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. 2026/27

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 and callee must be up and running at the same time.
  - Spatial: caller must know callee address
  - Functionality and communication are tightly coupled
- How to improve decoupling and flexibility?
- Persistent message-oriented communication
  - **Message Oriented Middleware (MOM)**

No AI/ML Processing

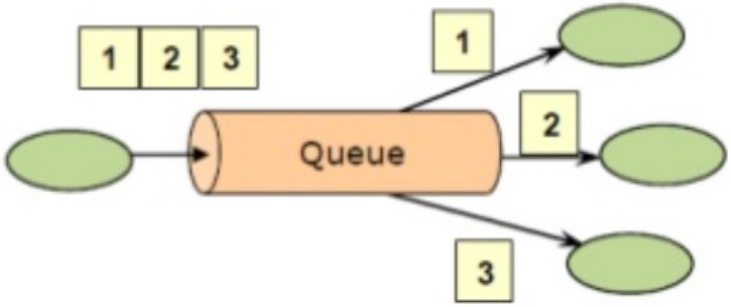
Message-oriented middleware

- 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**

No AI/ML Processing

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

No AI/ML Processing

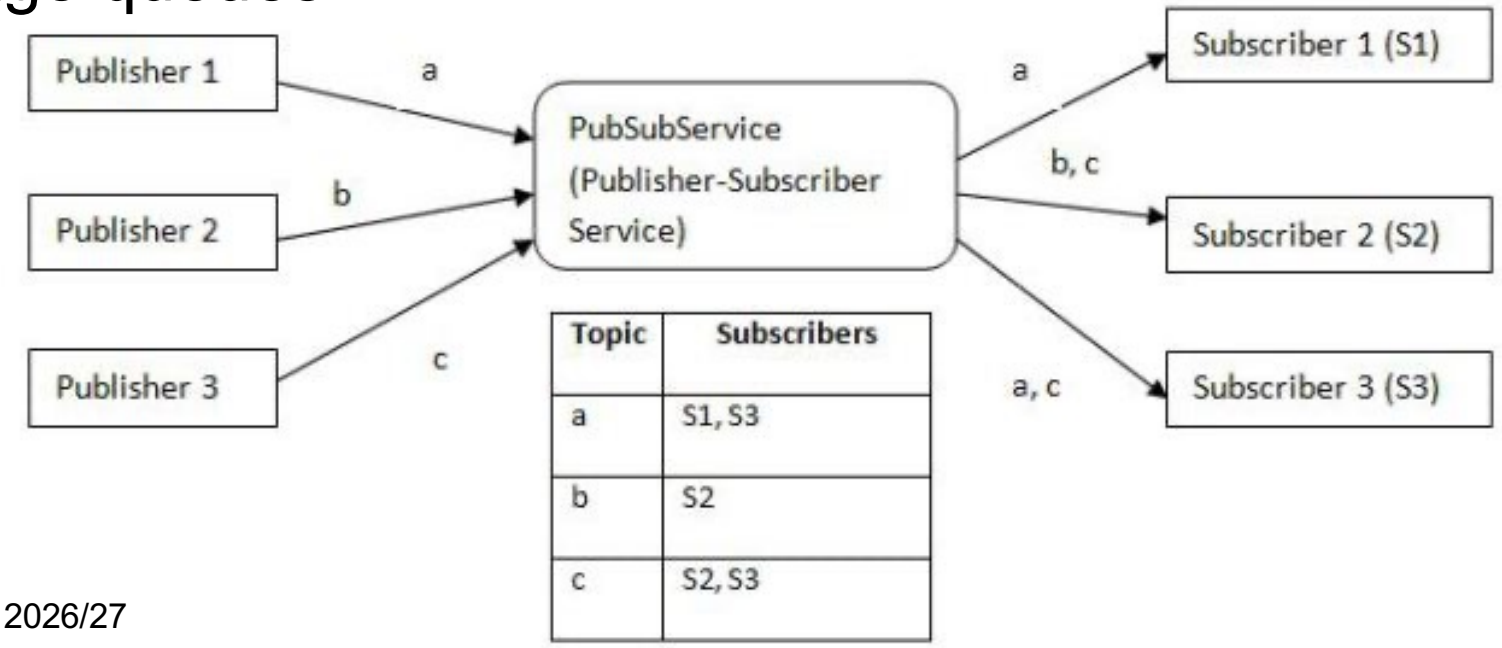
Message queue API

- 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 retrieves 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

No AI/ML Processing

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
- Pub/sub system collects subscriptions and routes messages to all matching subscribers
  - Distributed for scalability
- High degree of decoupling among components
  - In dynamic environments components can be added or removed
- Use cases:
  - Event notification, multicasting

No AI/ML Processing

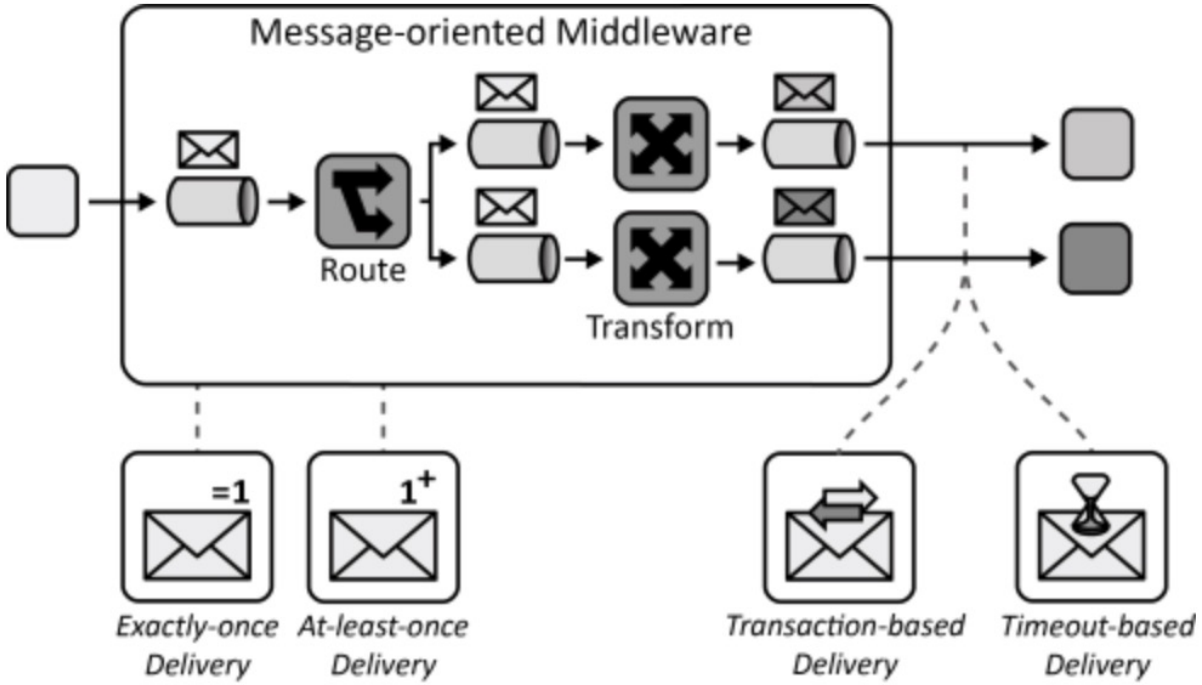
Publish/subscribe API

- Typical calls in pub/sub system:
  - `publish(message)`: called by a publisher to publish a message
    - Any data type, may include metadata
  - `subscribe(filter_expr, notify_cb, expiry) → sub_handle`: called by a subscriber to subscribe to messages
    - Inputs: filter expression, reference to a notification callback, and subscription expiry time
    - Returns a subscription handle
  - `notify_cb(sub_handle, message)`: called by the pub/sub system to deliver a matching message to subscribers
  - `unsubscribe(sub_handle)`: called by a subscriber to remove its subscription

No AI/ML Processing

MOM: functionalities

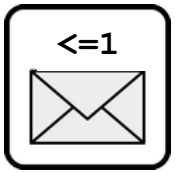
- MOM simplifies distributed communication by handling:
  - Availability of components: delivery semantics and models
  - Addressing and routing: locating the recipient(s) for messages and delivering messages along appropriate paths: message routing
  - Message format transformations: message broker



https://www.cloudcomputingpatterns.org/message_oriented_middleware

MOM: delivery semantics

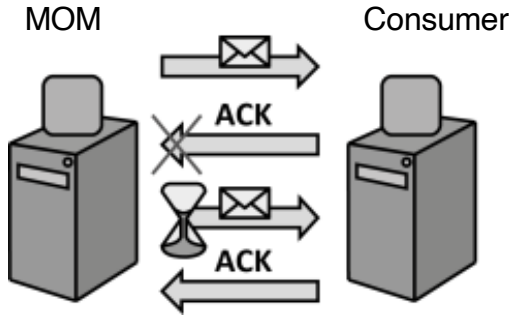
At-most-once delivery



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

At-least-once delivery





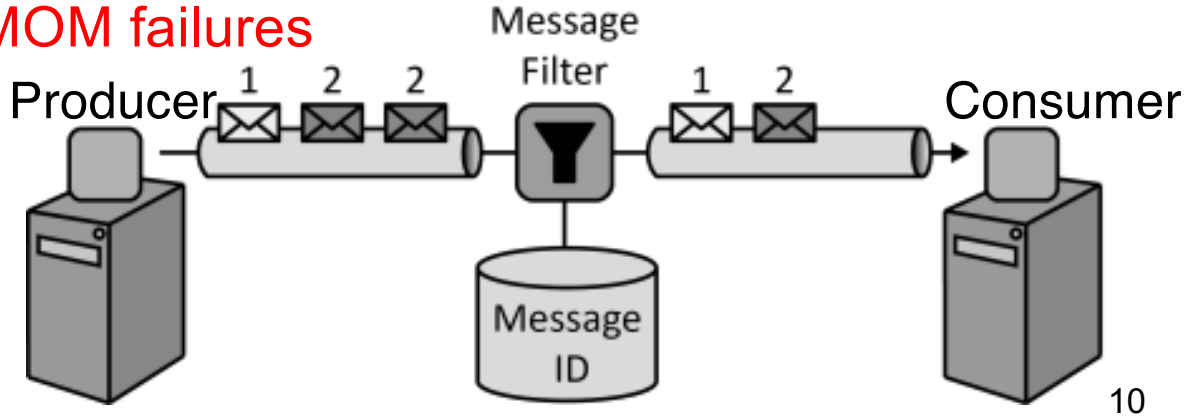
- Each message is delivered at least once
- Mechanism:
  - Consumer sends ack for each message
  - MOM resends message if ack is missing
- Messages are never lost, but they may be delivered multiple times
  - Why? Message received, but ack lost or consumer crashes after processing and before acking
  - Consumer must be able to handle duplicate processing (or be sure that re-processing the same message does not cause errors)

MOM: delivery semantics

Exactly-once delivery



- In theory: each message is delivered exactly once to consumer
  - But a lost ack is indistinguishable from a lost message
  - In practice: systems provide exactly-once processing
- Mechanisms: those for at-least-once plus
  - Deduplication (unique message ID): MOM filters message duplicates
  - Idempotent consumers: can safely process the same message multiple times
  - Messages survive MOM failures

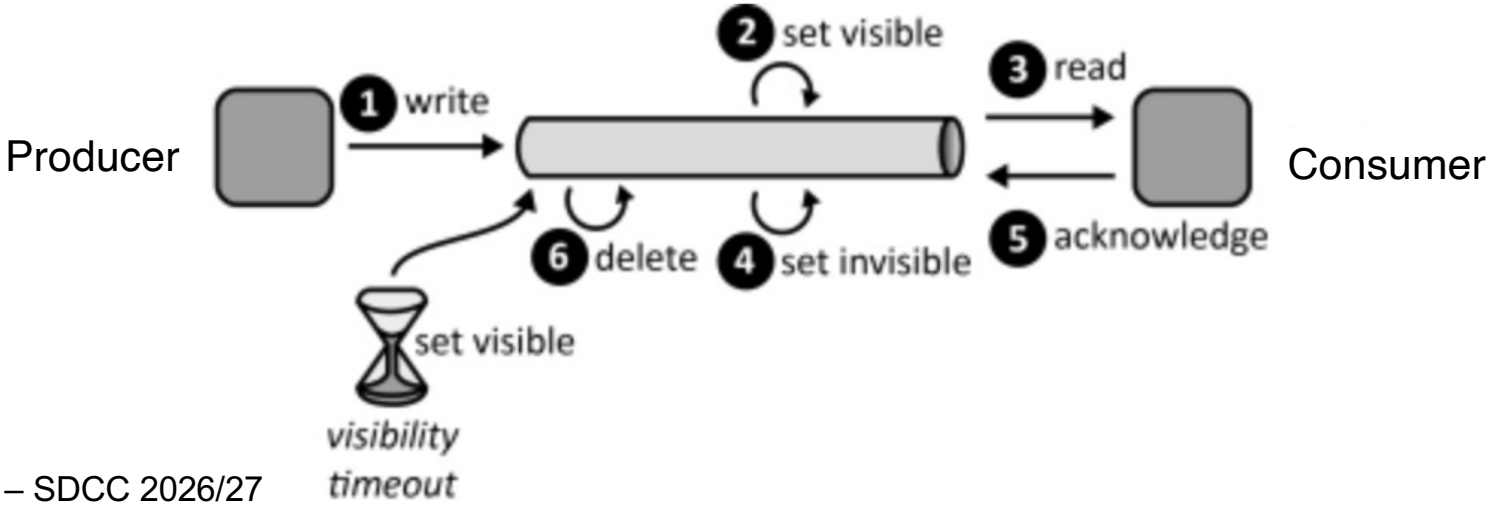


MOM: delivery semantics

Timeout-based delivery



- Messages are delivered at least once while reducing duplicate consumption
- Mechanism
  - Message stays in the queue and is marked invisible until *visibility timeout* expires
  - Invisible message cannot be read by other consumers
  - After consumer's ack, message is deleted from queue

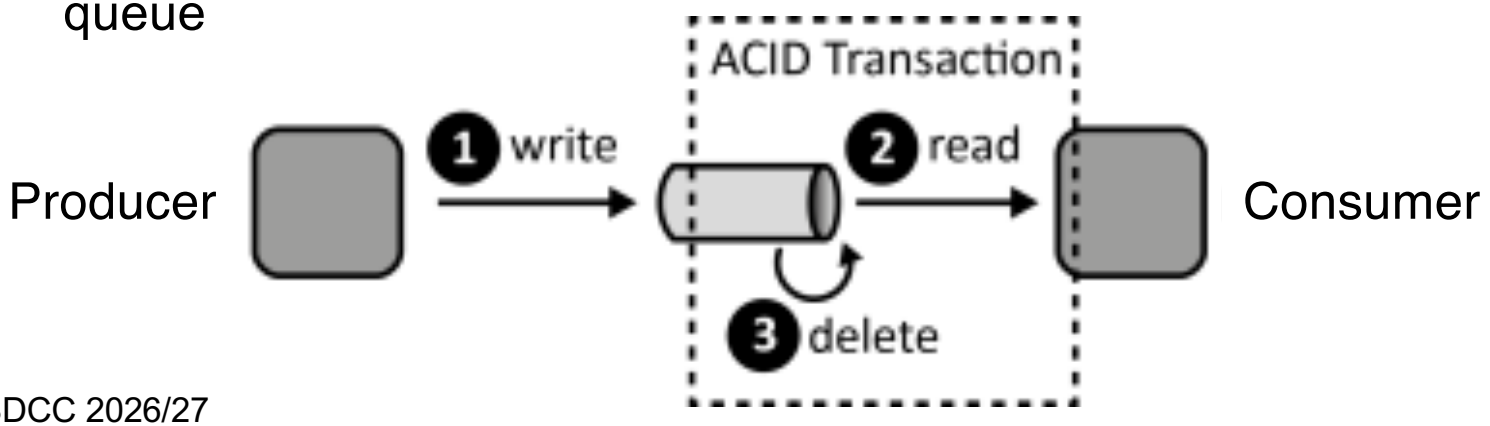


MOM: delivery semantics

Transaction-based delivery

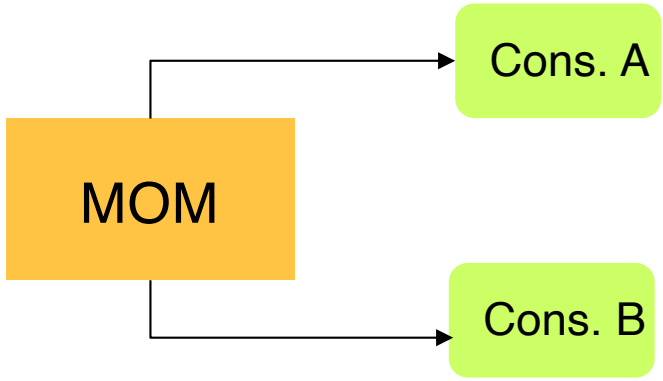


- Messages are deleted from the queue only after they have been successfully received and processed by the consumer
- Mechanism
  - MOM and consumer participate in a transaction: read and delete operations are part of the transaction, ensuring ACID properties
    - Commit: if successful, the message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

