TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Communication in Distributed Systems
Part 2

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2024/25

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Message-oriented communication

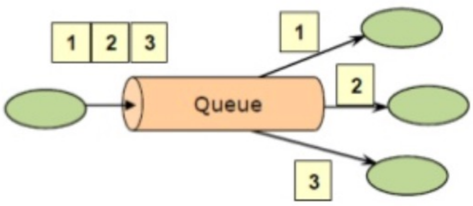
- RPC improves distribution transparency with respect to socket programming
- But still synchrony between interacting entities
  - Over time: caller waits the reply
  - In space: shared data
  - Functionality and communication are coupled
- Which communication models to improve decoupling and flexibility?
- Message-oriented communication
  - Transient
    - Berkeley socket
    - Message Passing Interface (MPI): see "Sistemi di calcolo parallelo e applicazioni" course
  - Persistent
    - **Message Oriented Middleware (MOM)**

Message-oriented middleware

- Communication middleware that supports sending and receiving messages in a persistent way
  – MOM offers intermediate-term storage capacity for messages
- Loose coupling among system/app components
  – Decoupling in time and space
  – Can also support synchronization decoupling
  – Goals: increase performance, scalability and reliability
  – Typically used in serverless and microservice architectures
- Two patterns:
  – **Message queue**
  – **Publish-subscribe** (pub/sub)
- And two related types of system:
  – **Message queue system** (MQS)
  – **Pub/sub system**

Queue message pattern

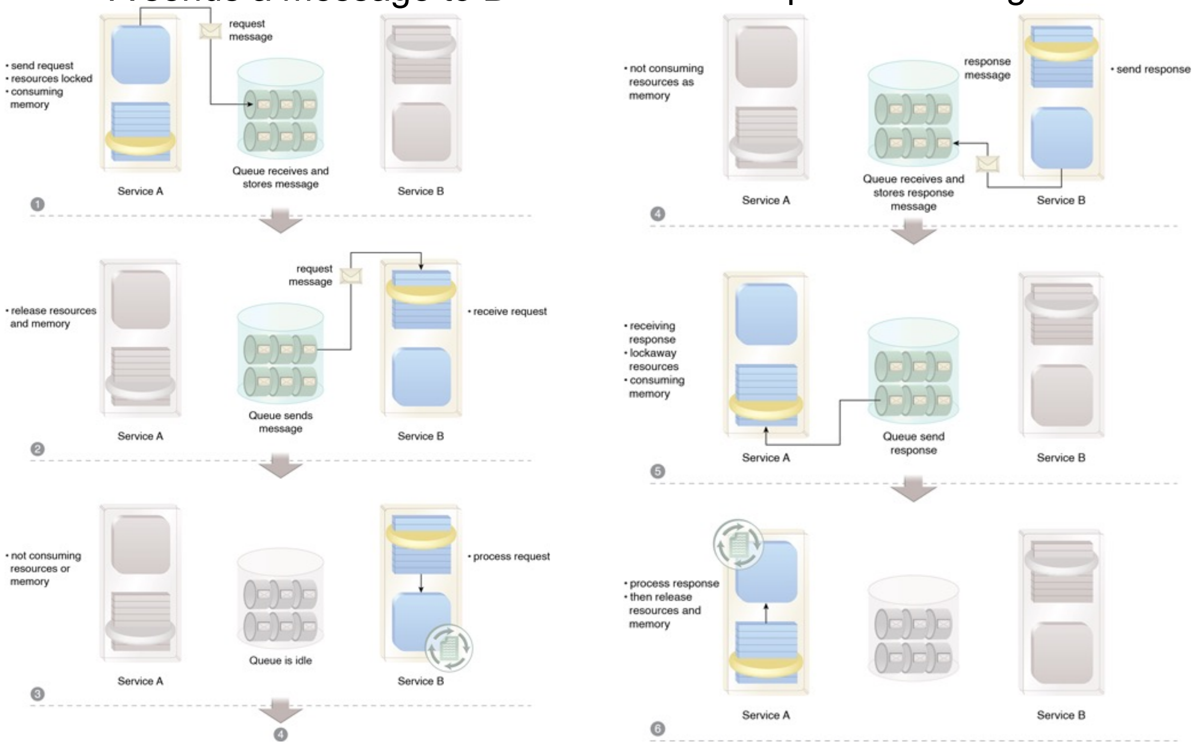
- Messages sent to queue are stored until they are retrieved by consumer
- Multiple producers can send messages to queue
- Multiple consumers can receive messages from queue
- But communication is one-to-one: producer's message is delivered to a single consumer



- When to use message queues
  – Examples: task scheduling, load balancing, logging or tracing

Queue message pattern

A sends a message to B B issues a response message back to A

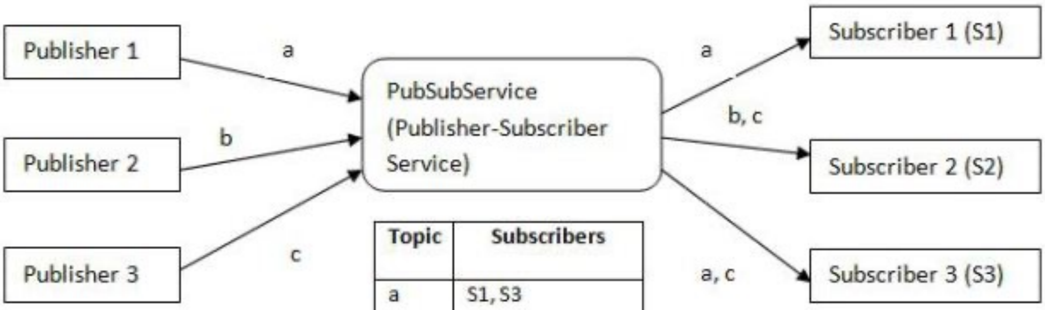


Message queue API

- Typical calls of MQS:
  - **put**: non-blocking send
    - Insert message into queue
  - **get**: blocking receive
    - Block until queue is nonempty and receive a message
    - Variant: allow searching for specific message in queue
  - **poll**: non-blocking receive
    - Check queue and receive message if available
    - Never block
  - **notify**: non-blocking receive
    - Install handler (callback function) to be automatically called when a message is put into queue

Publish/subscribe pattern

- Application components can publish asynchronous messages (e.g., event notifications), and/or declare their interest in message topics by issuing a *subscription*
- Each message can be delivered to multiple consumers



Topic	Subscribers
a	S1, S3
b	S2
c	S2, S3

Publish/subscribe pattern

- Multiple consumers can subscribe to topic with or without filters
- Subscriptions are collected by an *event dispatcher* component, responsible for routing events to all matching subscribers
  - For scalability reasons, its implementation is distributed
- High degree of decoupling among components
  - Easy to add and remove components: appropriate for dynamic environments

