-entropy: performance

- Push-pull
  - Fast and message-saving strategy: takes O(ln *N*) rounds to disseminate updates to *N* nodes, using O(*N* ln ln *N*) messages
  - *Round* (or *gossip cycle*): time interval in which every node initiates an exchange

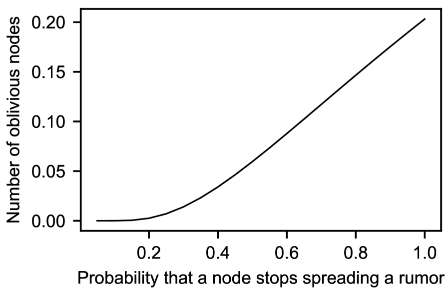




Rumor spreading

- Node *P*, having an update to report, contacts a randomly chosen node *Q* and forwards the update
- If *Q* has already been updated, *P* may lose interest in spreading the update further; with probability p_{stop} *P* stops contacting other nodes
- Fraction *s* of oblivious nodes (nodes that have not yet been updated) is equal to

$$s = e^{-(1/p_{stop}+1)(1-s)}$$



Consider 10,000 nodes		
$1/p_{stop}$	s	N_s
1	0.203188	2032
2	0.059520	595
3	0.019827	198
4	0.006977	70
5	0.002516	25
6	0.000918	9
7	0.000336	3

- To improve information dissemination (especially when p_{stop} is high), combine rumor spreading with anti-entropy

Framework for gossiping protocols

- Two nodes *P* and Q, where *P* selects Q to exchange information with
  - *P* runs at each round (Δ time units)

Active thread (node P):		Passive thread (node Q):
selectPeer(&Q)		
selectToSend(&buf)		
sendTo(Q, buf)	----->	receiveFromAny(&P, &req)
		selectToSend(&buf)
receiveFrom(Q, &resp)	<-----	sendTo(P, buf)
selectToKeep(view, resp)		selectToKeep(view, req)
processData(view)		processData(view)

selectPeer: randomly select a node to send the gossip message to
selectToSend: select some entries from node's local view to send
selectToKeep: select which received entries to store in node's local view; remove duplicate entries

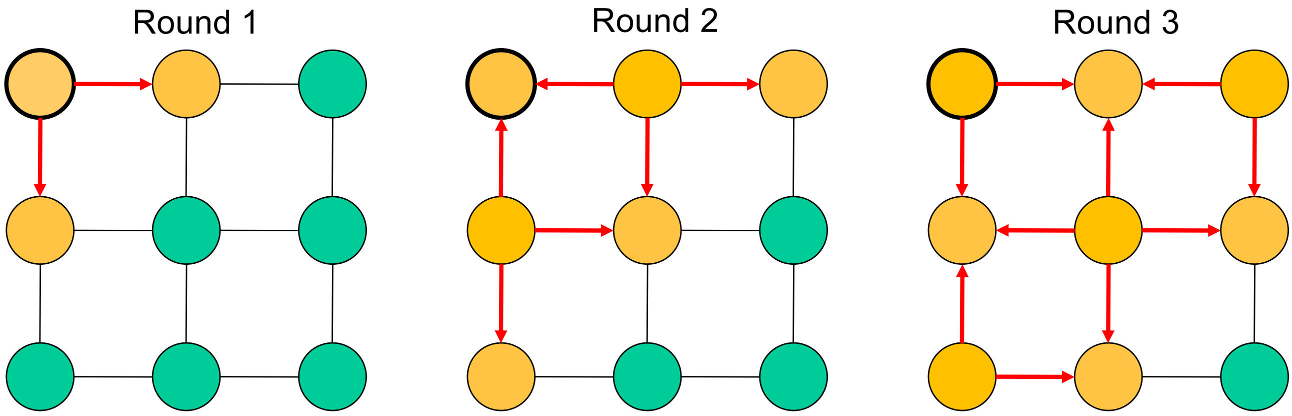
Kermarrec and van Steen, Gossiping in distributed systems, *SIGOPS Oper. Syst. Rev.*, 2007 https://www.distributed-systems.net/my-data/papers/2007.osr.pdf

Framework for gossiping protocols

- Simple? Not quite
- Several crucial aspects
  - Node selection to gossip: many alternatives including
    - Uniformly selecting from the set of currently available (alive) nodes
    - Selecting the peer that has been least contacted (e.g., used by CoachroachDB)
  - Data exchanged
    - What is exchanged is highly application-dependent
    - Choice of update strategy
  - Data processing
    - Again, highly application-dependent