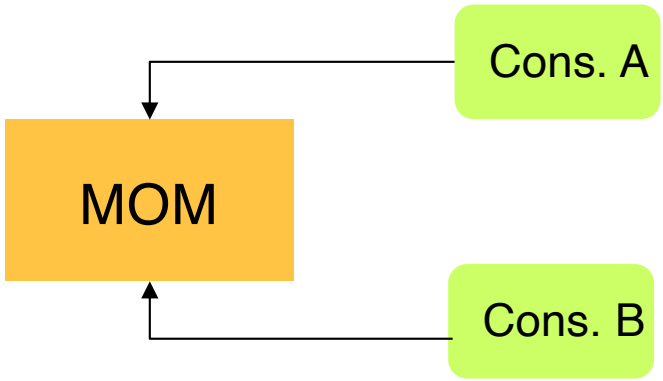


MOM: delivery models

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



Push delivery

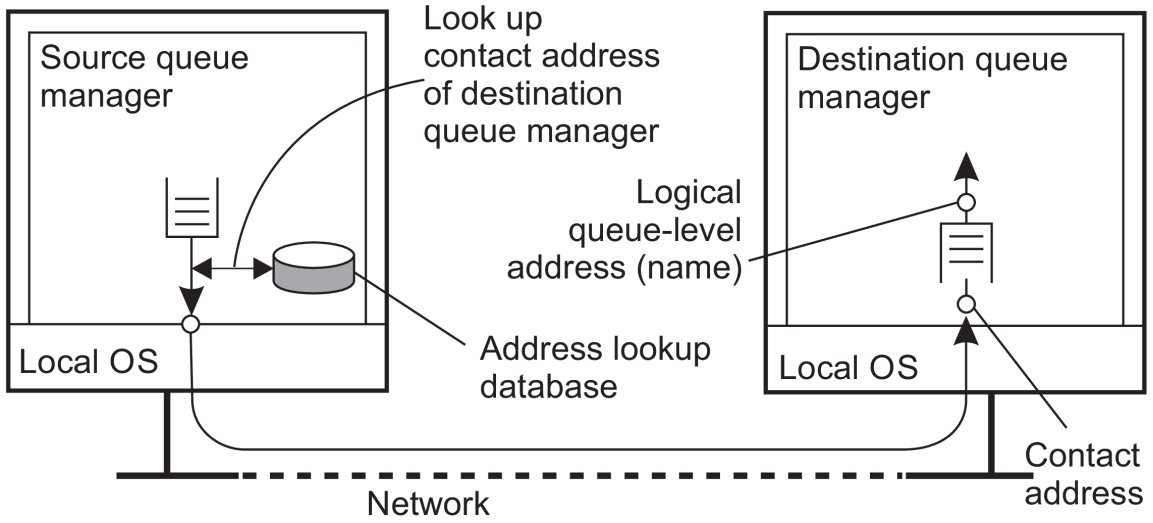


Pull delivery

No AI/ML Processing

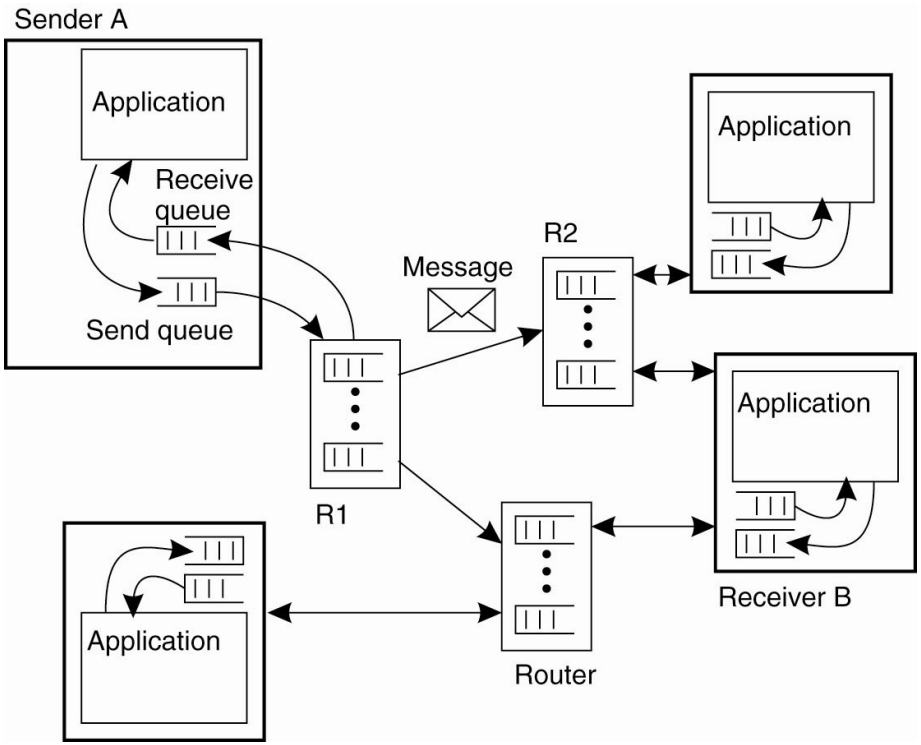
MOM: Message routing

- 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



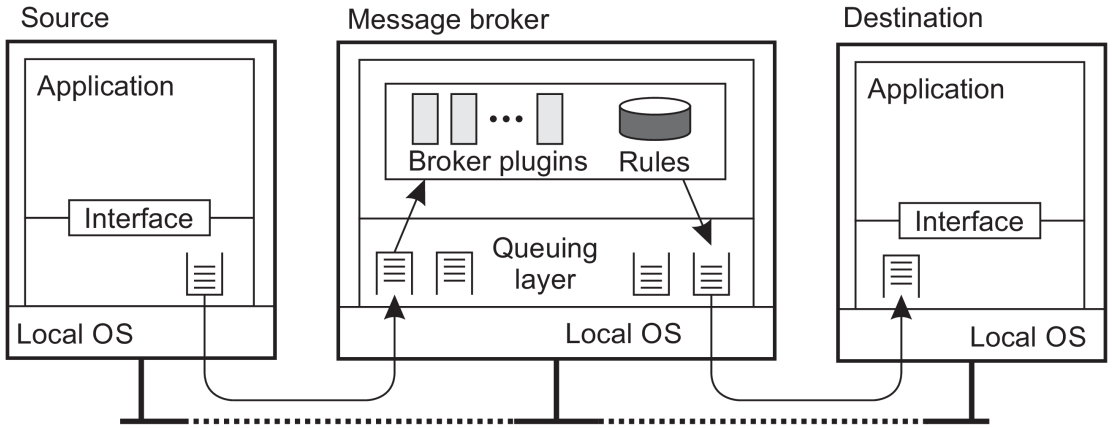
Message routing: overlay network

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



MOM: message broker

- Problem: applications rarely share a common data format, how to integrate them?
- Solution: message broker between producer and consumer to handle application heterogeneity
- Responsibilities
  - Message transformation: converts message types using a repository of conversion rules
  - Content-based routing: routes messages by content or type
- Usually distributed, for scalability and reliability



MOM frameworks

MOM systems and libraries	Cloud-based MOM services
Apache ActiveMQ https://activemq.apache.org	**Amazon Simple Queue Service**
Apache Kafka	Amazon Simple Notification Service
Apache Pulsar https://pulsar.apache.org	CloudAMQP: RabbitMQ as a Service
IBM MQ https://www.ibm.com/products/mq	Google Cloud Pub/Sub
NATS https://nats.io	Microsoft Azure Service Bus
RabbitMQ	
Redis https://redis.io/	
ZeroMQ https://zeromq.org	

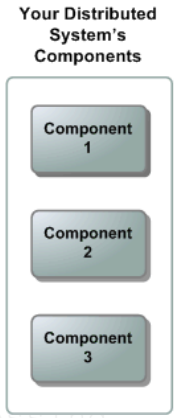
- Distinction between queue-based and pub/sub patterns is blurred
  - Many support both queue and pub/sub patterns
- Cloud MOMs abstract infrastructure management, allowing developers to focus on messaging patterns

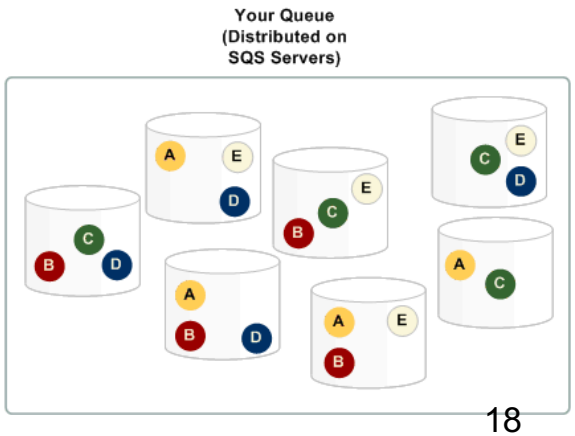
No AI/ML Processing

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 an AWS region: messages stored on multiple servers for high availability
  - Pull delivery model: consumers poll the queue to retrieve messages

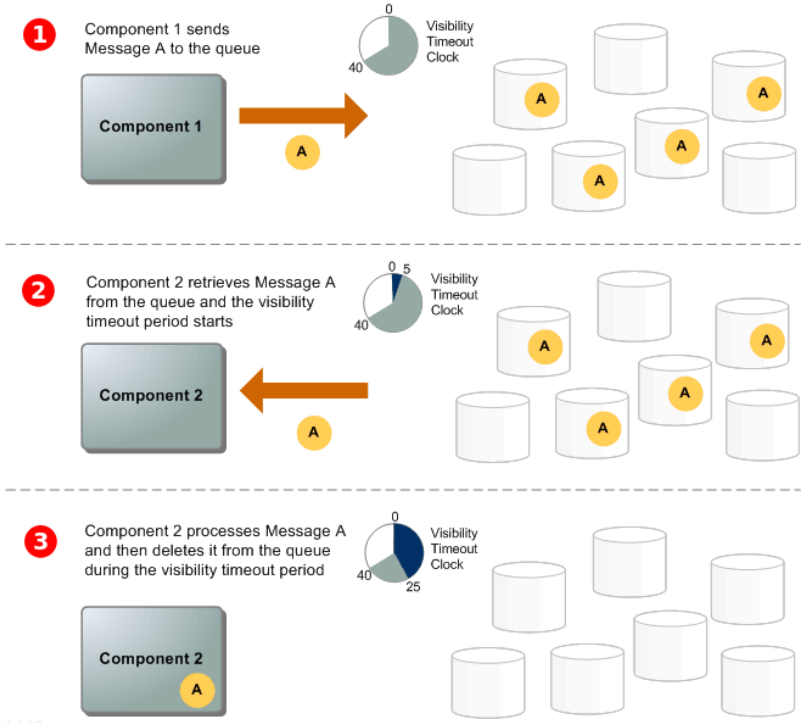




No AI/ML Processing

Amazon SQS: Message handling

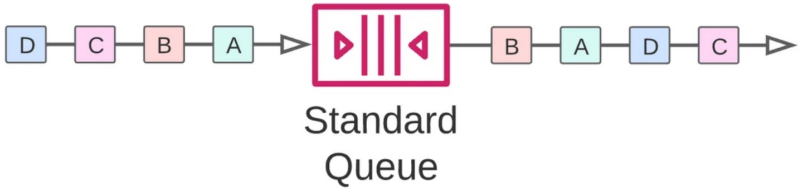
- Consumer must delete message after processing
  - Queue = temporary holding location, not permanent storage
  - Message retention period: configurable (max 14 days)
- Provides timeout-based delivery
  - When a consumer retrieves a message, it remains in the queue but becomes invisible to others (visibility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



No AI/ML Processing

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





No AI/ML Processing

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 1 MB)
  - For larger payloads: store on S3 and send an S3 reference instead
- `ReceiveMessage`
  - Retrieve one or more messages
    - Up to 10 messages: can specify the maximum number of messages to fetch (from 1 to 10)
  - No filtering before delivery, only 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

No AI/ML Processing

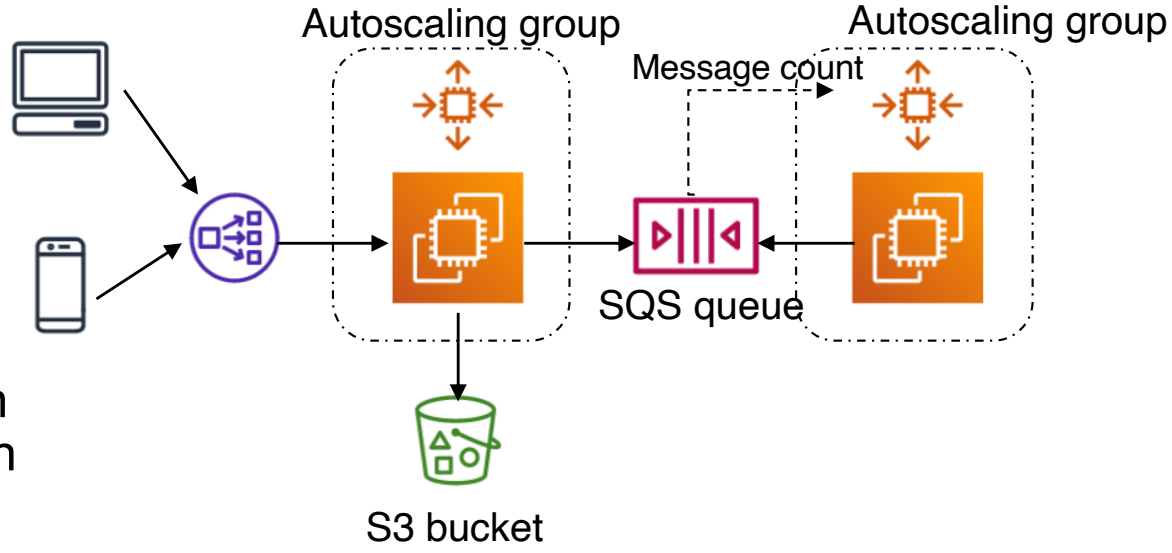
Amazon SQS: API

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

No AI/ML Processing

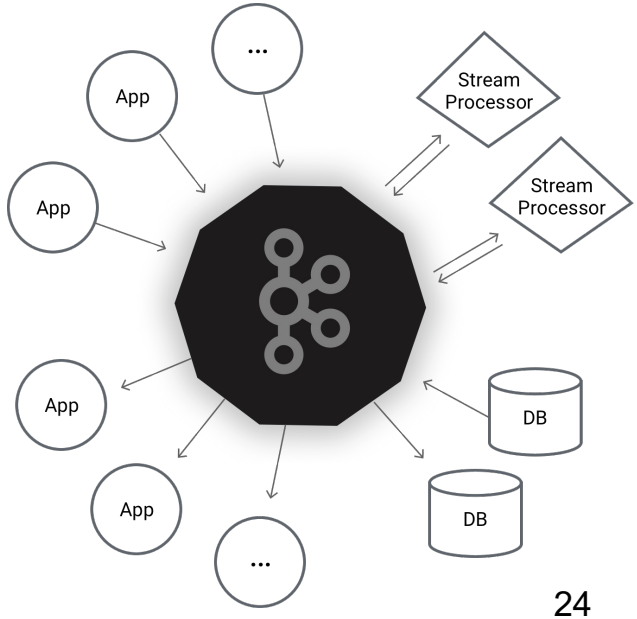
Amazon SQS: example

- Photo processing service using SQS
  - Goal: use SQS to decouple 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 instance gets a photo, processes it (e.g., resize) and deletes 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 out based on number of queued messages