Publish/subscribe pattern

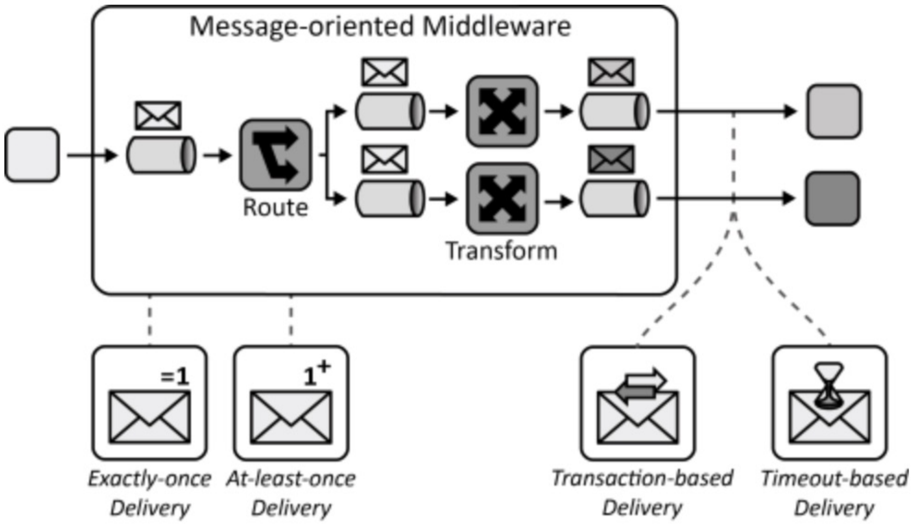
- A sibling of message queue pattern but further generalizes it by delivering a message to multiple consumers
  - Message queue: delivers messages to *only one* receiver, i.e., one-to-one communication
  - Pub/sub channel: delivers messages to *multiple* receivers, i.e., one-to-many communication

Publish/subscribe API

- Typical calls of pub/sub system:
  - **`publish(event)`**: called by publisher to publish an event
    - Events can be of any data type and may contain meta-data
  - **`subscribe(filter_expr, notify_cb, expiry) → sub_handle`**: called by subscriber to subscribe to events
    - Takes as input: filter expression, reference to notify callback for event delivery, and expiry time for subscription
    - Returns subscription handle
  - **`notify_cb(sub_handle, event)`**: called by pub/sub system to deliver to subscribers a matching event
  - **`unsubscribe(sub_handle)`**: called by subscriber to unsubscribe

MOM functionalities

- MOM handles the complexity of addressing, routing, availability of communicating application components (or applications), and message format transformations



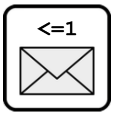
https://www.cloudcomputingpatterns.org/message_oriented_middleware

MOM functionalities

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

Delivery semantics in MOM

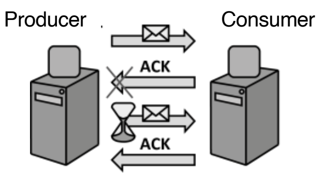
At-most-once delivery



- Message will be delivered not more than once
- Messages may be lost but are not redelivered

At-least-once delivery

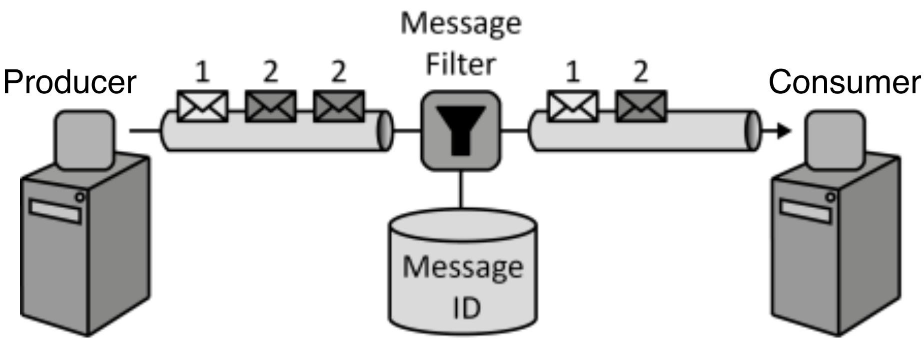
- Messages are never lost but they may be delivered more than once
- Design application to be *idempotent* (not affected adversely when processing same message more than once)
- *How can MOM ensure that messages are received successfully?*
  - Consumer sends ack for each message and MOM resends message if ack is not received



Delivery semantics in MOM

Exactly-once delivery

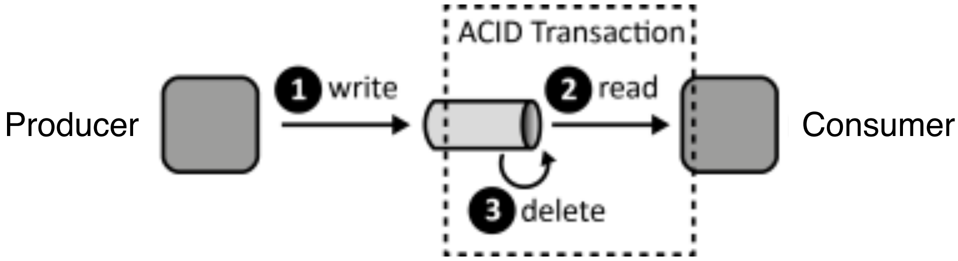
- *How can MOM ensure that a message is delivered only exactly once to a consumer?*
  - MOM also filters message duplicates
    - Upon creation, each message is associated with a unique ID, which is used to filter message duplicates during their traversal from producer to consumer
  - In addition, messages must survive MOM failures



Delivery semantics in MOM

Transaction-based delivery

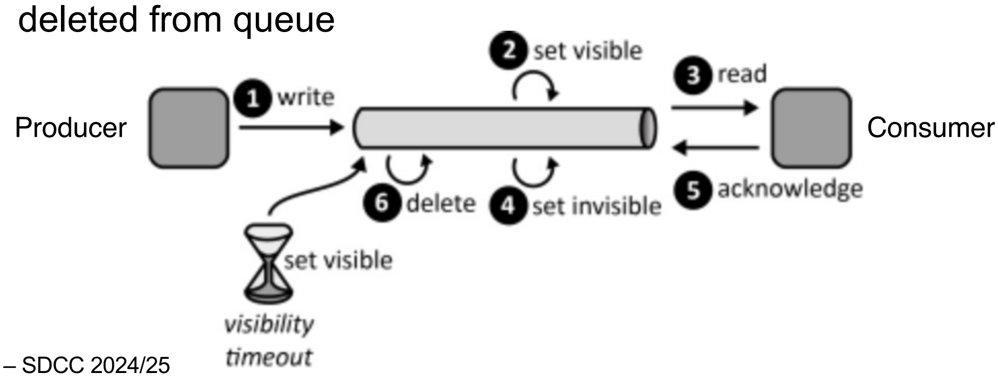
- *How can MOM ensure that messages are only deleted from a message queue if they have been received successfully?*
  – MOM and consumer participate in a transaction: read and delete operations are performed within a transaction, thus guaranteeing ACID behavior



Delivery semantics in MOM

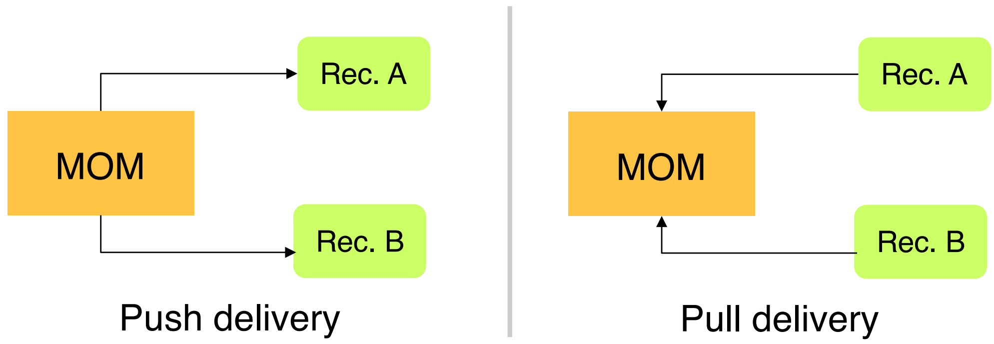
Timeout-based delivery

- *How can MOM ensure that messages are only deleted from a message queue if they have been received successfully at least once?*
  – Message is not deleted immediately from queue, but marked as being 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