Gossiping vs flooding: example

- Information dissemination is the classic and most popular application of gossiping protocols in DSs
  - Gossiping is generally more efficient than flooding
- Flooding-based information dissemination
  - Each node that receives a message forwards it to all its neighbors (including the sender)
  - Message is eventually discarded when TTL reaches 0

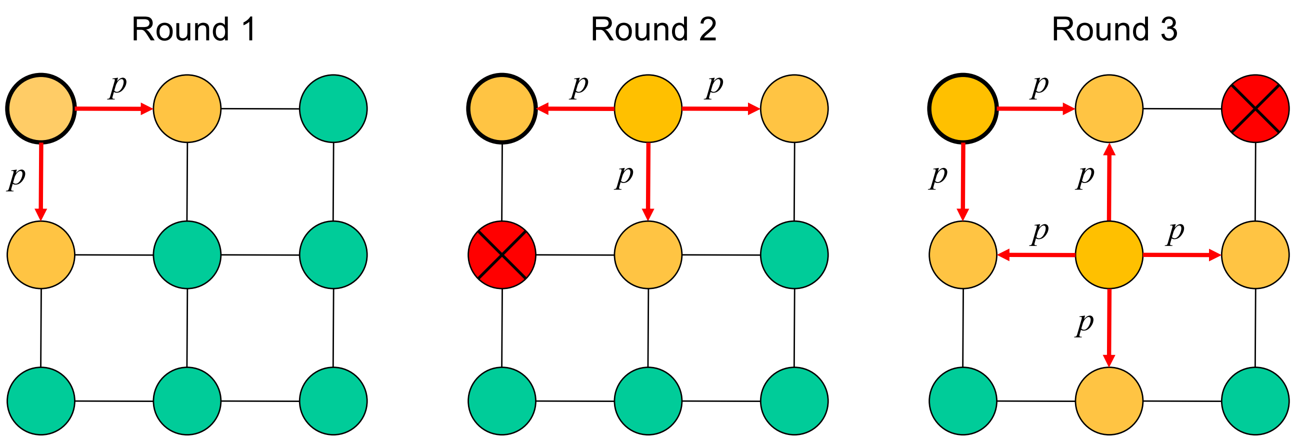


Sent messages: 18
Reached nodes: 8 out of 9

Gossiping vs flooding: example

- Use an example of rumor spreading algorithm
  - The node sends the message to each of its neighbors with probability *p*

```
for each msg m
    if random(0,1) < p then send m
```



Sent messages: 11
Reached nodes: 7 out 9

Gossiping vs flooding

- Gossiping key features
  - Probabilistic
  - Localized decision, yet leads to a global state
  - Lightweight
  - Fault-tolerant
- Flooding pros and cons
  - ✓ Universal coverage and minimal state information needed
  - ✗ Can flood the network with redundant messages
- Gossiping goals
  - Reduce redundant transmissions compared to flooding while attempting to retain its advantages
  - However, due to its probabilistic nature, it cannot guarantee that all the peers are reached and it generally takes longer to complete than flooding

Other application domains of gossiping

- Besides information dissemination…
- Group membership
  - To know the list of nodes in DS (who is part of the system)
- Peer sampling
  - To select nodes from a larger set of available nodes to interact with
- Resource management in large-scale DS
  - Including monitoring and failure detection
- Distributed computations for data aggregation in large-scale DS (e.g., sensor network)
  - Computation of aggregates, e.g., sum, average, maximum, minimum
  - Example: computing average
    - Let $v_{t,i}$ and $v_{t,j}$ be the values at time t stored at nodes i and j
    - During a gossip exchange, i and j exchange their current local value v_i and v_j and adjust it to

$$v_{t+1,i}, v_{t+1,j} \leftarrow (v_{t,i} + v_{t,j})/2$$

Gossiping case studies

1. Blind counter rumor mongering: example of gossip-based disseminatio protocol
2. Bimodal multicast: builds on gossiping to provide reliable multicast

Blind counter rumor mongering

- Why this name?
  - *Rumor mongering* (def.: “the act of spreading rumors”, also known as gossip): a node with a “hot rumor” periodically infects other nodes
  - *Blind*: the node loses interest regardless of who the recipient is (*why*)
  - *Counter*: the node loses interest after a fixed number of contacts (*when*)
- Two parameters to control gossiping
  - *B*: max number of neighbors a message is forwarded to
  - *F*: number of times a node forwards the same message to its neighbors

Portman and Seneviratne, The cost of application-level broadcast in a fully decentralized peer-to-peer network, ISCC 2002