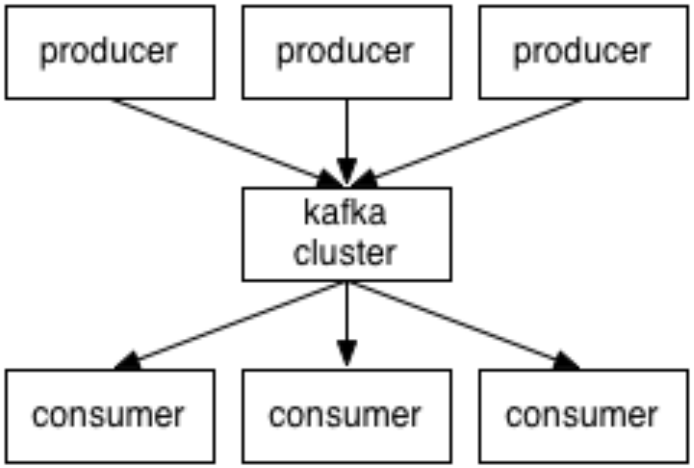
Apache Kafka

- 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



No AI/ML Processing

Kafka at a glance

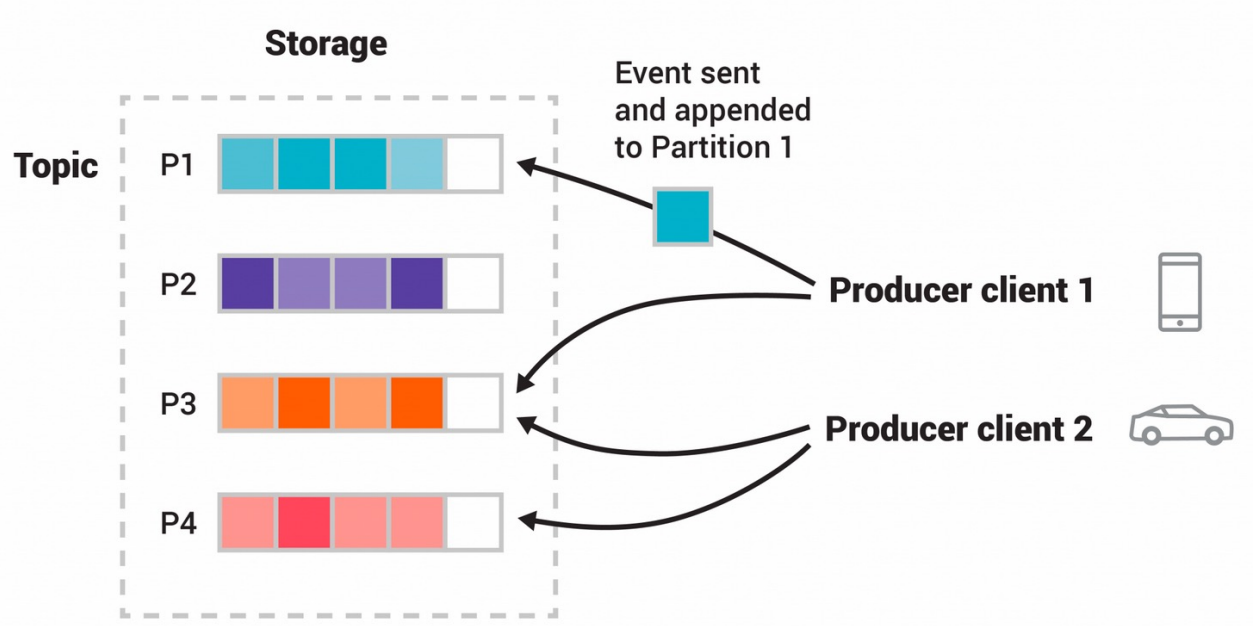


- **Kafka topics**: Kafka stores streams of events (aka *messages* or *records*) in categories known as topics
- **Kafka cluster**: composed of servers called brokers, which can span multiple data centers or cloud regions
  - Brokers receive and store events
- **Producers**: publish (write) events to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

No AI/ML Processing

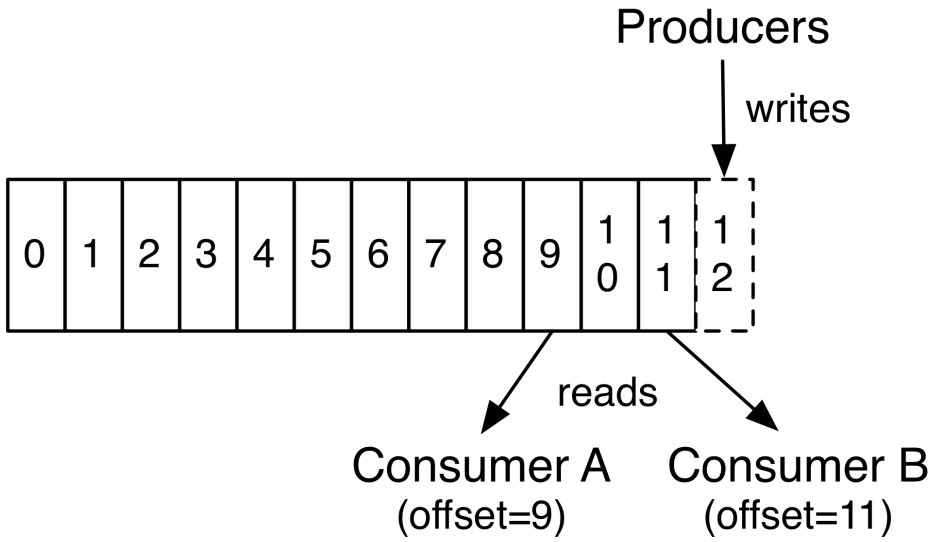
Kafka: topics and partitions

- 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: unit of parallelism for topic



Kafka: partitions and parallelism

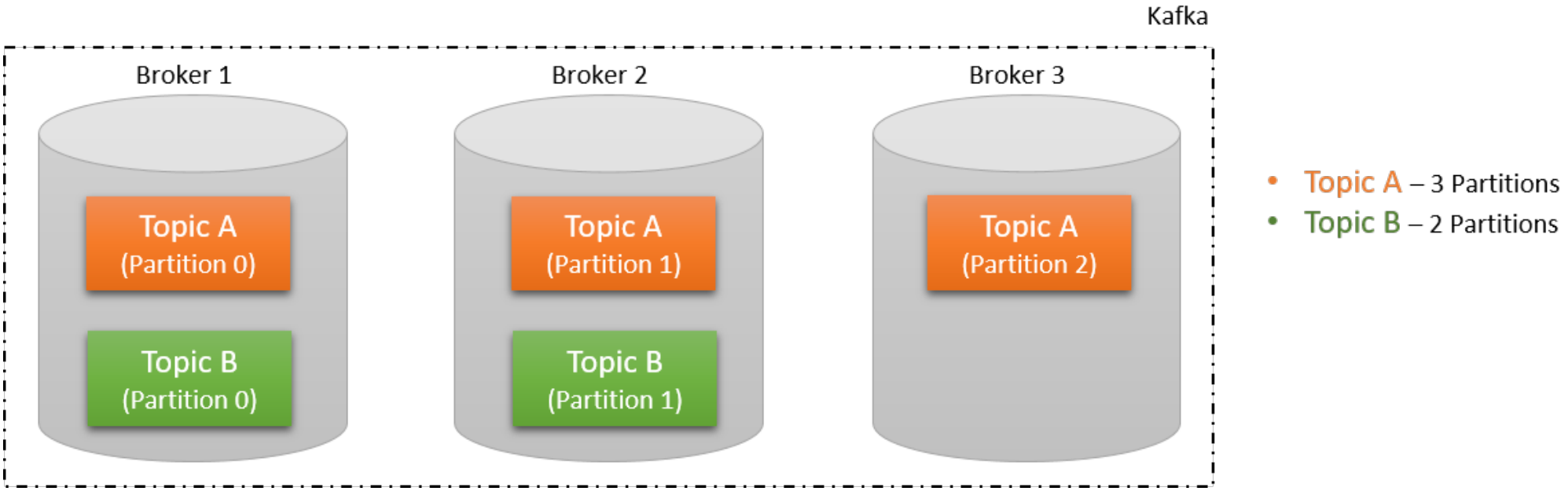
- Parallelism and scalability come from having multiple partitions per topic
- Producers publish events to topic partitions
- Consumers read events from the topic
- Each partition is:
  - A numbered, ordered, immutable sequence of events
  - Appended to, never modified *(*immutable events)
  - Similar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



No AI/ML Processing

Kafka: partitions and design choices

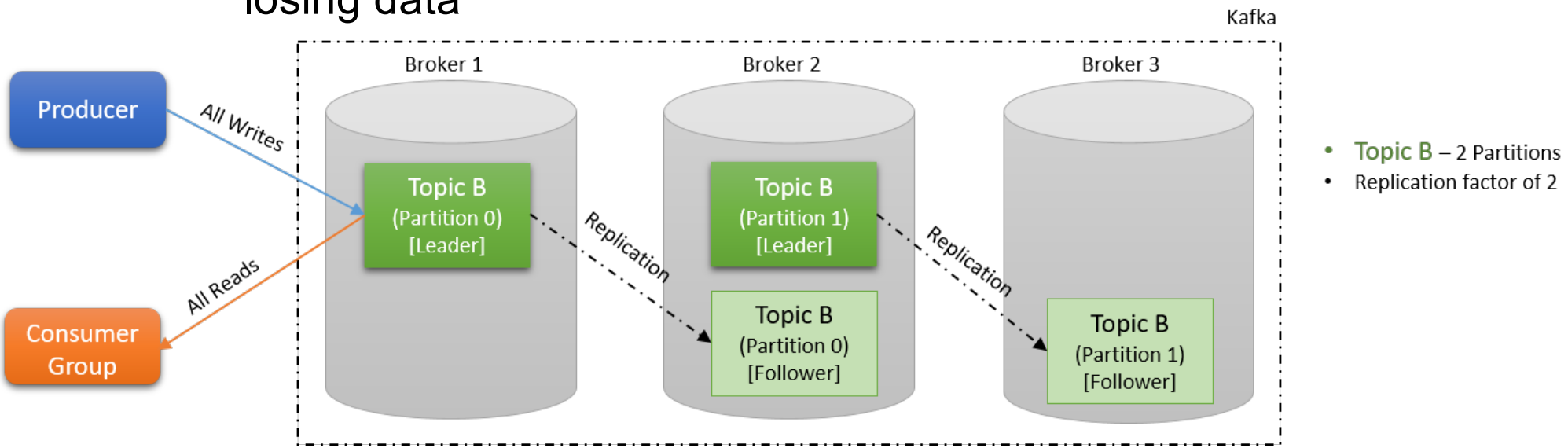
- Improve scalability: topic partitions are distributed across multiple brokers
  - ✓ I/O throughput increases through parallel reads and writes
    - Multiple producers can write concurrently to different partitions
    - Multiple consumers can read concurrently from different partitions



No AI/ML Processing

Kafka: partitions and design choices

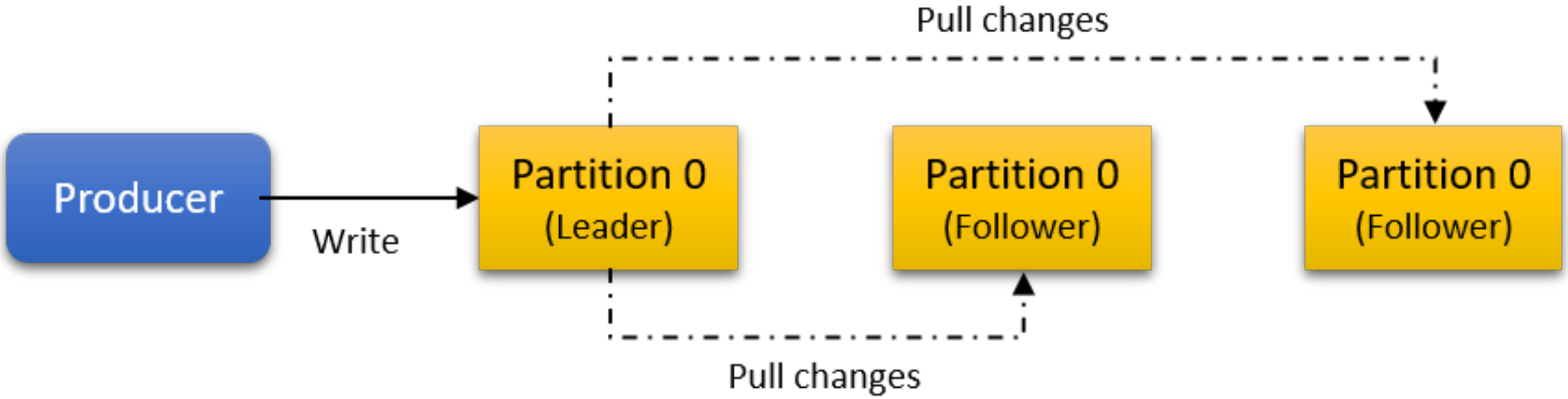
- Improve fault tolerance: partitions can be *replicated* across brokers
  - Each partition has:
    - 1 leader: handles all writes and reads
    - 0 or more followers: replicate the leader data
- Replication factor = total number of replicas (including leader)
  - Replication factor = *N* → up to *N*-1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



No AI/ML Processing

Kafka: partitions and design choices

- Simplify data consistency management: leader-based replication
  - Each partition has one leader, which orders all writes
    - Producers write to leader
    - By default, consumers read from leader
  - Followers replicate leader and serve as backups
  - Followers are:
    - in-sync: updated with leader
    - out-of-sync: lagging behind



No AI/ML Processing

Kafka: partitions and design choices

- Share responsibility and balance load: each broker acts as leader for some partitions and follower for others

- Coordinate brokers as they need a consistent view of cluster: live brokers, topics and partitions, partition leaders, configuration
  - Done by a controller, backed by a consensus algorithm (Raft)

No AI/ML Processing

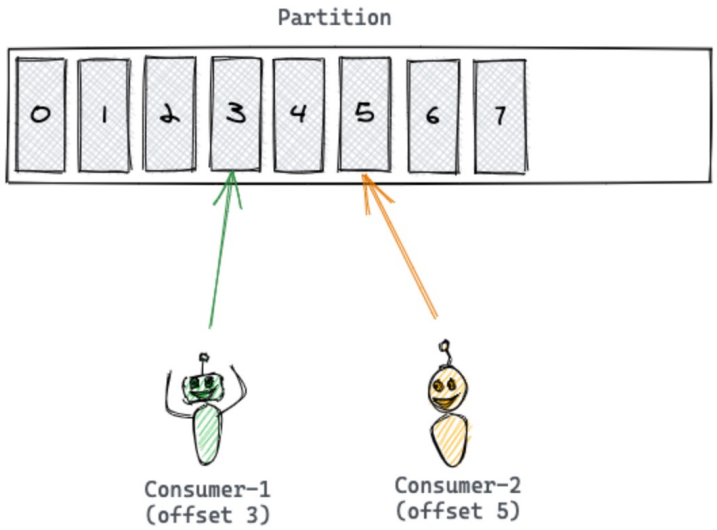
Kafka: producers

- Producers = data sources
- Publish events to topics of their choice
  - Send events only to the broker leader for the target partition (no routing tier)
- Responsible for assigning events to partitions:
  - Events with a key: key-based partitioning
    - Event contains a partition key (e.g., 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
  - Events without a key: sticky partitioning
    - Producer sends all messages to the same partition until the batch is sent
  - Custom partitioning
- Multiple producers can write concurrently to the same partition