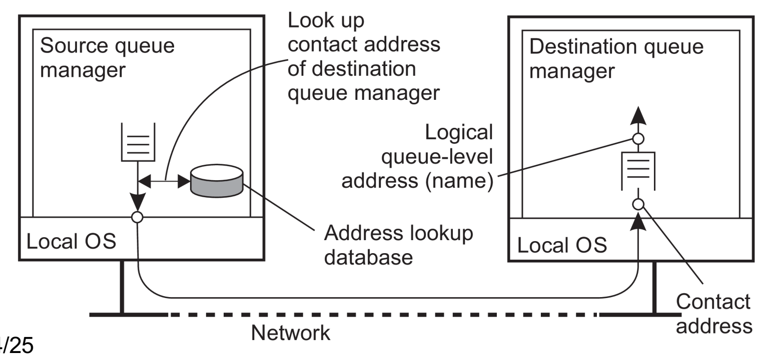
Delivery model

- How messages are retrieved by receivers (i.e., subscribers or consumers)
- Options: push vs. pull delivery
- Push: receiver is notified by MOM when a message is available
- Pull: receiver polls MOM for new messages



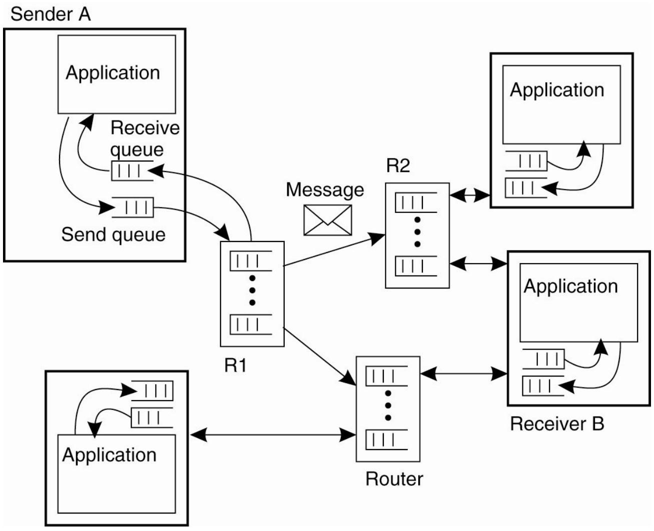
Message routing: general model

- Queues are managed by queue managers (QMs)
  - An application can put messages only into a local queue
  - Getting a message is possible by extracting it from a local queue only
- QMs need to route messages
  - Work as message-queuing "relays" that interact with distributed applications and each other
  - Form an overlay network
  - There can also be special QMs that operate only as routers



Message routing: overlay network

- Overlay network to route messages
  - By using routing tables
  - Routing tables are stored and managed by QMs
- Overlay network needs to be maintained over time
  - Routing tables are often set up and managed manually: easier but …
  - Dynamic overlay networks require to manage at runtime mapping between queue name and its location

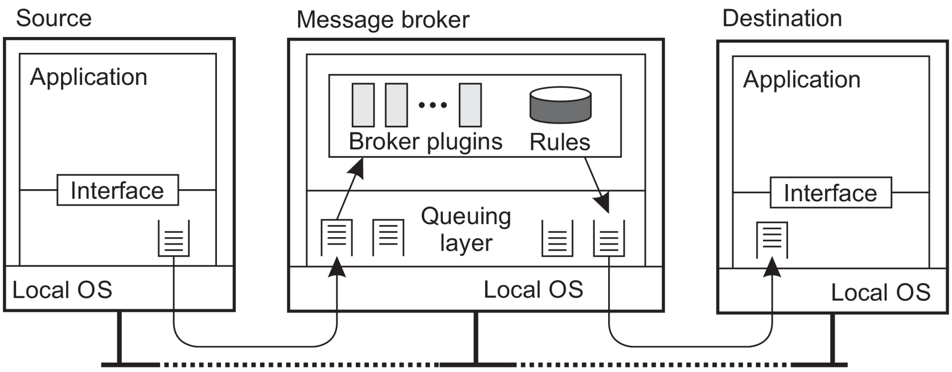


Message transformation: message broker

- New/existing apps that need to be integrated into a single, coherent system rarely use common data format
- How to handle data heterogeneity?
- Let's now focus on ***message broker***
  - Usually takes care of application heterogeneity in MOM

Message broker: general architecture

- Message broker handles application heterogeneity
  - Converts incoming messages to target format providing access transparency
  - Often acts as application gateway
  - Manages repository of conversion rules and programs to transform message types
  - May provide content-based routing capabilities
  - Must be scalable and reliable: distributed implementation



MOM frameworks

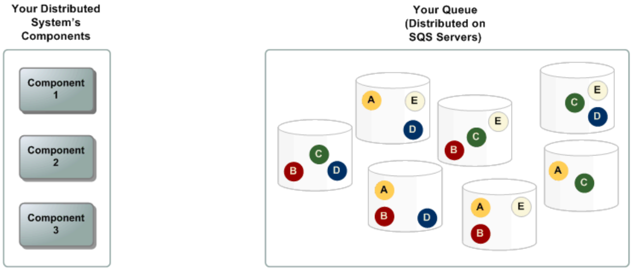
- Main MOM systems and libraries
  - Apache ActiveMQ https://activemq.apache.org
  - **Apache Kafka**
  - Apache Pulsar https://pulsar.apache.org
  - IBM MQ https://www.ibm.com/products/mq
  - NATS https://nats.io
  - **RabbitMQ**
  - ZeroMQ https://zeromq.org
- Distinction between queue message and pub/sub patterns is often lacking
  - Some frameworks support both (e.g., Kafka, NATS)
  - Others not (e.g., pub/sub in Redis https://redis.io/topics/pubsub)

MOM frameworks

- Also as Cloud services
  - **Amazon Simple Queue Service (SQS)**
  - Amazon Simple Notification Service (SNS)
  - CloudAMQP: RabbitMQ as a Service
  - Google Cloud Pub/Sub
  - Microsoft Azure Service Bus

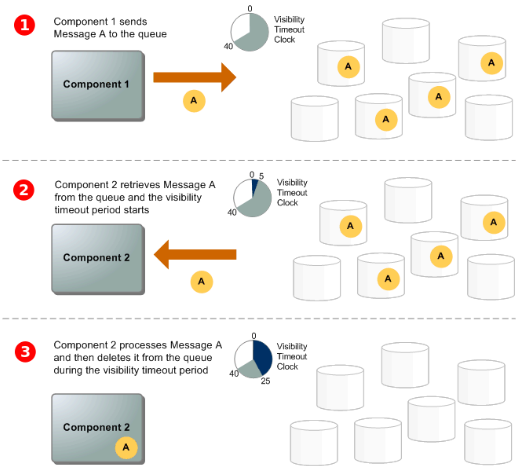
Amazon Simple Queue Service (SQS)

- Reliable, scalable Cloud-based message queue service
  - Goal: decouple application components, which can run independently and asynchronously and be developed with different technologies and languages
- Architecture
  - Message queues fully managed by AWS
  - Message queues distributed on multiple SQS servers
  - SQS servers replicated within a region: message copies stored on multiple servers for high availability
  - Pull delivery model: consumers poll message from queue