Blind counter rumor mongering

- Gossiping protocol

```
A node n initiates a broadcast by sending message m to B
   of its neighbors, chosen at random
When node p receives a message m from node q
  If p has received m no more than F times
    p sends m to B neighbors, chosen uniformly at random,
    among those that p believes have not yet seen m
```

  - Note that *p* knows whether its neighbor *r* has already seen *m* only if *p* has previously sent *m* to *r*, or if *p* has received *m* from *r*

- Performance (*B*=*F*=2) compared to flooding
  - Lower number of messages: ~50%
  - Incomplete coverage: ~90%
  - Slower dissemination: ~2x

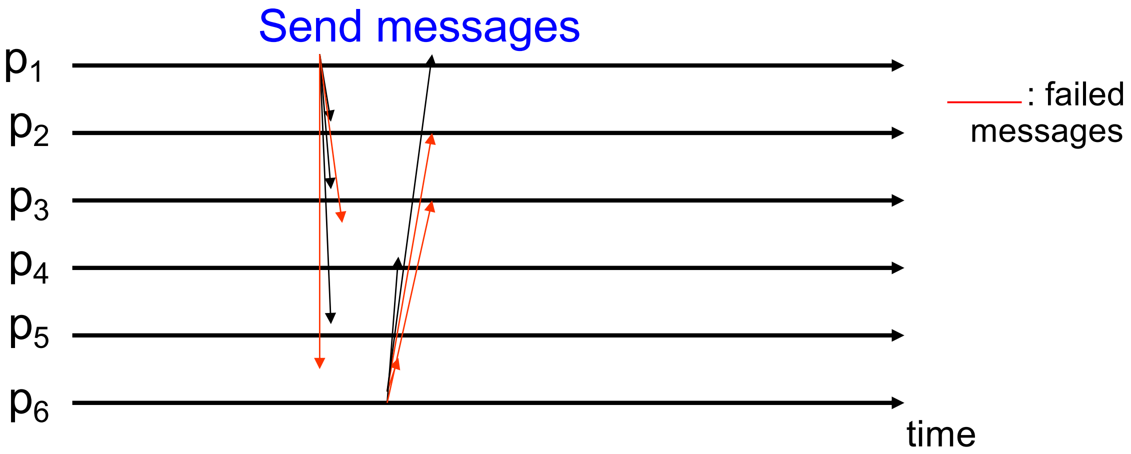
Bimodal multicast

- Aka pbcast (probabilistic broadcast)
- 2-phase protocol:
  1. **Message distribution**: a process sends a multicast message with no particular reliability guarantees
  2. **Gossip repair**: after receiving a message, a process begins to gossip about it to a set of peers
     - Gossip occurs at regular intervals, giving processes a chance to compare their states and fill gaps in their message sequence

- Used by Fastly CDN for cache invalidation
  https://www.fastly.com/blog/building-fast-and-reliable-purging-system

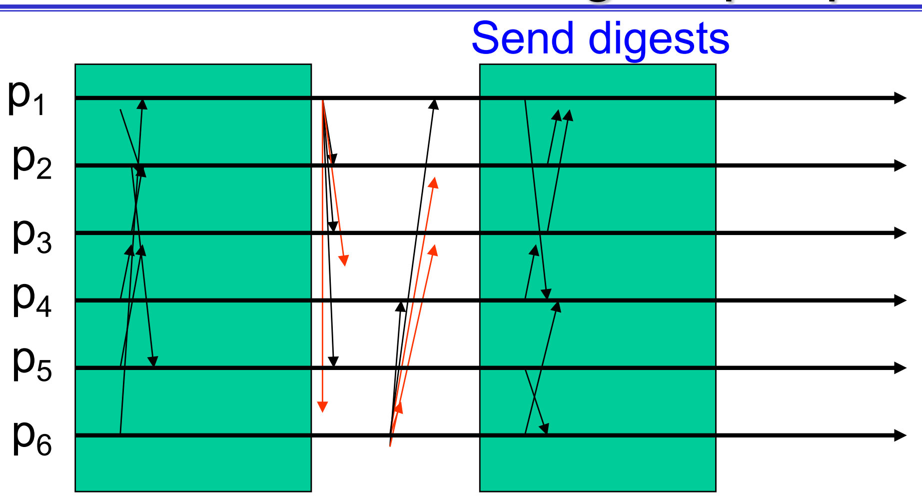
Birman et al., Bimodal multicast, *ACM Trans. Comput. Syst.,* 1999