No AI/ML Processing

Kafka: consumers

- Kafka uses a pull-based delivery model for consumers
  https://kafka.apache.org/44/design/design/#push-vs-pull
- A consumer tracks its progress using offset
  - Offset identifies which events have been consumed
- Multiple consumers can read concurrently the same partition, each reading at different offsets



No AI/ML Processing

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

No AI/ML Processing

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

No AI/ML Processing

Kafka: message retention and storage

- Kafka brokers store messages reliably on disk
- Messages are stored in log segments (files on disk)
- On each broker, messages are flushed from broker's in-memory buffer to disk
  - A log flush policy controls when messages are written to disk (time interval and number of messages)
- Unlike other MOM systems, Kafka does not delete messages after delivery, but retains messages
- Issue: how to free up disk space?
  - Topics are configured with retention time (how long messages should be stored)
    - Upon expiry, messages are marked for deletion
  - Alternatively, retention can be specified in size
- To save space, also log compaction
  - Only latest value for each message key in a topic is retained

No AI/ML Processing

Hands-on Kafka

- Setup
  - Download and install Kafka
    https://kafka.apache.org/downloads
    - Configure Kafka properties in `server.properties`, e.g.,
      - `listeners`, which are endpoints the broker listen on: `listeners=PLAINTEXT://localhost:9092`
      - log flush policy
  - Start Kafka environment
    - Generate cluster UUID
    - Format log directories
    - Start Kafka broker (default port: 9092)

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

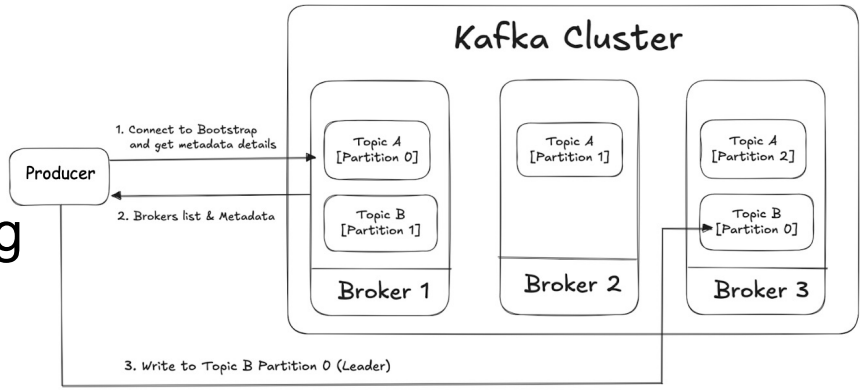

No AI/ML Processing

Hands-on Kafka

- Goal: use Kafka CLI to create a topic, publish and consume events to/from it, and delete it
- Create topic test with 3 partitions and no replication (`replication-factor=1`)

```
$ kafka-topics --create --bootstrap-server localhost:9092
   --replication-factor 1 --partitions 3 --topic test
```

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



No AI/ML Processing

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
>univ:Tor Vergata
```


No AI/ML Processing

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
```

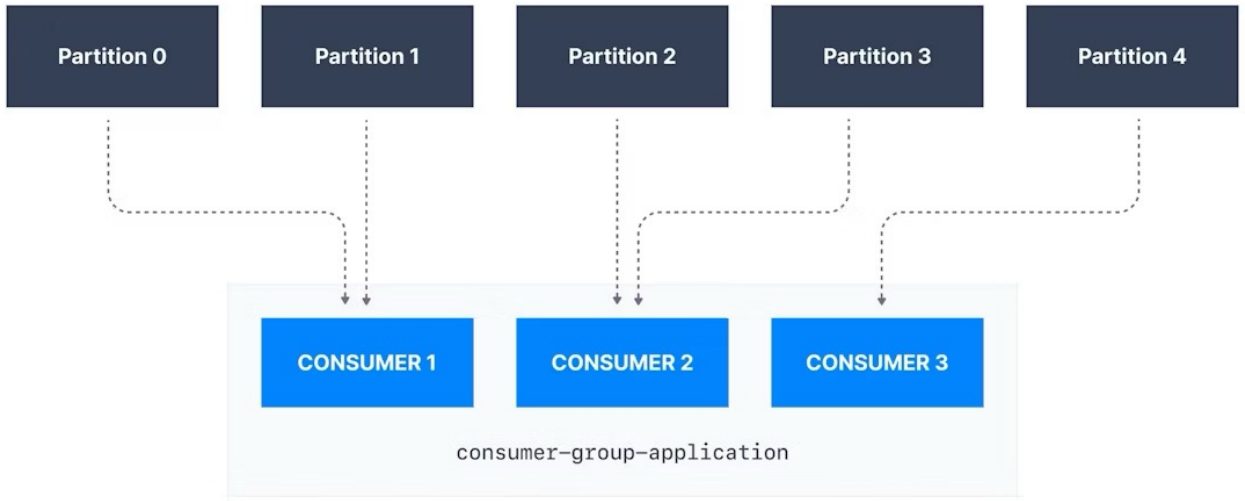
- Stop Kafka

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


No AI/ML Processing

Kafka: consumer group

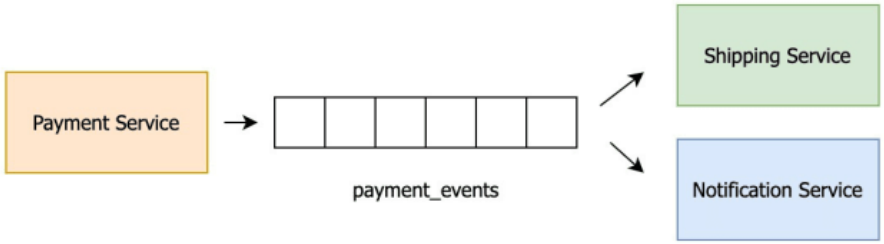
- Consumer Group: set of consumers that cooperate to consume data from a topic and share a group ID
  - Represents a *logical* subscriber
  - Topic partitions are divided among consumers in the group
    - Reassigned when consumers join/leave the group
  - Each event is delivered to *only one* consumer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



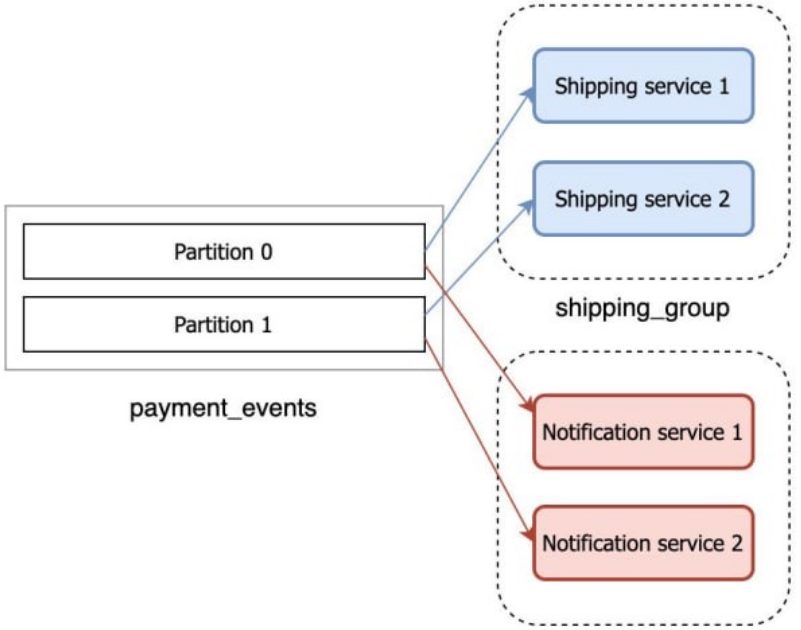
No AI/ML Processing

Kafka: consumer group

- Example: microservices communicating through Kafka



  - Each microservice has its own consumer group
    - Different microservices: each receives *all* events (pub/sub)
  - When we scale out a microservice, its replicas join the same group
    - Partitions are split among replicas: each event is processed by one replica
    - Max parallelism = number of partitions



Hands-on Kafka: consumer group

- Launch 3 consumers in the same consumer group named `my-group`
  - Each consumer is assigned to a different topic partition

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

- Produce messages in the topic and see how they are distributed among the consumers
  - Each consumer will read only the messages from its partition

No AI/ML Processing

Kafka: event ordering guarantees

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

  message: m1 m2 m3 m4 m5
  offset: 10 11 12 13 14

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

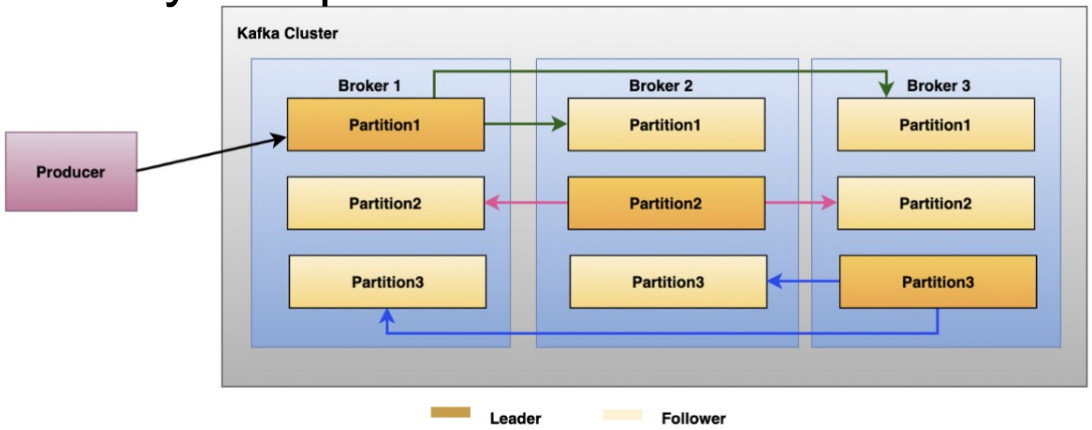
  m1 → m2 → m3 → m4 → m5

- Kafka provides ordering guarantees, but *only within a partition*
  - Total order of events within each partition (*per-partition ordering*)
  - No ordering across partitions of the same topic
- Per-partition ordering + key-based partitioning: sufficient for most applications

No AI/ML Processing

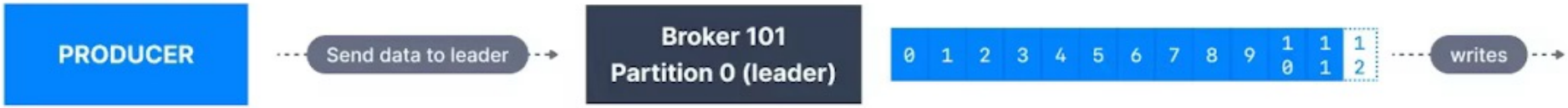
Kafka: replication

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

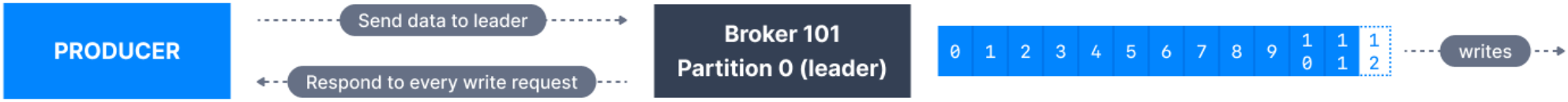


Kafka: write guarantees

- Controlled by `acks` and `min.insync.replicas`
- `acks` (producer): how many replicas in the ISR must acknowledge a write before the producer considers it successful
- `acks=0`: no ack (fire and forget), no retry



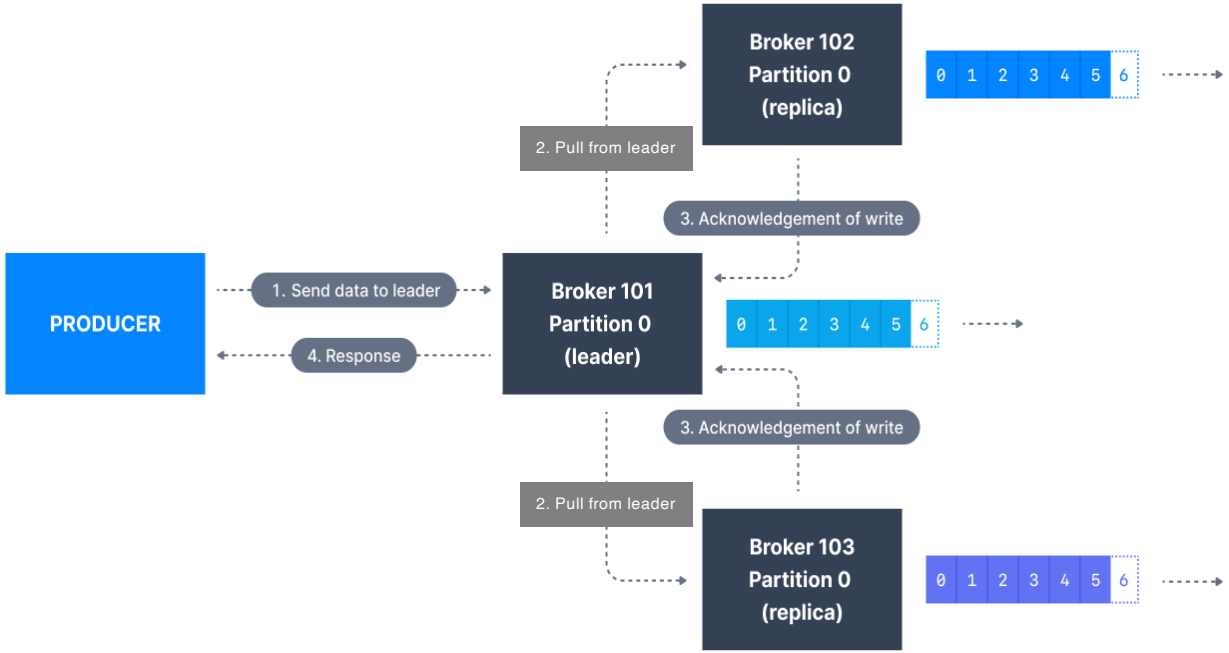
- `acks=1`: ack from partition leader only
  - If no ack at timeout, producer retries



No AI/ML Processing

Kafka: write guarantees

- `acks=all`: leader acks once written to all ISR members



- ISR changes *dynamically*: a replica is removed from ISR when it fails or lags, and rejoins when it catches up
- `min.insync.replicas` (topic): minimum ISR size needed to accept a write sent with `acks=all`
  - If ISR < `min`, leader rejects write and producer gets error; otherwise, write is acked and stored on at least `min` replicas

No AI/ML Processing

Kafka: write guarantees

- Strong durability:

  `acks=all` and `min.insync.replicas>=2`

- Example: `replication-factor=3` and `acks=all`; acknowledged write is on at least:
  - `min=1`: leader only
  - `min=2`: leader + 1 follower (recommended)
  - `min=3`: leader + 2 followers (strongest durability, lowest availability)
- Setting `min.insync.replicas`
  - The higher, the stronger the durability
  - But writes are rejected more often (when ISR < `min`)

No AI/ML Processing

Kafka: visibility to consumers

- Replication to followers is asynchronous
  - Followers fetch after the leader appends
- An event is committed once it has been replicated to all ISR
- High watermark: offset below which all events are committed
  - Lowest log end offset among ISR replicas
- Consumers read only below the high watermark
  - Even though they fetch from leader

```
leader    ■■■■■■■■■ log end = 10
follower  ■■■■■■■■  log end = 9
follower  ■■■■■■■   log end = 8
high watermark = 8 (consumers read offsets 0-7)
```

  - Exception: `read_committed` consumers also wait for the transaction to commit (read only up to the last stable offset, which is ≤ high watermark)