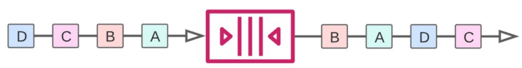
Amazon SQS: Features

- Consumer must delete message from queue
  - A queue is a temporary holding location
  - Configurable message retention period (max 14 days)
- SQS provides timeout-based delivery
  - Received message remains in queue but is locked during consumer processing (visibility timeout)
  - If processing fails, lock expires and message is available again



Amazon SQS: Features

- Consumers use polling to receive messages from queue
  - *Short* polling: SQS queries only a subset of servers
  - *Long* polling: SQS queries all servers
- SQS queue type can be standard or FIFO
- Standard queue (default)
  - Best-effort ordering, thus occasionally out-of-order delivery might occur
  - Duplicates can be received



Standard Queue

- FIFO queue
  - In-order delivery, i.e., messages are received and processed in the same order in which they were transmitted
  - Avoids duplicates
  - ✗ Reduced throughput

FIFO Queue

Amazon SQS: API

- `CreateQueue`, `ListQueues`, `DeleteQueue`
  - Create, list, delete queues
- `SendMessage`
  - Add message to queue (message size up to 256 KB)
  - How to send message payload larger than 256 KB?
    - Store payload on S3 and send a reference to it inside message
- `ReceiveMessage`
  - Retrieve one or more messages from queue
  - Can't specify which messages to retrieve, only maximum number of messages (up to 10)
- `DeleteMessage`
  - Remove specified message from queue

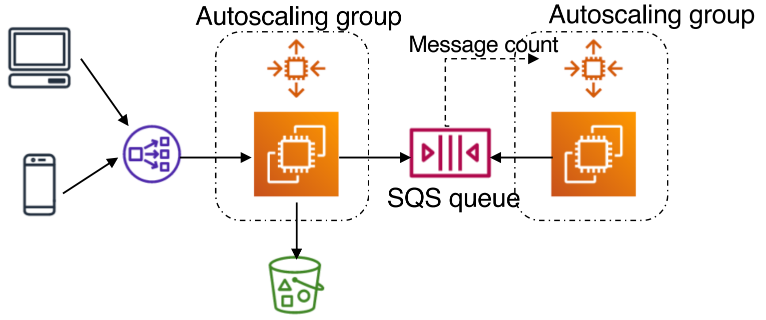
https://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/Welcome.html

Amazon SQS: API

- `ChangeMessageVisibility`
  - Change visibility timeout of the specified message in a queue (when received, message remains in the queue upon it is explicitly deleted by consumer)
  - Default visibility timeout is 30 sec.
- `SetQueueAttributes`, `GetQueueAttributes`
  - Control queue settings, get information about a queue

Amazon SQS: example

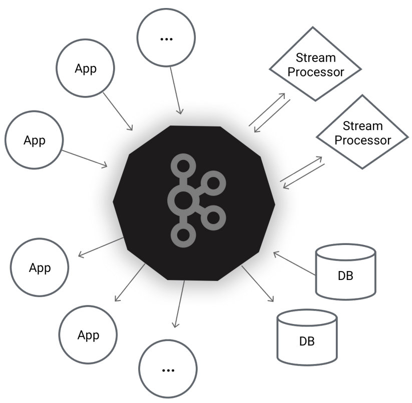
- Cloud app for photo processing service
  - Let's use SQS to decouple app front-end and back-end, load balancing and fault tolerance
  - App front-end sends to queue a message with S3 link to image
  - Pool of EC2 instances takes requests from queue and resizes images
    - In case of failure during processing, message is again visible in queue
    - Back-end EC2 instances can be scaled horizontally according to number of queued messages



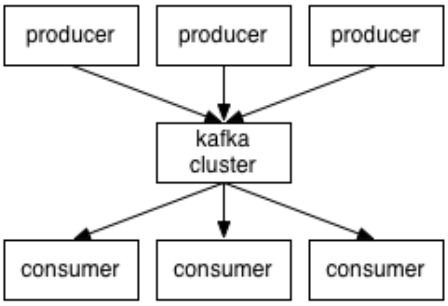
Apache Kafka

- Open-source, distributed even streaming platform
  - To publish and subscribe to streams of events
  - To store streams of events durably and reliably
  - To process streams of events

  https://kafka.apache.org/
- Started by LinkedIn in 2010
- Used at scale by tech giants (LinkedIn, Netflix, Uber, …)
- Written mainly in Scala
- Horizontally scalable
- Fault-tolerant
- High throughput ingestion
  - Billions of events



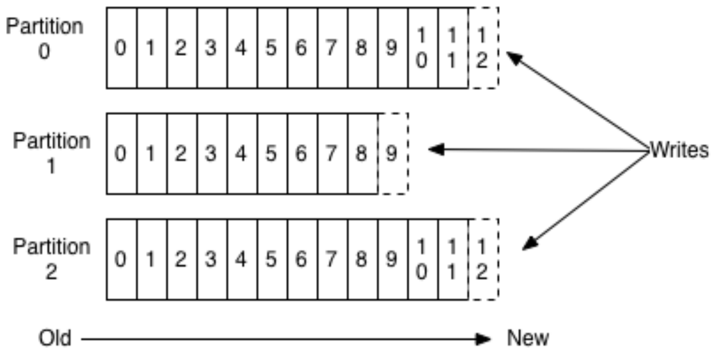
Kafka at a glance



- Kafka stores streams of *events* (aka *messages*, *records*) in categories called *topics*
- Kafka cluster: composed by servers known as *brokers*, that 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

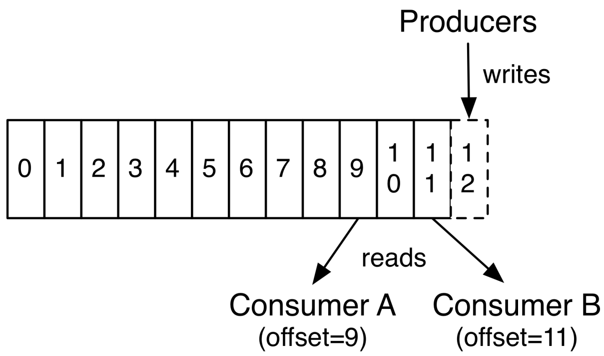
Kafka: topic and partitions

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



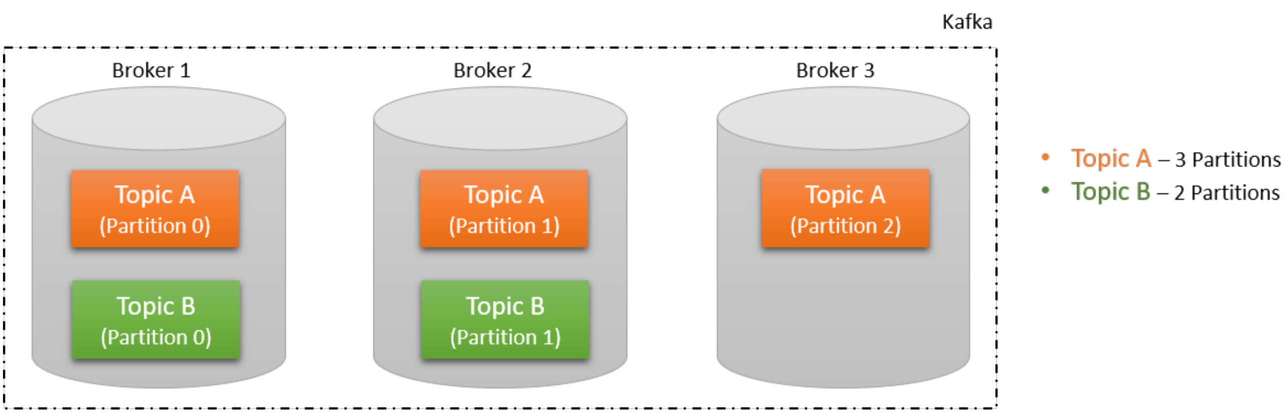
Kafka: partitions

- Producers publish events to topic partition
- Consumers read events from topic
- Each partition is a *numbered*, *ordered*, *immutable* sequence of records that is *appended to*
  - Records written to partition are immutable
  - Like a commit log
- Each record is associated with a monotonically increasing sequence number, called offset