Bimodal multicast: message distribution



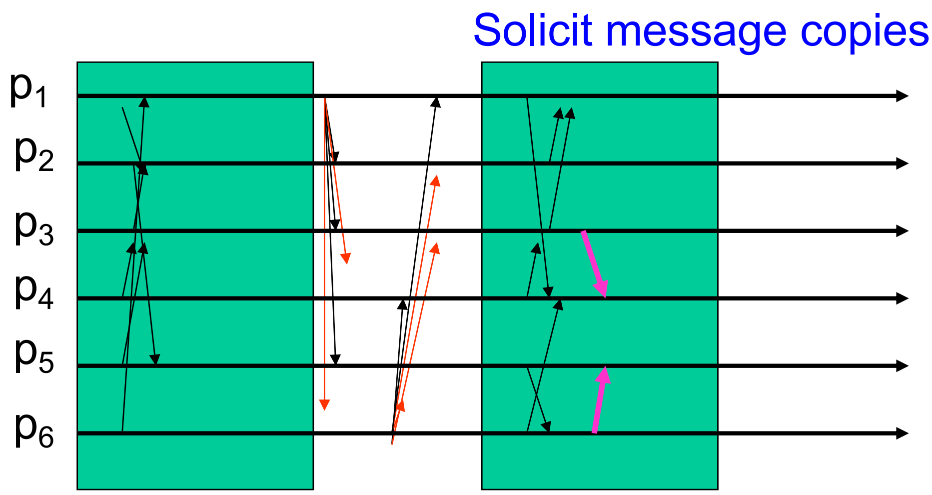
- Start with *unreliable* multicast to rapidly distribute messages
- Partial distribution may occur:
  - Some message may not reach all nodes
  - Some process may be faulty

Bimodal multicast: gossip repair



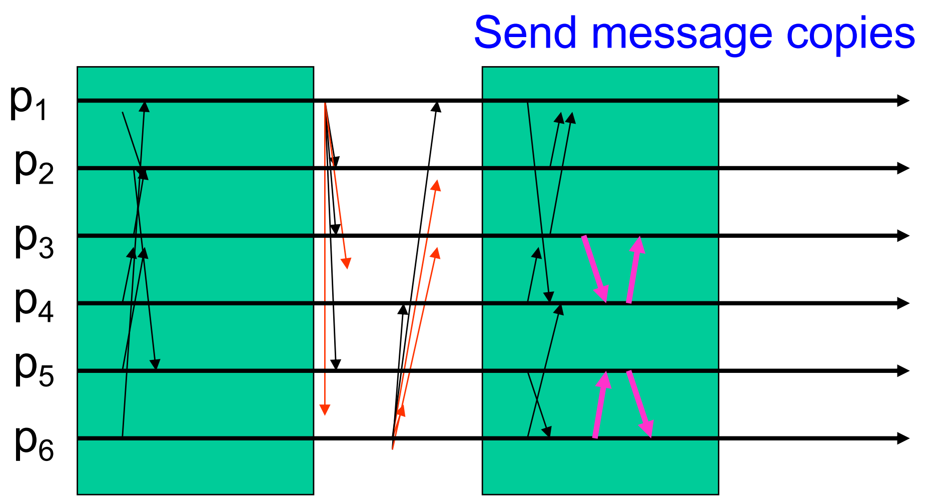
- Periodically (e.g., every 100 ms), each process sends a *digest* describing its state to a *randomly* selected process
- The digest only identifies messages, without including them

Bimodal multicast: gossip repair



- The recipient checks the gossip digest against its own message history and *requests* copies of any missing messages from the process that sent the gossip

Bimodal multicast: gossip repair

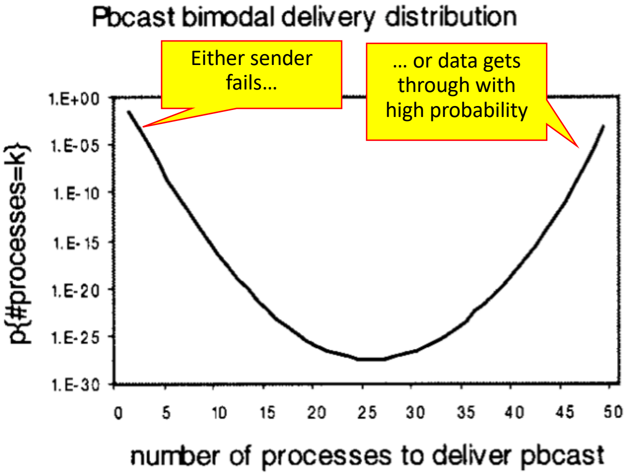


- Processes reply to solicitations received during a gossip round by retransmitting the requested messages
- Some optimizations exist (not examined)

Bimodal multicast: why "bimodal"?