No AI/ML Processing

Kafka: delivery semantics

- **At-most-once**: guarantees that events are never re-delivered but may be lost
  - Producer disables retries (i.e., **`acks=0`**); loss is possible
  - Consumer commits its offset *before* processing event
    - What happens if consumer crashes after committing but before processing?
- **At-least-once**: guarantees no event loss, but events may be duplicated
  - Producer waits for all ISR (i.e., **`acks=all`**); no loss, but duplicates are possible (e.g., retry after a lost ack)
  - Consumer commits its offset *after* processing an event
    - What happens if consumer crashes before committing?
  - If `acks=1`: weaker semantics
    - Loss is possible: leader can crash after ack but before replication

https://kafka.apache.org/44/design/design/#message-delivery-semantics

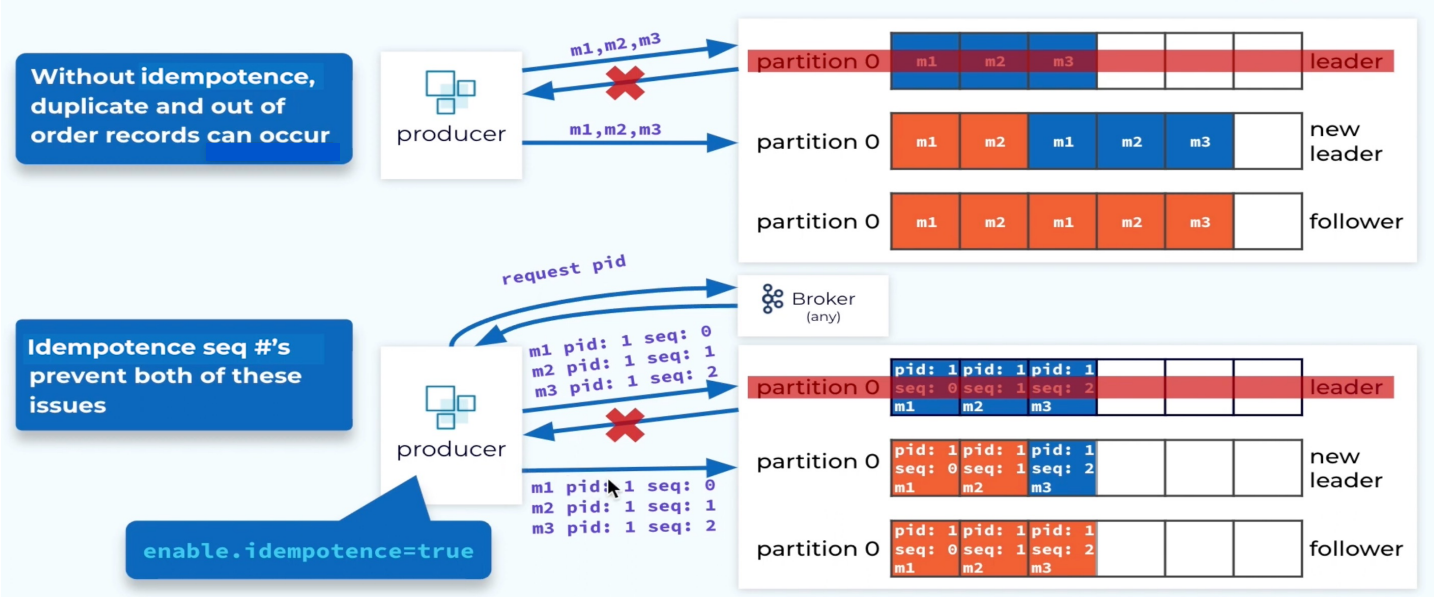
https://docs.confluent.io/kafka/design/delivery-semantics.html

No AI/ML Processing

Kafka: idempotent producer

- Goal: avoid duplicate writes to log
- How?
  - Each message sent to a topic partition carries producer ID and sequence number
  - Leader detects duplicates caused by retries and discards them (acknowledged but not appended)
- Also preserves partition ordering when retries occur (e.g., timeouts, leader failures)

Limit: producer ID does not survive restarts



No AI/ML Processing

Kafka: delivery semantics

- **Exactly-once**: the effect of each input event appears exactly once in the output; no loss and no duplicates
- At the cost of: higher latency and lower throughput
- What is needed:
  - Producer: `acks=all` (default) and idempotent producer (default)
  - Strong durability: `min.insync.replicas>=2` (with `replication-factor≥3`)
  - Transactions enabled: atomic writes across partitions + transactional offset commits + `read_committed` on consumers
- Not fully exactly-once
  - External side effects



https://www.conduktor.io/blog/exactly-once-semantics-when-it-works

No AI/ML Processing

Kafka: delivery semantics

- Take-away message: choose delivery semantics according to application
- At-most-once:
  - Fastest; messages may be lost
  - E.g., telemetry
- At-least-once (default):
  - No loss; duplicates possible (consumers should be idempotent)
    - Duplicates can occur on consumer (e.g., mid-processing crash) or after producer restarts (new producer ID)
  - E.g., order status update
- Exactly-once:
  - No loss and no duplicates; highest cost
  - E.g., financial transactions, inventory counters
- Common choice: at-least-once + idempotent consumers

No AI/ML Processing

Kafka: trade-offs

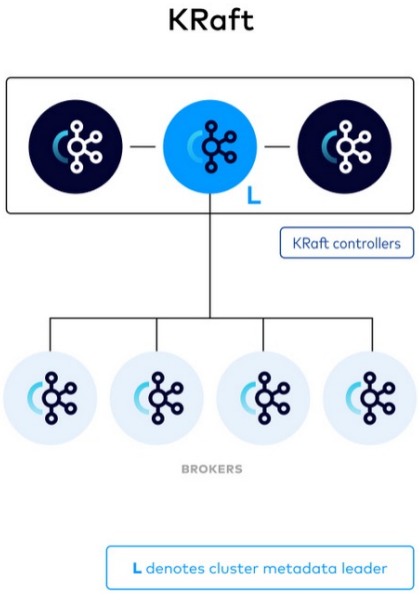
- Consistency vs. availability
  - In-sync replicas are up-to-date with leader and new leader is elected only among them
  - If leader fails and no in-sync replica is available, the partition is unavailable

- Latency vs. durability
  - Producers choose via `acks`
    - `acks=all`: highest durability, highest latency (all ISRs)
    - `acks=1`: lower latency, but messages may be lost if leader crashes before replication
    - `acks=0`: lowest latency, highest risk of loss

No AI/ML Processing

Kafka: broker coordination

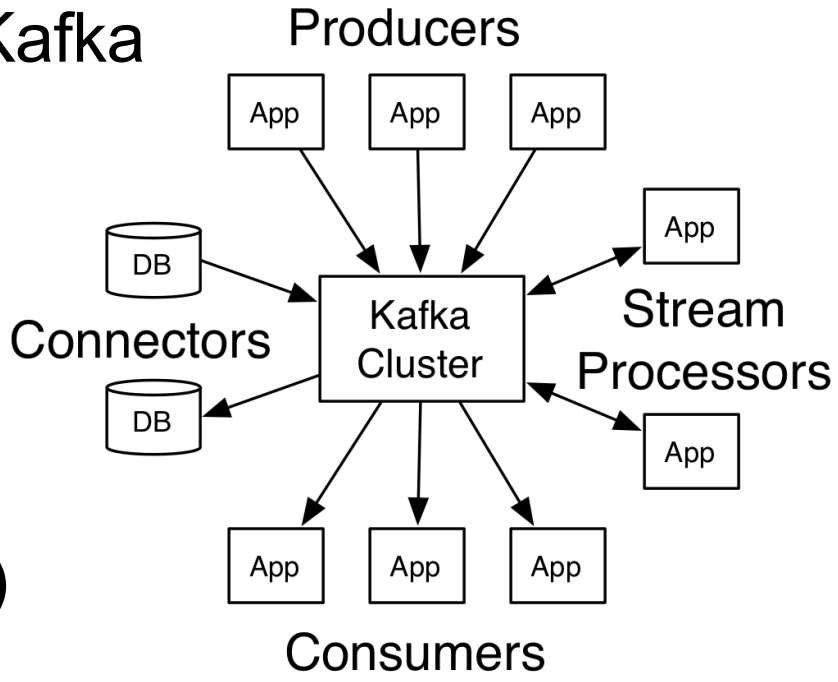
- Brokers need a consistent view of the cluster
- When a partition leader fails, new leader must be elected
- Done by a controller, backed by a consensus protocol
- KRaft (Kafka Raft): default
  - Metadata is stored in an internal replicated log of Kafka, kept consistent by Raft across a small quorum of controller nodes
  - Raft elects the active controller, which elects partition leaders; brokers fetch the metadata log and cache it



- Before Kafka 4.0: metadata and controller election relied on ZooKeeper
  - External distributed coordination system
  - X Additional system to operate, scalability limits as partitions grow, slower controller failover

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, …
- Streams API: for real-time processing of streaming data
- Admin API: manages clusters and resources (topics, partitions, brokers, etc.)

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

No AI/ML Processing

Kafka: client libraries

- Kafka officially provides an SDK for Java
- Community-driven Kafka client libraries for other languages, including
  - Go
    - Confluent https://github.com/confluentinc/confluent-kafka-go
    - Sarama https://github.com/IBM/sarama
  - Python
    - Confluent https://github.com/confluentinc/confluent-kafka-python

No AI/ML Processing

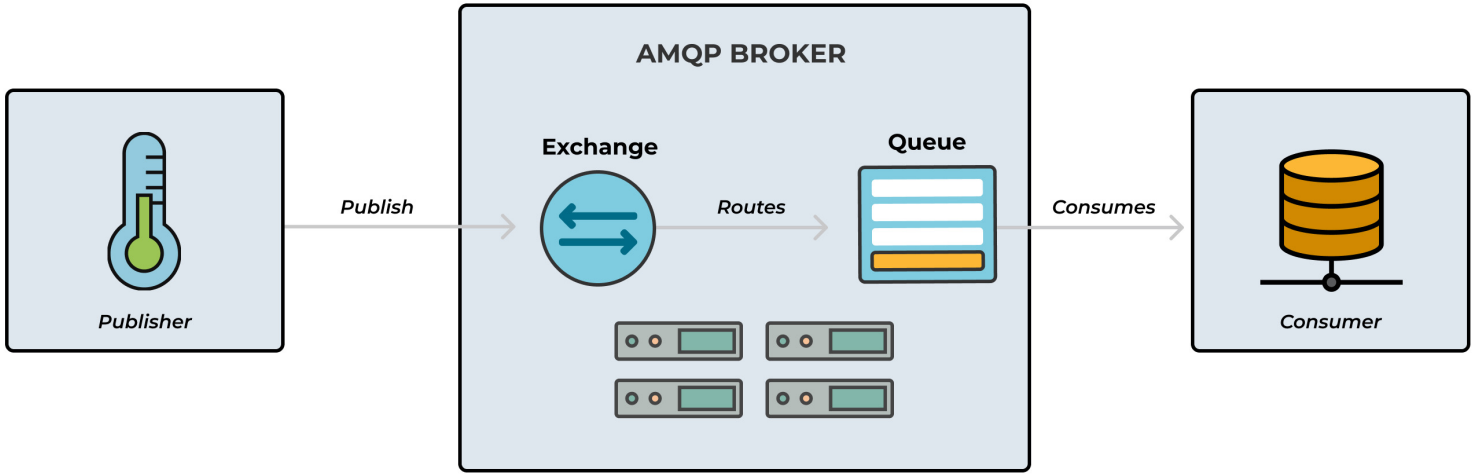
Interacting with MOM: messaging protocols

- Application-layer open standard protocols for MOM interaction
- **AMQP** (Advanced Message Queuing Protocol)
- MQTT (Message Queue Telemetry Transport)
  https://mqtt.org
  - Lightweight, binary protocol for low-bandwidth and low-power devices, used in IoT applications
- Goals:
  - Platform- and vendor-agnostic
  - Interoperability: facilitates communication between different MOMs

No AI/ML Processing

Messaging protocols and IoT

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



- Advantages of 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

No AI/ML Processing

AMQP: features

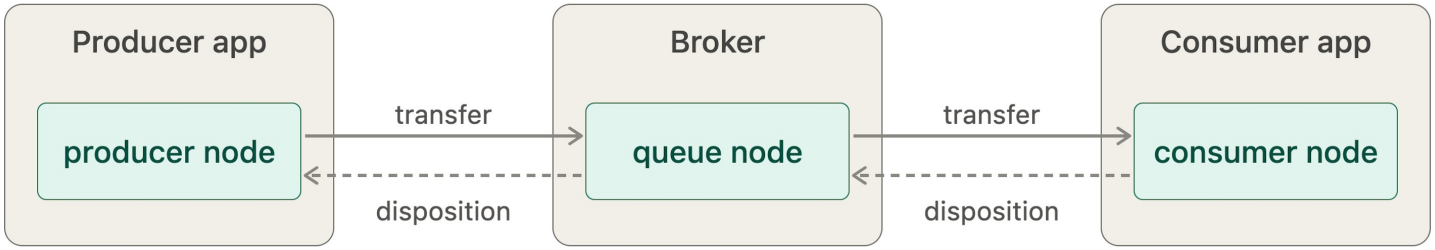
- Open standard (ISO/IEC 19464) messaging protocol between clients and brokers

  https://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf
- Application-level, binary protocol
  - Over TCP, also TLS and SASL for authentication
  - Delivery guarantees
- Version 1.0 defines only protocol and message format
  - No server model: queues, topics and routing left to broker
  - Interoperability between clients and brokers of different vendors
- Supports queue-based and pub/sub patterns
- Implementations
  - Brokers: Apache ActiveMQ, Azure Event Hub, RabbitMQ https://www.rabbitmq.com/docs/amqp
  - Client libraries: Apache Qpid Proton, Azure go-amqp (Go)

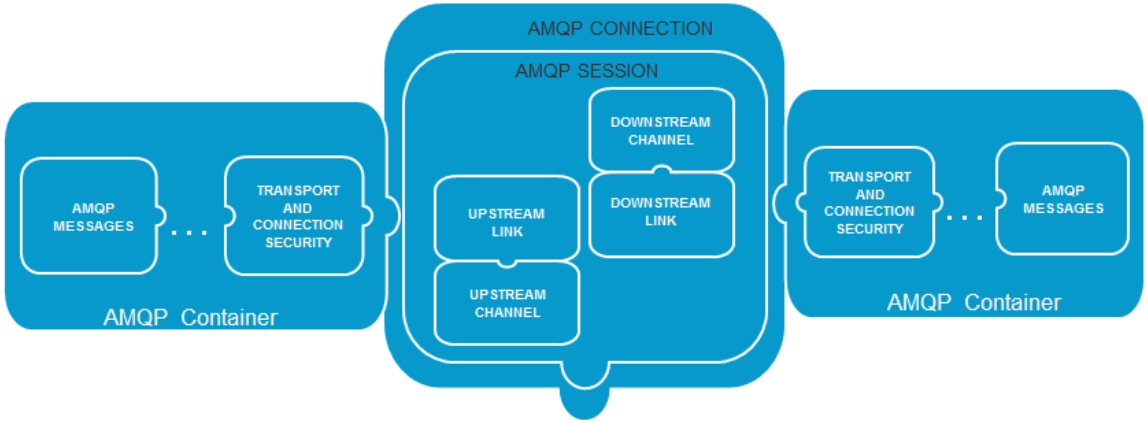
No AI/ML Processing

AMQP: architecture

- 3 actors: producers, consumers, and brokers



- AMQP containers (applications, brokers) host nodes
  - Node: named entity that produces, consumes or stores messages (e.g., queue)
- AMQP connection (over TCP) carries sessions, which carry links
  - Link: one-way path between two nodes



AMQP: messages and flow control

- Message flow
  - Sender sends message in a transfer
  - Receiver controls rate by granting link credit (flow control)
  - Receiver reports outcome with disposition (e.g., `accepted`)
- A delivery is settled once its outcome is known
  - Producer: waits for broker outcome
  - Consumer: settles only after processing; if it stops before settling, broker redelivers message: at-least-once
- The receiver grants link credit: how many messages the sender may send
  - Credit returns as deliveries are settled
  - Credit = 1: one unsettled message per consumer at a time
    - See example 2 for fair dispatching among workers
  - Larger credit: higher throughput, but load may be uneven