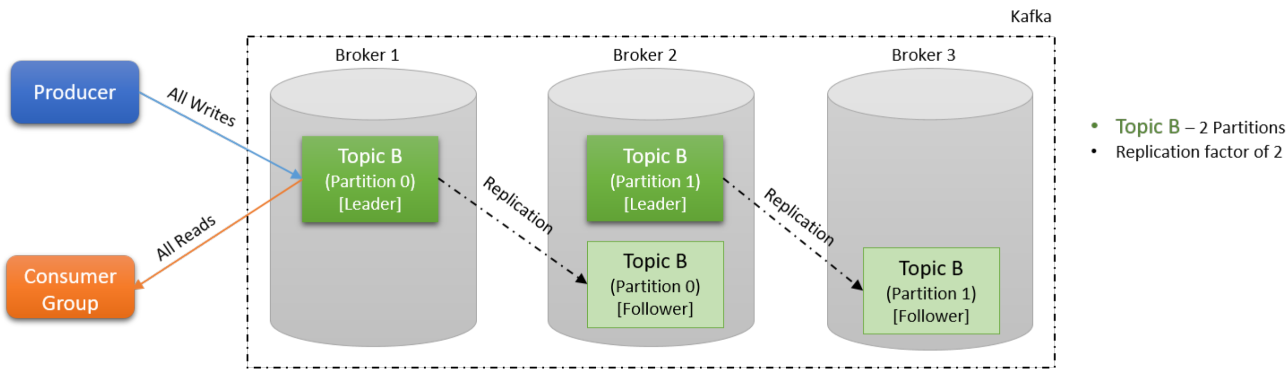
Kafka: partitions and design choices

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



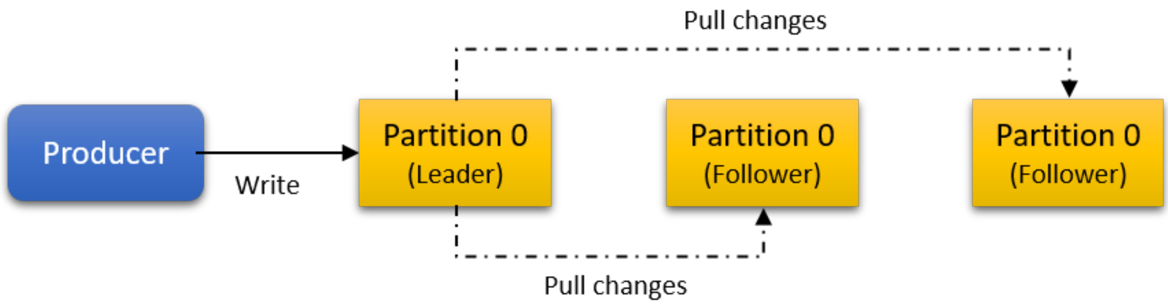
Kafka: partitions and design choices

- To improve fault tolerance: each topic partition can be *replicated* across brokers
  - Each partition has one leader and 0 or more followers
    - followers > 0 in case of replication
  - *replication-factor* = total number of replicas including leader
  - *replication-factor* = *N* → up to *N*-1 brokers can fail before losing stored events



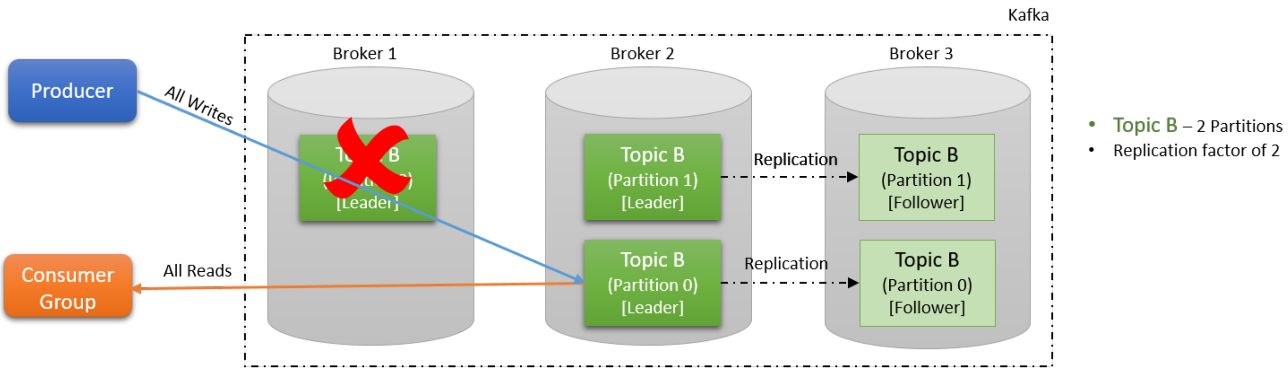
Kafka: partitions and design choices

- To simplify data consistency management: only leader handles read and write requests
  - Producers read from leader, consumers write to leader
  - Followers replicate leader and act as backups
  - Followers can be *in-sync* (i.e., fully updated replica) with leader or *out-of-sync*



Kafka: partitions and design choices

- To share responsibility and balance load: each broker is leader for some of its partitions and follower for others
  - Brokers can rely on Apache Zookeeper or KRaft for coordination, including leader election

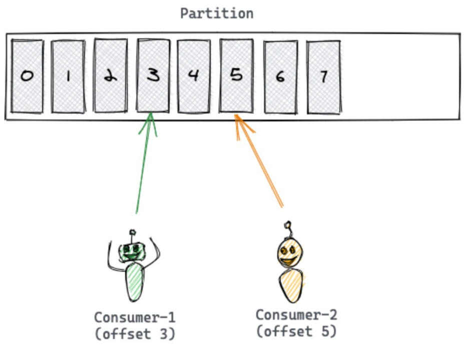


Kafka: producers

- Producers = data sources
- Publish events to topics of their choice
  - Producers send events directly (i.e., without any routing tier) to the broker which is leader for the partition
- Responsible for choosing which event to assign to which partition within the topic: how?
  - *Key-based* partitioning, i.e., producer uses a partition key within event to write it to a given partition
    - Partition is chosen based on key hash
    - E.g., if key=user_id, then all events of a given user are sent to same partition
  - Round-robin (default, if key is not specified)
- Multiple producers can write to same partition

Kafka: consumers

- Kafka uses pull delivery model for consumers
  https://kafka.apache.org/documentation.html#design_pull
- Consumer uses offset to keep track of which events it has already consumed
- Same partition can be read by multiple consumers, each reading at different offsets



Kafka: consumers

- Why pull?
- Push
  – Broker actively pushes events to consumers
  ✗ Broker has to deal with different consumers with diverse needs and capabilities and control transmission rate
  ✗ Broker has to decide push timing: whether to send a message immediately or accumulate more data and then send
- Pull
  – Consumers are in charge of retrieving events from broker
  – Consumers have to maintain offset to identify next event to read
  ✓ More scalable (less burden on brokers) and flexible
  ✗ If broker has no events, consumers may end up busy waiting for events to arrive