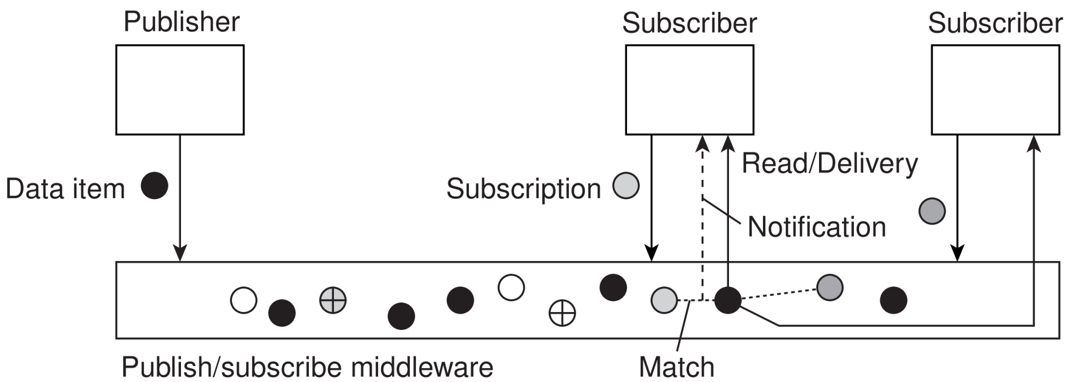
- Are there two phases?
- No; dual "modes" of result
  1. Bimodal delivery pattern: pbcast is almost always delivered to most or to few processes, and almost never to some processes

     Atomicity = almost all or almost none



  2. Bimodal delivery latencies: one distribution of very low latencies (messages arriving without loss in the initial phase) and a second distribution with higher latencies (messages that had to be repaired afterward)

Pub-sub event matching



- Subscriber specifies which events it is interested in (subscription *S*)
- Publisher publishes event *N*: does *N* match *S*?
- Challenge: efficiently implement event matching

Event matching: centralized architecture

- Naive solution: centralized architecture
  - Single server handles all subscriptions and notifications
- Server responsibilities:
  - Handles subscriptions from subscribers
  - Receives events from publishers
  - Checks events against subscriptions
  - Notifies matching subscribers

✓ Simple to realize, feasible for small-scale deployments

✗ Scalability

✗ SPOF

Event matching: distributed architecture

- Achieve matching scalability
- Simple solution: partition subscriptions

1. Hierarchical architecture: master distributes matching across multiple workers
   - Each worker stores and handles a subset of subscriptions
   - Master receives events and distribute them to workers for matching
   - Partitioning strategy
     - Topic-based pub/sub: hash topic names to map subscriptions and events to workers

   ✗ Single master
2. Flat architecture: no single master, matching is spread across distributed servers
   - Partitioning strategy
     - Topic-based pub/sub: hash topic names to select server

Event matching: distributed architecture

- Other solutions: decentralized servers (overlay network)
- Challenge: how to route notifications to subscribers?

1. Unstructured overlay: use flooding or gossiping to disseminate event notifications
   - Store a subscription at only one server, but disseminate notifications to all servers: matching is distributed across servers
   - Selective routing: install filters to ignore paths toward nodes that are not interested, reducing unnecessary messages
2. Structured overlay: use a DHT to disseminate event notifications

References

- Chapter 4 and Section 5.6 of van Steen & Tanenbaum book

- Kafka doc https://kafka.apache.org/documentation
- Conductor's Kafkademy https://docs.conduktor.io/learn
- Kafka: A Distributed Messaging System for Log Processing https://pages.cs.wisc.edu/~akella/CS744/F17/838-CloudPapers/Kafka.pdf
- Sax, Apache Kafka, *Encyclopedia of Big Data Technologies*, 2018 https://link.springer.com/rwe/10.1007/978-3-319-77525-8_196

- RabbitMQ https://www.rabbitmq.com

- Montresor, Gossip and epidemic protocols, *Wiley Encyclopedia of Electrical and Electronics Engineering*, 2017 http://disi.unitn.it/~montreso/ds/papers/montresor17.pdf
- The cost of application-level broadcast in a fully decentralized peer-to-peer network https://ieeexplore.ieee.org/document/1021785
- Bimodal multicast https://dl.acm.org/doi/pdf/10.1145/312203.312207