No AI/ML Processing

RabbitMQ

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



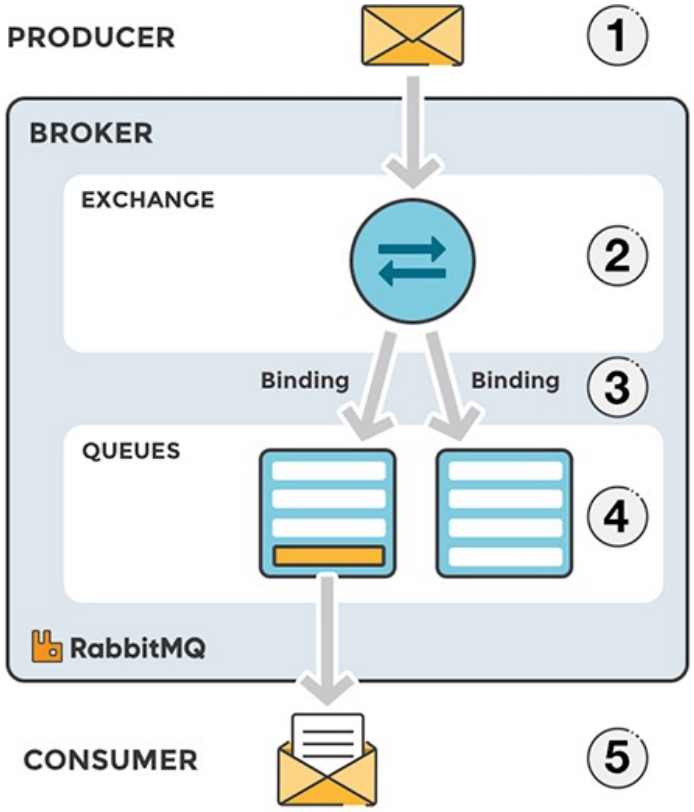
- Multiple messaging protocols supported
  - AMQP, MQTT, and STOMP
- Push delivery model (can also support pull)
- FIFO ordering guarantee at queue level
- Cross-platform
  - Can run on multiple operating systems and cloud environments
- Wide range of development tools (Java, Go, Python, …)

No AI/ML Processing

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

No AI/ML Processing

RabbitMQ: quorum queues

- Queue type that is durable and replicated across the nodes of RabbitMQ cluster
  - Based on Raft consensus algorithm
- Each queue has one leader and several followers
  - Publishers and consumers interact with leader
  - Message is committed when a majority (quorum) of the replicas has stored it
  - Only then the publisher gets the outcome accepted
- Fault tolerance
  - If leader fails, replicas elect a new one
  - Queue works while a majority of replicas is up
  - Without majority queue is unavailable (consistency over availability)

No AI/ML Processing

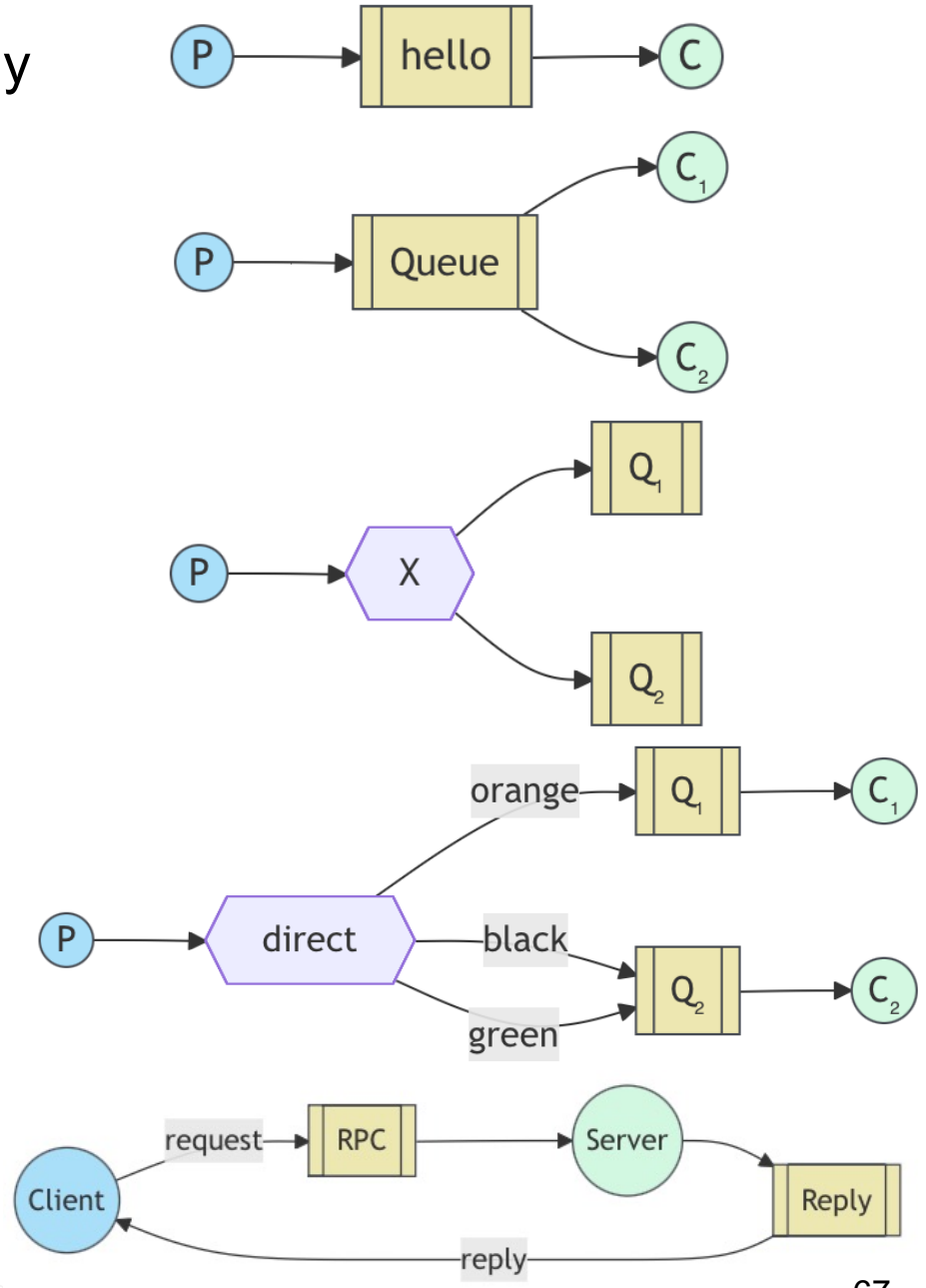
RabbitMQ: delivery semantics

- At-most-once delivery
  - If the consumer fails before processing the message, the message will be lost
- At-least-once can be configured
  - Consumer sends an explicit ack after successfully processing the message

No AI/ML Processing

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

No AI/ML Process

RabbitMQ and Go

- Use RabbitMQ, Go and AMQP 1.0 for:

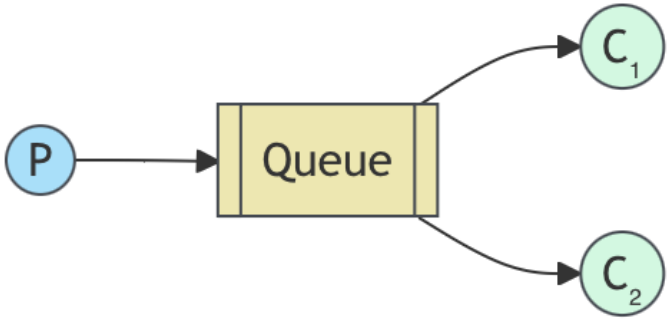
Ex. 1: Message queue

https://www.rabbitmq.com/tutorials/tutorial-one-go-amqp10



Ex. 2: Work queue

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



Code on Teams/course website

No AI/ML Processing

RabbitMQ and Go

- Setup:

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
   ```

   – Useful commands for `rabbitmqctl`

   ```
   list_channels
   list_consumers
   list_queues
   reset
   ```

   – Web UI for management and monitoring (port 15672)

2. Install Go AMQP 1.0 client library

   ```
   $ go get github.com/rabbitmq/rabbitmq-amqp-go-client
   ```

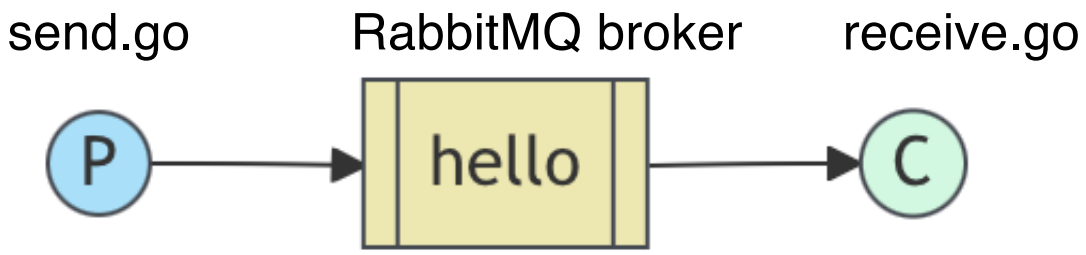
   See https://pkg.go.dev/github.com/rabbitmq/rabbitmq-amqp-go-client

No AI/ML Processing

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

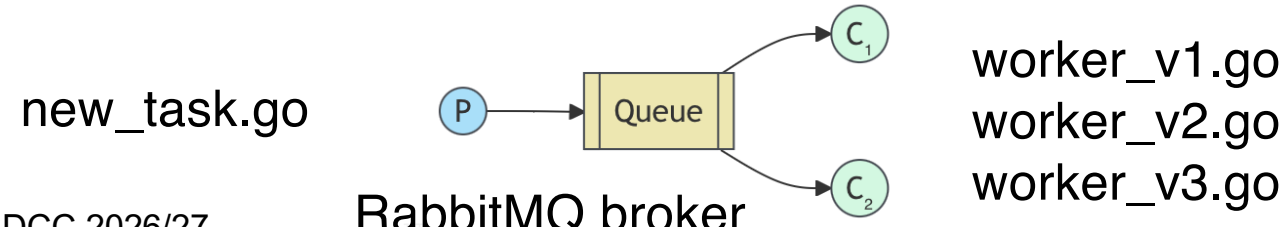


No AI/ML Processing

RabbitMQ and Go: example 2

2. Work queue pattern

- Version 1 (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
- Version 2 (worker_v2.go):
  - To not lose a message when a worker crashes after receiving it but before finishing processing, settle the message only after the task is done
  - Which delivery semantics?
- Version 3 (worker_v3.go):
  - Fair task distribution by preventing a worker from receiving a new message while it still has unacknowledged messages



Multicast communication

- **Multicast communication**: group communication pattern where data is sent to *multiple* receivers 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
- Unicast replication (source sends individual copies to each receiver) does not scale
  - 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 hosts

No AI/ML Processing

Application-level multicast

- Basic idea:
  - Organize nodes into an overlay network
  - Use overlay network to disseminate data
  - Overlay: structured or unstructured
- Structured application-level multicast
  - Explicit communication paths
  - Which overlay topology?
    - 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

No AI/ML Processing

Unstructured application-level multicast

- How to realize unstructured application-level multicast

✓ Flooding

  – Node *P* sends 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 *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: 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 forwards the message to another randomly chosen subset, and so on

No AI/ML Processing

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

No AI/ML Processing

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 *N* (number of nodes)
  - Reliability and robustness: thanks to message redundancy

No AI/ML Processing

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

No AI/ML Processing

Strategies to spread updates

- Let's consider the two principal operations

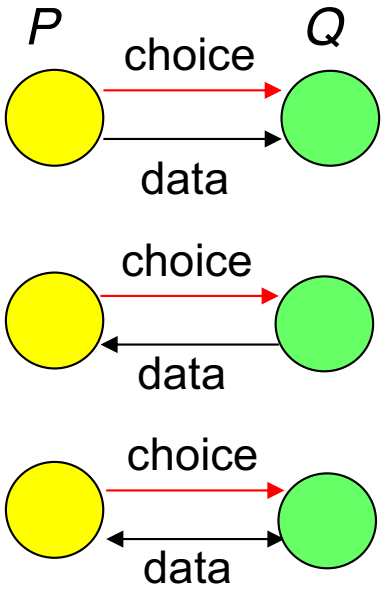
1. Anti-entropy: each node periodically selects another node randomly and exchanges updates (i.e., state differences), with the goal of having identical states on both node afterwards

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

No AI/ML Processing

Anti-entropy

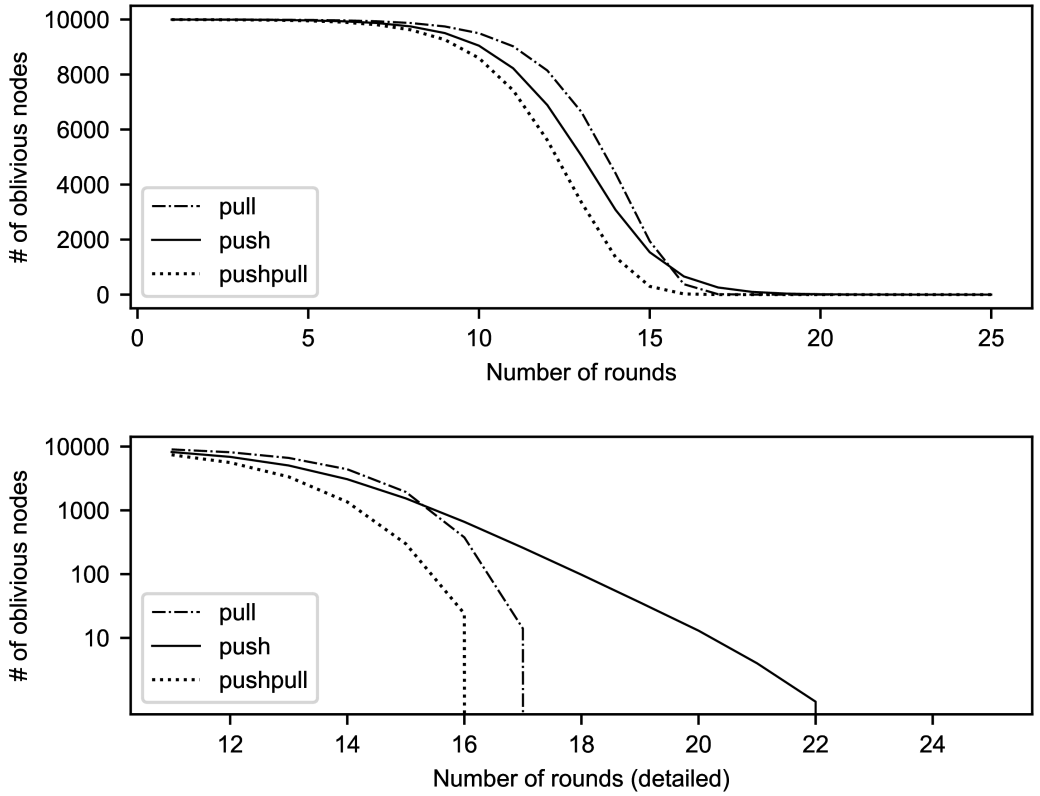
- Goal: increase node state similarity, thus decreasing "disorder" (the reason for the name!)
- Node *P* selects node *Q* randomly: how does P update *Q*?
- Different update strategies:
  1. Push: *P* only pushes its own updates to *Q*
  2. Pull: *P* only pulls new updates from *Q*
  3. Push-pull: *P*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



No AI/ML Processing

Anti-entropy: performance

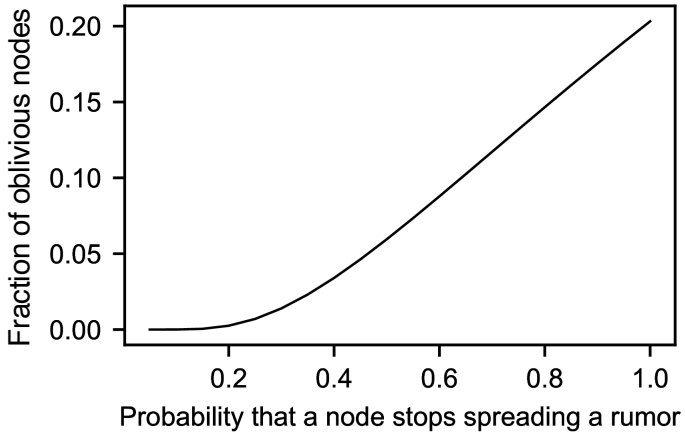
- Push-pull
  - Fast and message-saving strategy: takes O(ln *N*) rounds to disseminate updates to *N* nodes, using O(*N* ln ln *N*) messages
  - *Round* (or *gossip cycle*): time interval in which every node initiates an exchange



Rumor spreading

- Node *P*, having an update, contacts a randomly chosen node *Q* (F=1) 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

No AI/ML Processing

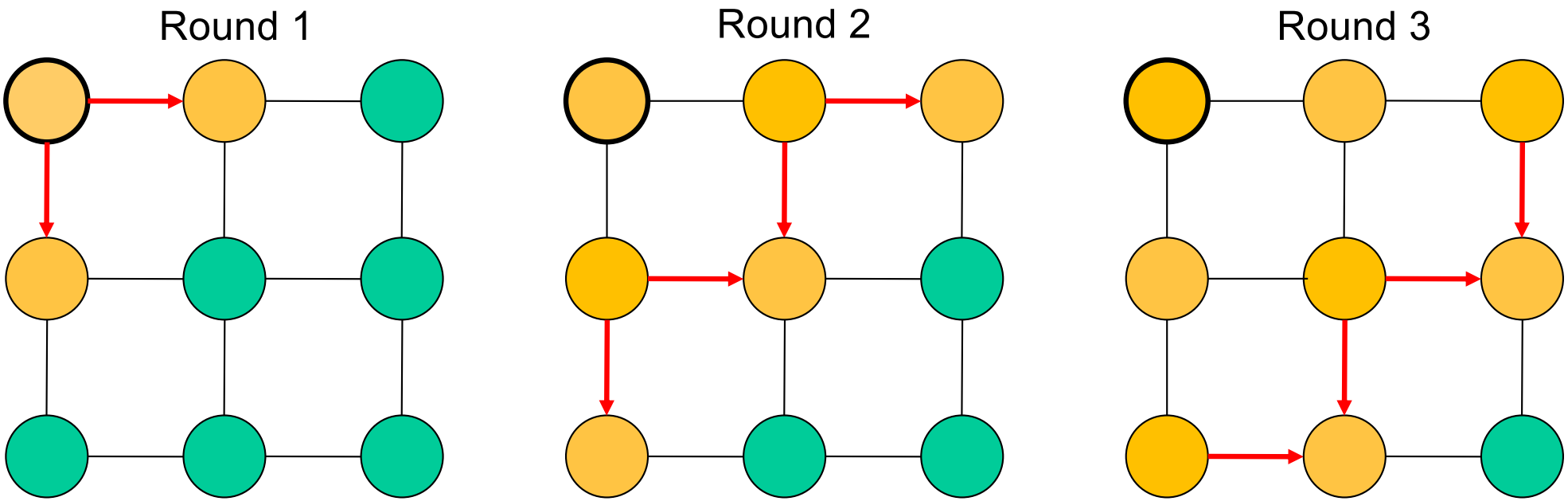
Framework for gossiping protocols

- 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 CockroachDB)
  - Data exchanged
    - What is exchanged is highly application-dependent
    - Choice of update strategy
  - Data processing
    - Again, highly application-dependent

No AI/ML Processing

Gossiping vs flooding: example

- Information dissemination: 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 (excluding the sender)
  - Message is eventually discarded when TTL reaches 0



Sent messages: 10
Reached nodes: 8 out of 9

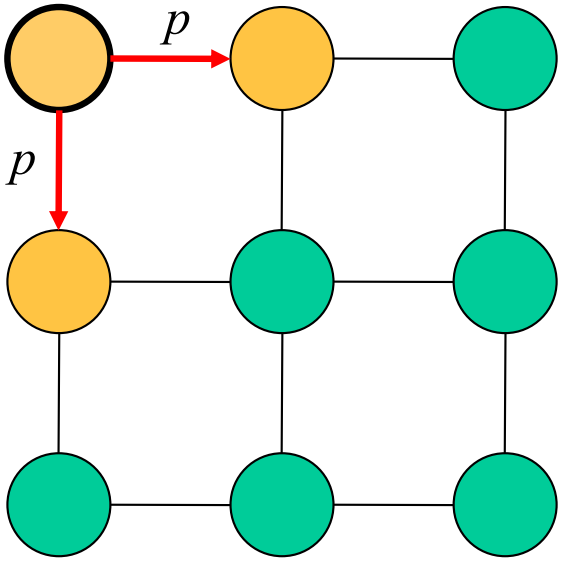
No AI/ML Processing

Gossiping vs flooding: example

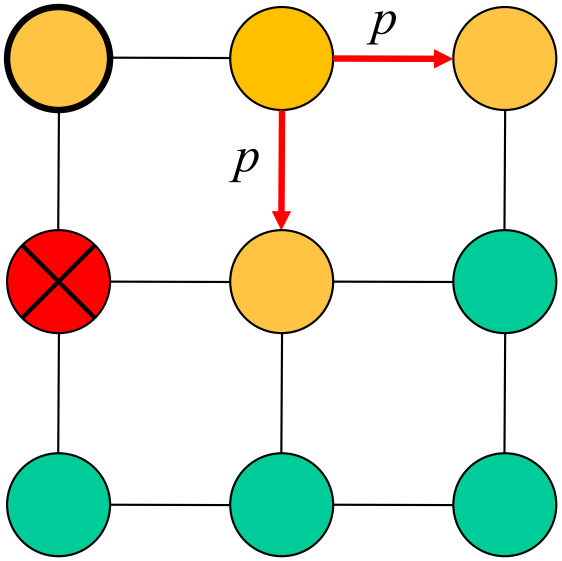
- Example of rumor spreading algorithm
  - The node sends the message with probability p to each of its neighbors (excluding the sender)

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

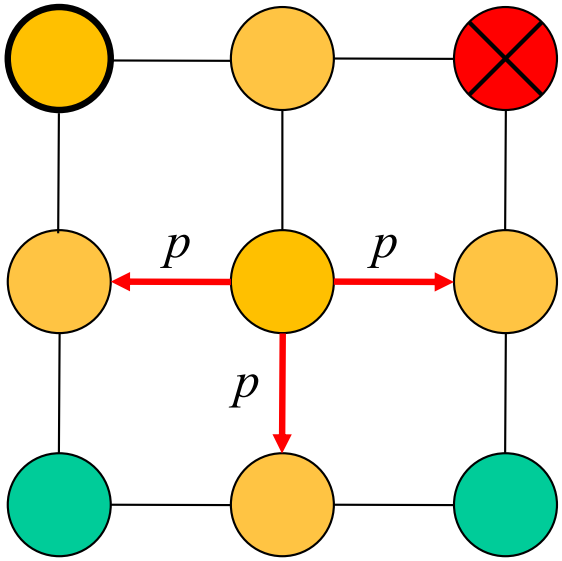
Round 1



Round 2



Round 3



Sent messages: 7
Reached nodes: 7 out 9

No AI/ML Processing

Gossiping vs flooding

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

No AI/ML Processing

Other application domains of gossiping

- Besides information dissemination, many applications in large-scale distributed systems
- Group membership
  - To know the nodes in DS
- Peer sampling
  - To select a partial view of nodes in DS
- Resource management
  - Including monitoring and failure detection
- Distributed computation
  - 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

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

No AI/ML Processing

Gossiping: case studies

1. Blind counter rumor mongering: example of gossip-based dissemination protocol

2. Bimodal multicast: uses gossiping to provide reliable multicast

3. Gossip-based failure detection

No AI/ML Processing

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

No AI/ML Processing

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

No AI/ML Processing

Bimodal multicast

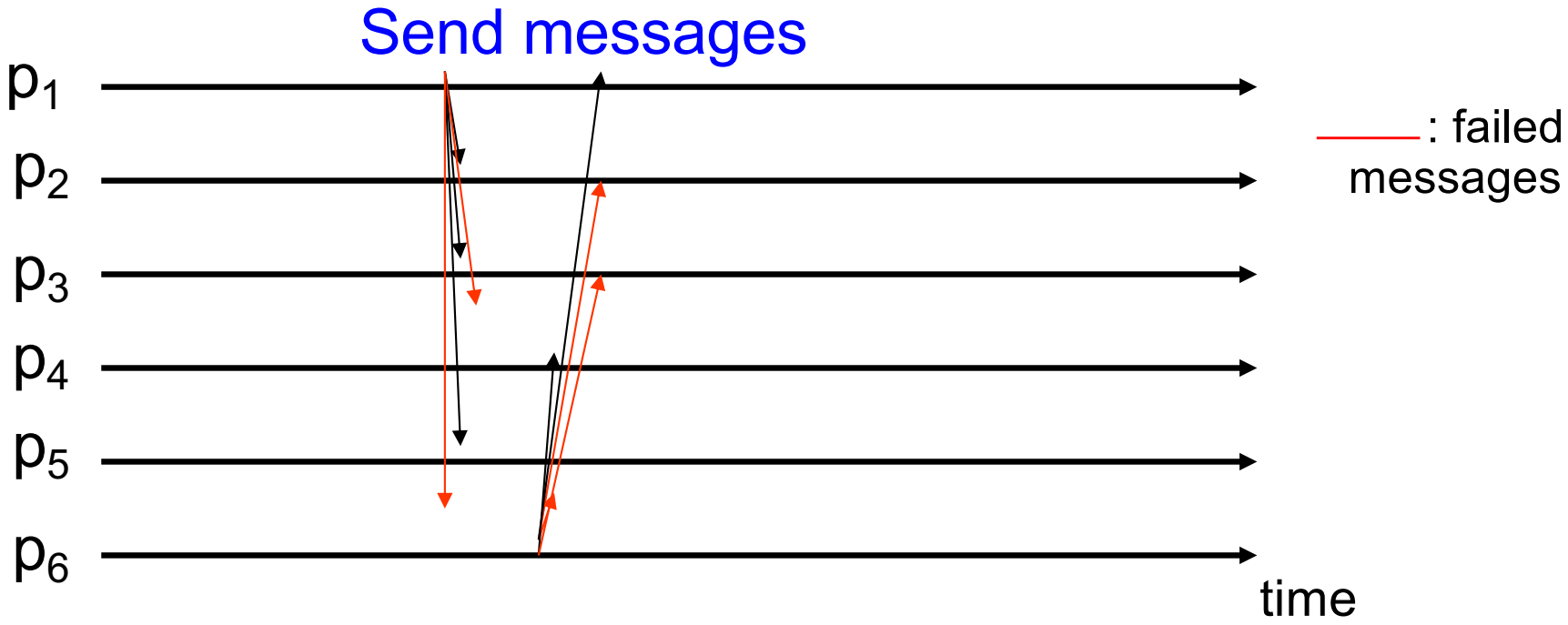
- Aka pbcast (probabilistic broadcast)
- 2-phase protocol:
  1. **Message distribution**: a process sends a multicast message with no reliability guarantee
  2. **Gossip repair**: after receiving a message, a process begins to gossip about it to a set of peers
     - Gossip occurs at regular intervals, 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

No AI/ML Processing

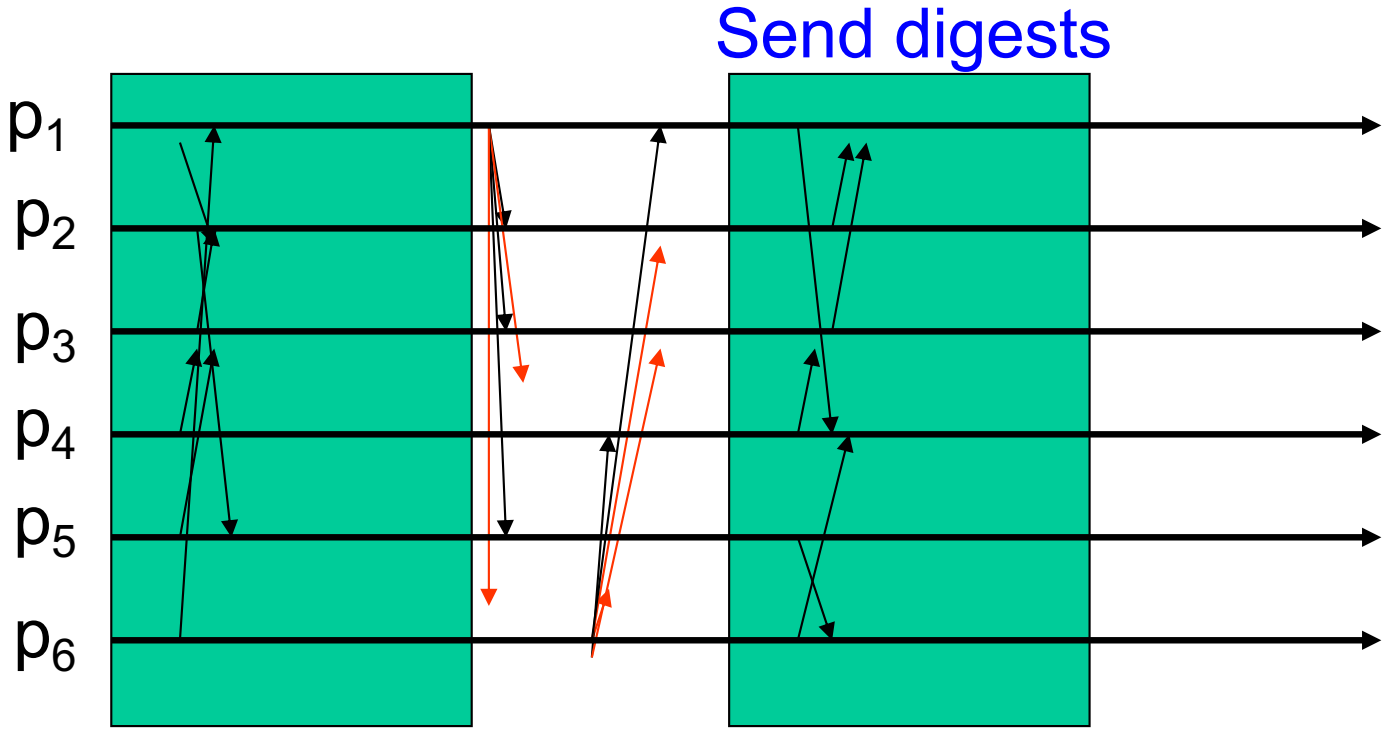
Bimodal multicast: message distribution



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