Kafka: consumers

- How can consumer read in a fault-tolerant way?
  - Once consumer reads events, it stores its committed offset in a special Kafka topic called `__consumer_offsets`
  - After recovering from crash, consumer can replay events using committed offset
  - By default, `auto-commit` is enabled

Kafka: brokers

- Kafka brokers store messages reliably on disk
- Differently from other queue message and pub/sub systems, Kafka does not delete messages after delivery, but retains messages
- Issue: need to free up disk space, how?
  - 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

Hands-on Kafka

- Preliminary steps:
  - Download and install Kafka
    https://kafka.apache.org/downloads
    - Configure Kafka properties in `server.properties` (e.g., `listeners` and `advertised.listeners`)
  - Start Kafka environment
    Kafka can be started using KRaft or ZooKeeper, let's use Kafka with ZooKeeper (included with Kafka download)
    - Start ZooKeeper (default port: 2181)

    ```
    $ zookeeper-server-start zookeeper.properties
    ```

    Alternatively `$ zKserver start`
    - Start Kafka broker (default port: 9092)

    ```
    $ kafka-server-start server.properties
    ```


Hands-on Kafka

- Let's use Kafka CLI tools to create a topic, publish and consume some events to/from topic and delete it
- Create a topic named test with 1 partition and non-replicated
  - `bootstrap_server`: specify one Kafka broker

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

- Write some events into topic

```
$ kafka-console-producer --bootstrap-server localhost:9092
  --topic test
> first message
> another message
```

- Read events from beginning of topic

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

Hands-on Kafka

- Read events from a given offset (e.g., 2) and a specific topic partition

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

- List available topics

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

- Delete topic

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

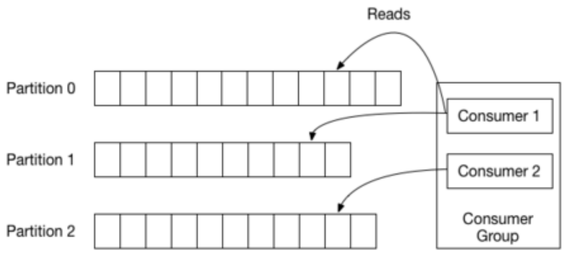
- Stop Kafka and Zookeeper

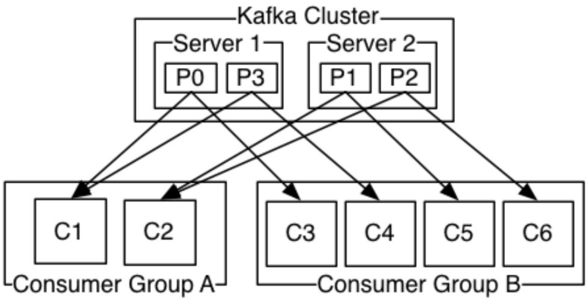
```
$ kafka-server-stop
$ zookeeper-server-stop
```

Alternatively `$ zKserver stop`

Kafka: consumer group

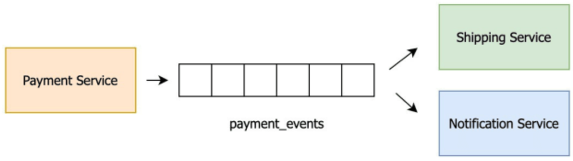
- Consumer Group: set of consumers which cooperate to consume data from some topic and share a group ID
  - A Consumer Group maps to a *logical* subscriber
  - Topic partitions are divided among consumers in the group for load balancing and can be reassigned in case of consumer join/leave
  - Every event is delivered to *only one* consumer in group
  - Every group maintains its offset per topic partition



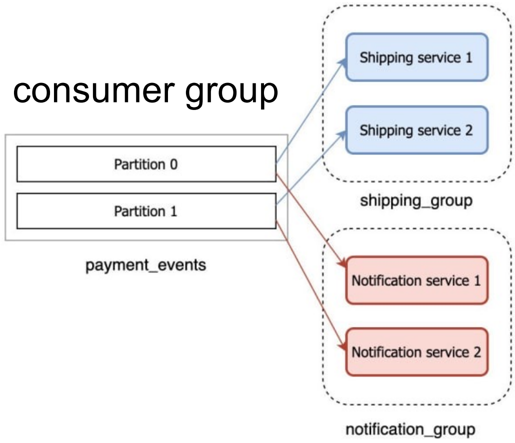


Kafka: consumer group

- How can many consumers read the same events from the topic?
  - Use different group IDs
- Example: microservices communicate using Kafka



- How to scale?
  - Each microservice has its own consumer group

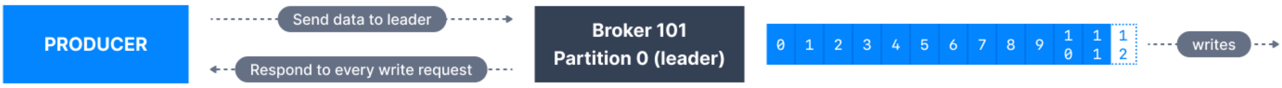


Kafka: ordering guarantees

- Events published by producer to topic partition will be appended in the order they are sent
- Consumer reads events in the order they are stored in the partition
- Strong guarantee about ordering *only within a partition*
  - Total order over events within partition, i.e., *per-partition ordering*
  - Kafka does not preserve event order among different topic partitions
- Per-partition ordering plus ability to partition events among partitions by key is sufficient for most applications

Kafka: delivery semantics

- **At-least-once** (default): no event loss, but events may be duplicated and out-of-order (wrt producer)
  – Producer: wait for ack only from partition leader; if none, retry
  – How? Set `acks=1`
    - `acks` is the the number of brokers who need to acknowledge receiving the event before it is considered a successful write
  – Consumer: commit offset after processing event

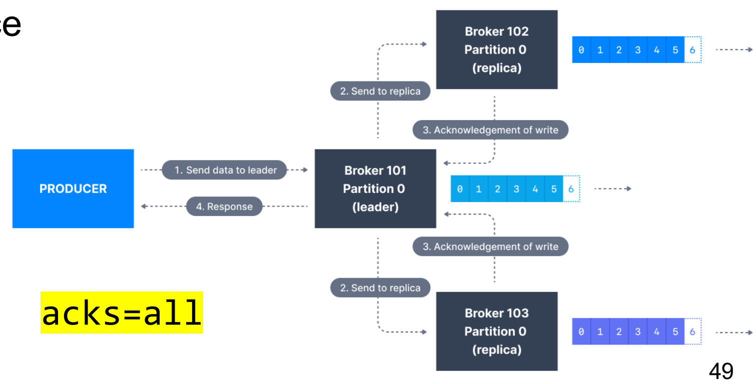


`acks=1`

See https://kafka.apache.org/documentation/#semantics

Kafka: delivery semantics

- **Exactly-once**: no event loss, no duplicates and partition-level ordering, but higher latency and lower throughput
  – Producer: wait for ack from all in-sync partition replicas
  – How? Set `acks=all` on producer
  – Requires also producer ID and event sequence number in each event sent from producer (aka idempotent producer) to detect and avoid duplicates and maintain log order
  – Requires also committed offsets and in-sync replicas
  – Not fully exactly-once



`acks=all`

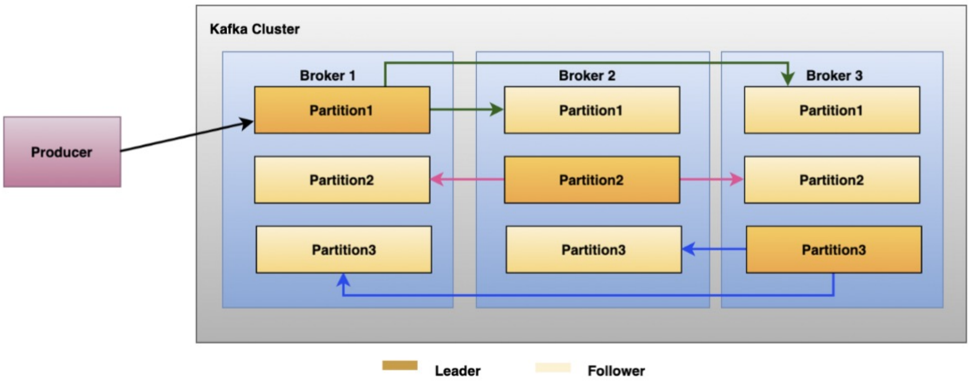
Kafka: delivery semantics

- User can also implement **at-most-once**: messages may be lost but are never re-delivered
  - Producer: disable retries (i.e., `acks=0`)
  - Consumer: commit offset before processing message

- Take-away message: choose delivery semantics that makes sense for your application context

Kafka: fault tolerance

- Kafka replicates topic partitions for fault tolerance
  - Leader coordinates to update followers when new events arrive
  - Set of in-sync replicas known as ISR



- If leader crashes, a follower can be elected as new leader by Zookeeper or KRaft

Kafka: fault tolerance

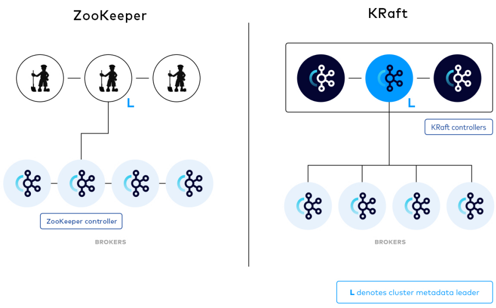
- Kafka makes an event available for consumption only after all replicas in ISR for that partition have applied it to their log
  - Events may not be immediately available for consumption: tradeoff between consistency and availability
- Producers can either waiting for event to be committed or not (by setting `acks`)
  - Tradeoff between latency and durability

Kafka and ZooKeeper

- Zookeeper: hierarchical, distributed key-value store https://zookeeper.apache.org/
  - Coordination service for distributed systems, which provides facilities for locking, leader election, monitoring
  - Maintains a namespace, organized as tree
  - Simple operations on tree: create and delete nodes, read and update data contained in a node
  - Used within many open-source distributed systems
- Kafka uses ZooKeeper for metadata management
  - List of brokers
  - Configuration for topics and permissions
  - Leader election: to determine partition leader
  - Zookeeper allows Kafka to know about changes (e.g., new topic, deleted topic, broker crash, broker restart)

From ZooKeeper to KRaft

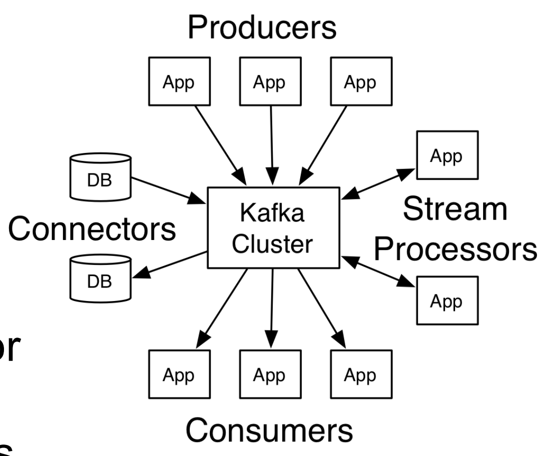
- Zookeeper cons
  - ✗ Different system for metadata management and consensus
  - ✗ Can become bottleneck as Kafka cluster grows
- Apache Kafka Raft (KRaft) in newer releases
  - – Kafka cluster metadata stored in Kafka cluster itself
  - ✓ Simpler architecture
  - ✓ Faster and more scalable metadata update operations
  - – Metadata replicated to all brokers, making failover from failure faster
  - – Consensus protocol based on Raft



Kafka: APIs

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

- 5 APIs (Java and Scala only)
- Producer API: publish data to Kafka topics
- Consumer API: read data from Kafka topics
- Kafka Connect API: build and run reusable connectors (producers or consumers) that connect Kafka topics to apps or external systems (source or sink)



  - – Many pre-built connectors: AWS S3, RabbitMQ, MySQL, Postgres, AWS Lambda, …

Kafka: APIs

- Kafka Streams API: transform streams of data from input topics to output topics
  - Kafka is an event streaming platform (not only pub-sub)
- Admin API: manage and inspect topics, brokers, and other Kafka objects

Kafka: client library

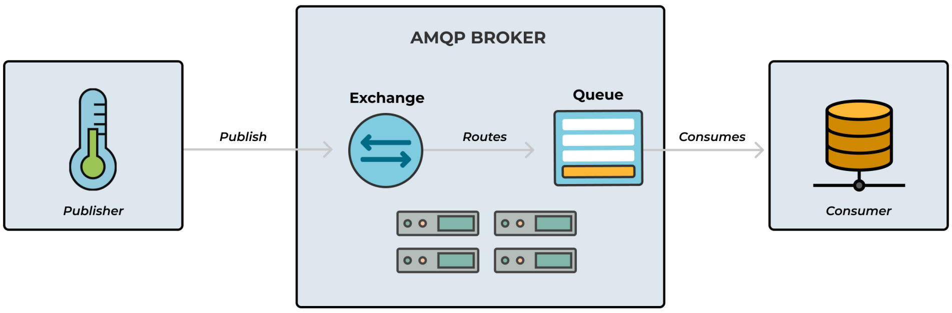
- Kafka officially provides only SDK for Java
- For other languages, implementations of client library provided by community, including
  - Go
    https://github.com/confluentinc/confluent-kafka-go
    https://github.com/segmentio/kafka-go
  - Python
    https://github.com/confluentinc/confluent-kafka-python

Messaging protocols

- Application-layer open standard protocols to interact with MOM
  - **AMQP** (Advanced Message Queueing Protocol)
    - Binary protocol
  - MQTT (Message Queue Telemetry Transport) https://mqtt.org
    - Binary, ligthweight protocol
  - STOMP (Simple Text Oriented Messaging Protocol) https://stomp.github.io/
    - Simple, text-based protocol
- Goals:
  - Platform- and vendor-agnostic
  - Provide interoperability between different MOMs

Messaging protocols and IoT

- Widely used in Internet of Things (IoT)
  - Use messaging protocol to send data from sensors to services that process data



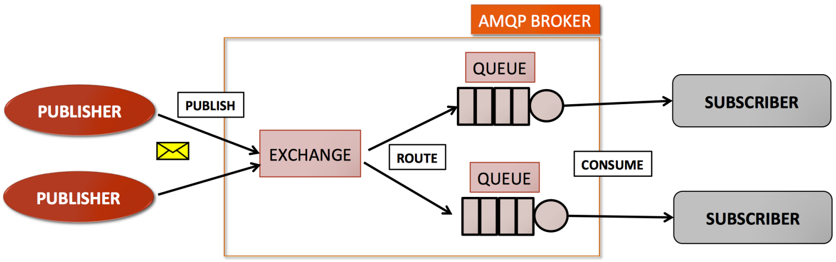
- Why? Exploit MOM advantages for IoT
  - Decoupling
  - Resiliency: temporary message storage provided by MOM
  - Traffic spikes handling: data persisted in MOM and processed eventually