No AI/ML Processing

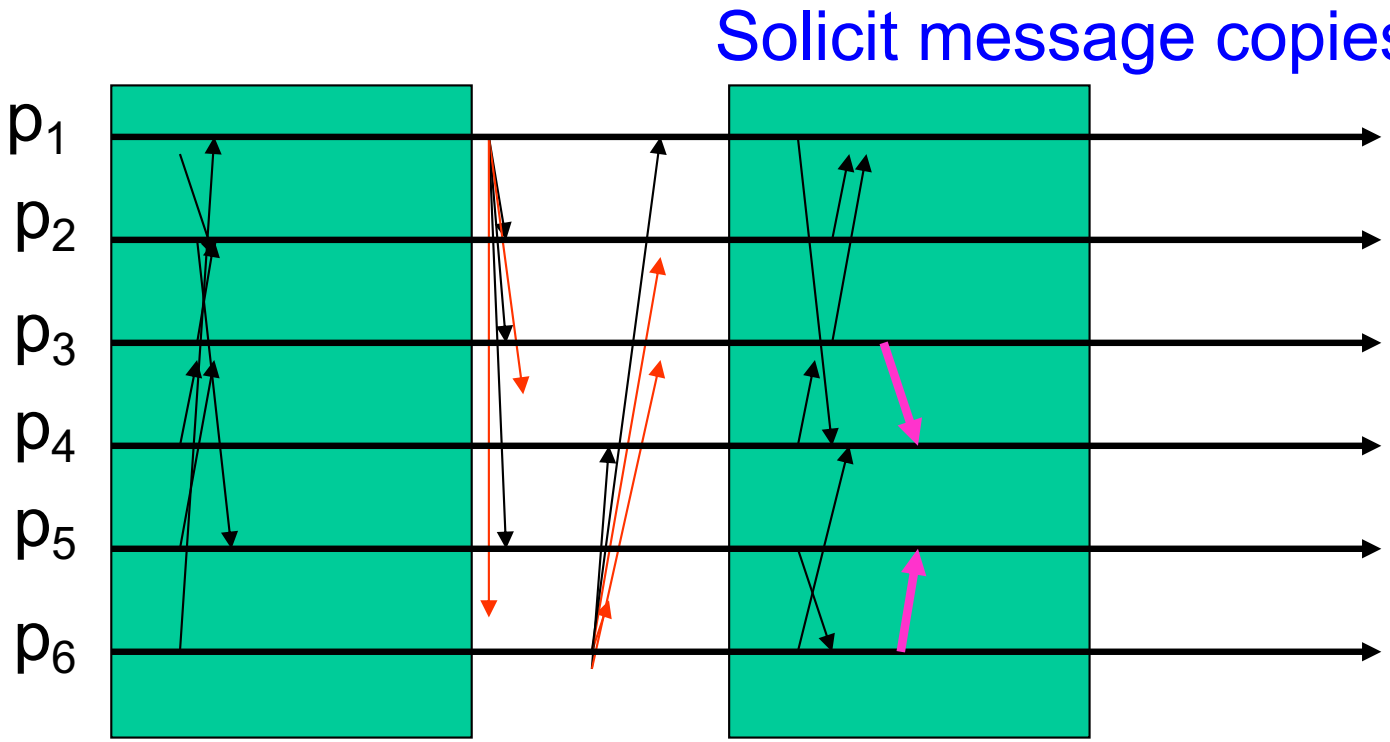
Bimodal multicast: gossip repair



- 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

No AI/ML Processing

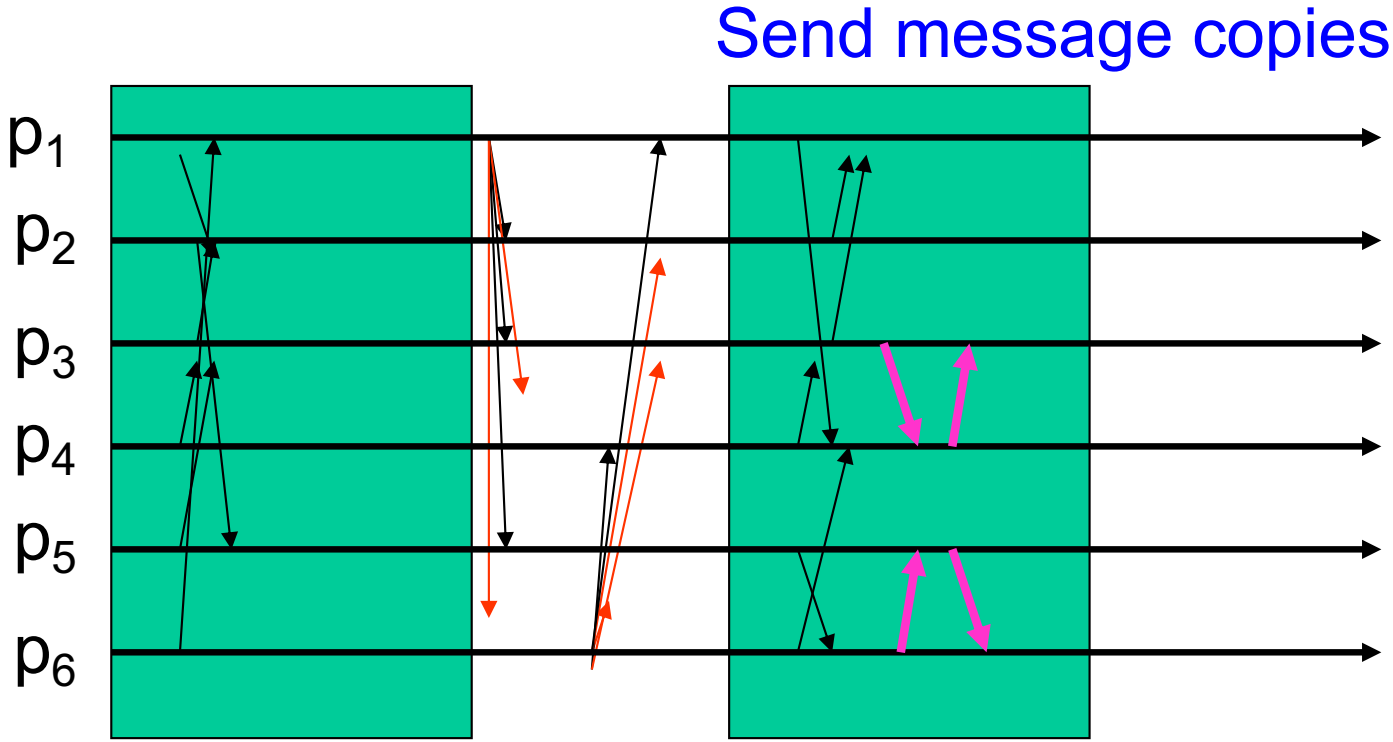
Bimodal multicast: gossip repair



- 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

No AI/ML Processing

Bimodal multicast: gossip repair



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

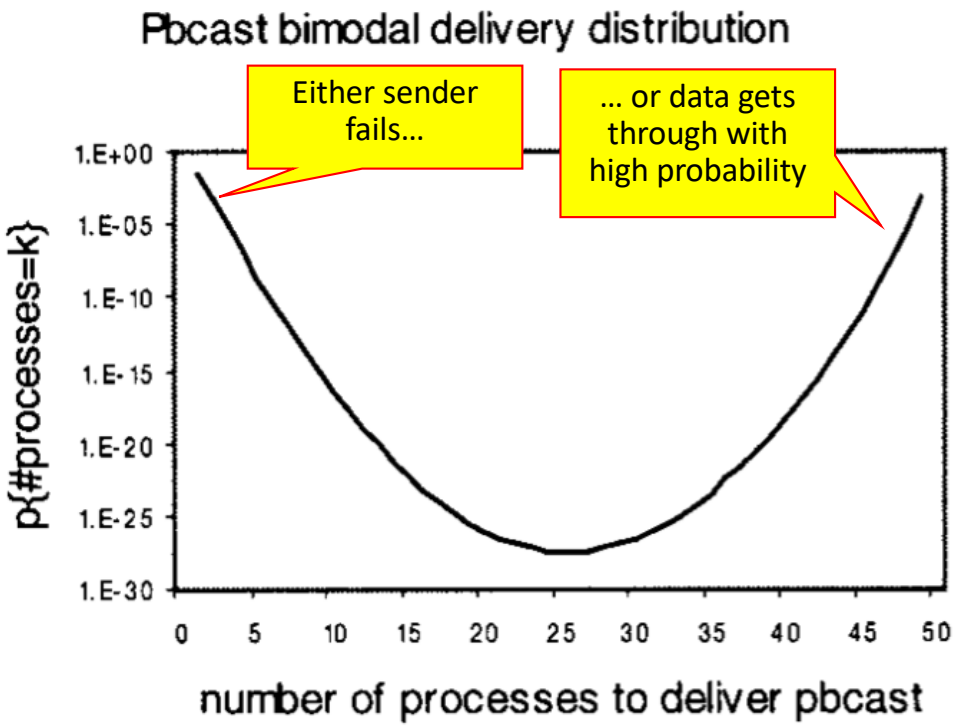
No AI/ML Processing

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)

No AI/ML Processing

Failure detection

- Failure detector: find which nodes are suspected to have crashed
- Requirements in large-scale systems
  - Completeness: eventually detect every crashed node
  - Accuracy: few false positives
  - Speed of detection
  - Scalability: per-node load does not grow with *N*
- Heartbeating approaches
  - Centralized: all nodes send heartbeats to one monitor
    - X Bottleneck, single point of failure
  - All-to-all: every node sends heartbeats to all others
    - X $O(N^2)$ messages per period
  - Gossip-based: each node sends one message per period to a random node
    - X No central point, load per node independent of *N*

No AI/ML Processing

Failure detection: gossip-based

- Each node keeps a list with one entry per known node *j*
  - Heartbeat counter H_j and local time T_j of last increase of H_j
- Every T_{gossip} time units, a node
  1. Increments its own heartbeat counter
  2. Sends its counters to a randomly chosen node
- On receiving, merge the counters entry by entry
  - Keep the higher value
  - If H_j increased, set T_j = current local time
- Failure detection
  - If H_j has not increased for T_{fail} time units, node *j* is suspected failed
  - Failed entry is kept for a further $T_{cleanup}$ before deletion, so that stale gossip about *j* cannot re-insert it as alive
- Only local clocks are used: no clock synchronization is needed

No AI/ML Processing

Failure detection: gossip-based

- Example: 4 nodes, T_{fail}=20; node 2 merges at local time = 70

	Node 2 before counter @update	From node 4 counters	Merge	Node 2 after counter @update age
Node 1	10 @64	11	higher → update	11 @70 age 0
Node 2	15 @70	14	lower → ignore	15 @70 age 0
Node 3	9 @45	9	same → no change	9 @45 age 25
Node 4	12 @62	13	higher → update	13 @70 age 0

Node 3: age 70 – 45 = 25 > T_{fail} → suspected failed

Counter increased No change Suspected failed

No AI/ML Processing

Failure detection: gossip-based

- Setting
  - T_{gossip}: trade-off between detection speed and network load
  - T_{fail} must exceed time taken by a heartbeat reach all nodes ($O(\log N)$ gossip rounds)
    - Too short: false positives; too long: slow detection
  - $T_{cleanup}$: same order of T_{fail}
- Robust: tolerates message loss and node crashes (heartbeats reach a node through many paths)
- Cost: each message carries the full list, $O(N)$ entries
  - Reduce by gossiping only a part of the list, or only recent changes
- Risk: slow or overloaded node, or network partition, can look like a failure

No AI/ML Processing

Failure detection: gossip-based

- How used in practice
- SWIM (used by Serf and Consul)
  https://doi.org/10.1109/DSN.2002.1028914
  - Each period, a node pings one random node; if no ack, it asks *k* other nodes to ping it indirectly
  - Still no ack: the node is marked *suspect*, then declared failed after a timeout
  - Membership changes are piggybacked on ping messages: constant load per node, independent of *N*
- Cassandra: gossip heartbeats combined with φ accrual detector
  https://dl.acm.org/doi/pdf/10.1145/1773912.1773922
  - Outputs a suspicion level based on the history of heartbeat arrival times, not fixed T_{fail}
- Redis Cluster: nodes exchange gossip messages and mark a node as failed when enough nodes agree

No AI/ML Processing

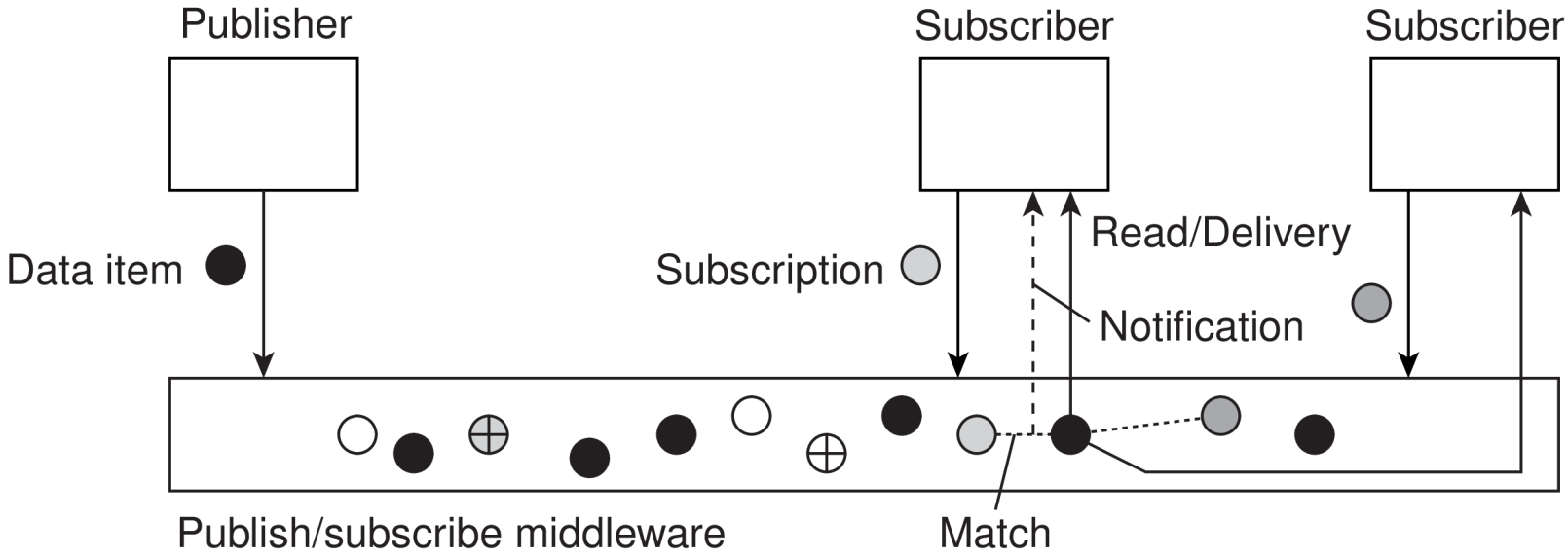
Recent applications of gossiping

- Gossip-based learning in the context of distributed machine learning
  - SABD course
- Swarm robotics
  - Robots use gossip to share sensor observations across dynamic, mobile networks without central control

  Marco Dorigo, https://www.youtube.com/watch?v=gOcCcOHKZX0
- Decentralized RAG (Retrieval-Augmented Generation) https://arxiv.org/html/2611.07928v1

Pub-sub event matching

- Distributed pub-sub event matching: a case study to tie together routing, decoupling, and scalability



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

No AI/ML Processing

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

- Distribute the matching load across servers to achieve scalability
- Simple solution: partition subscriptions among servers
  - Each server stores and matches a subset of subscriptions
  - Topic-based pub/sub: `server = hash(topic)`

1. Hierarchical architecture
   - Master receives events and subscriptions and distributes them to workers
   - x Single master: bottleneck and SPOF
2. Flat architecture
   - No master: every server knows the partitioning (hash function and server list)
   - Any server that receives a subscription or event computes `hash(topic)` and forwards it to the responsible server
   - Dynamic scenario?

Event matching: distributed architecture

- Alternative solution: route subscriptions or events without global partitioning via decentralized overlay networks

3. Unstructured overlay: use flooding or gossiping
   - Full dissemination: store a subscription at one server, but disseminate notifications to all servers; matching is distributed across servers
   - Selective routing: propagate subscription filters to prune paths toward uninterested servers, reducing unnecessary messages
4. Structured overlay: use a DHT to disseminate subscriptions or events

References

- Chapter 4 and Section 5.6 of van Steen & Tanenbaum book
- Kafka doc https://kafka.apache.org/documentation
- Conduktor'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
- Video: Confluent's Kafka Internals https://www.youtube.com/playlist?list=PLa7VYi0yPIH14oEOfwbcE9_gM5lOZ4ICN
- RabbitMQ https://www.rabbitmq.com
- Montresor, Gossip and epidemic protocols, *Wiley Encyclopedia of Electrical and Electronics Engineering*, 2017 https://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

No AI/ML Processing