AMQP: characteristics

- Open standard protocol for MOM, 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
- Implementations
  - Apache ActiveMQ, RabbitMQ, Apache Qpid, Azure Event Hubs, Pika (Python implementation), …

AMQP: model

- AMQP architecture involves 3 main actors:
  - Publishers, subscribers, and brokers



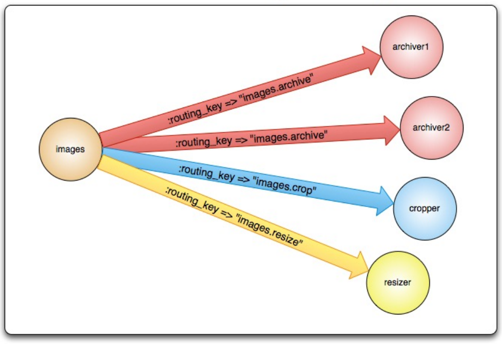
- AMQP entities (within broker): queues, exchanges and bindings
  - Messages are published to *exchanges* (like post offices or mailboxes)
  - Exchanges distribute message copies to *queues* using rules called *bindings*
  - AMQP brokers either push messages to consumers subscribed to queues, or consumers pull messages from queues on demand

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

AMQP: routing

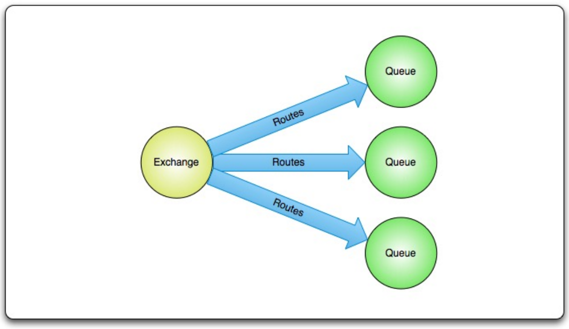
- Different types of exchanges that route messages differently
  - Direct exchange: delivers messages to queues based on message routing key

Direct exchange routing



  - Fanout exchange: delivers messages to all queues that are bound to it

Fanout exchange routing

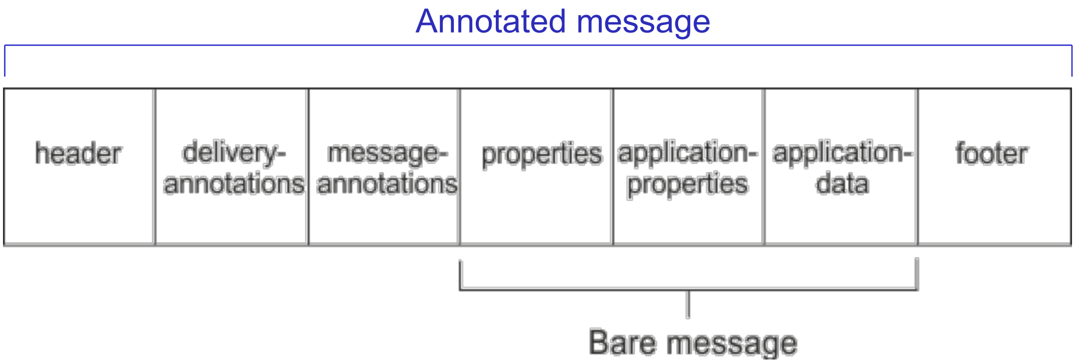


AMQP: routing

- Different types of exchanges that route messages differently
  - Topic exchange: delivers messages to one or many queues based on topic matching
    - Often used to implement publish/subscribe pattern variations
    - Commonly used for multicast routing of messages
    - Example use: distributing data relevant to specific geographic location (e.g., points of sale)
  - Headers exchange: delivers messages based on multiple attributes expressed as headers
    - To route on multiple attributes that are more easily expressed as message headers than routing key

AMQP: messages

- AMQP defines two types of messages:
  - Bare messages, supplied by sender
  - Annotated messages, seen at receiver and added by intermediaries during transit
- Message header conveys delivery parameters
  - Including durability requirements, priority, time to live



RabbitMQ

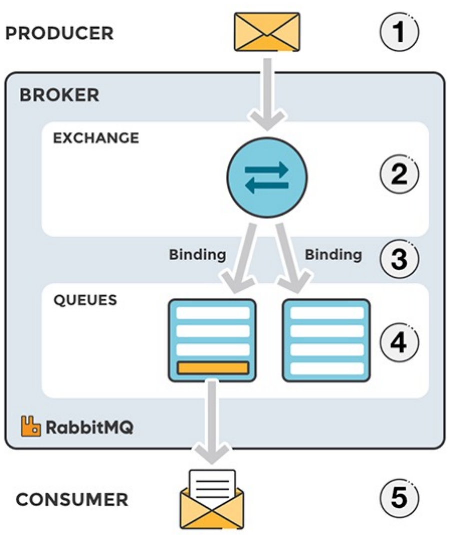
- Open-source message broker https://www.rabbitmq.com/

- Uses push delivery model
- Offers FIFO ordering guarantee at queue level
- Supports multiple messaging protocols
  - AMQP, STOMP and MQTT
- Runs on many operating systems and cloud environments
- Provides a wide range of development tools for popular languages (Java, Go, Python, …)

RabbitMQ: architecture

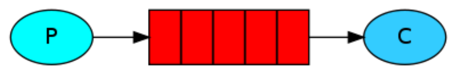
- Messages are not published directly to a queue
- Producer sends messages to an exchange, which routes messages to different queues with the help of bindings and routing keys
  - Binding: link between a queue and an exchange

Message flow in RabbitMQ

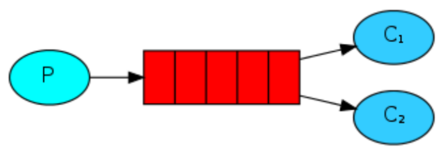


- RabbitMQ broker can be distributed, e.g., forming a cluster https://www.rabbitmq.com/distributed.html
  - Supports quorum queue: durable, replicated FIFO queue based on Raft consensus algorithm

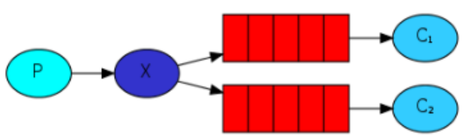
RabbitMQ: use cases



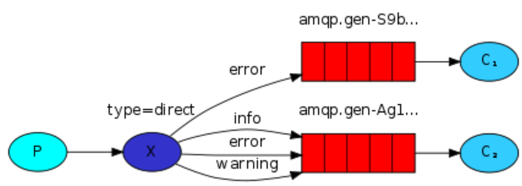
1. Store and forward messages sent by a producer and received by a consumer (message queue pattern)



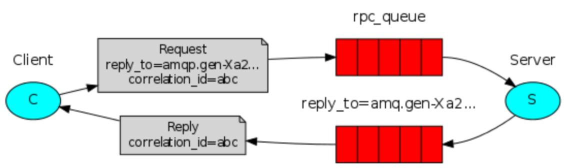
2. Distribute tasks among multiple workers (work queue pattern)



3. Deliver messages to many consumers (pub/sub pattern) using a *message exchange*



4. Receive messages selectively: producer sends messages to an *exchange,* that selects the queue



5. Run a function on a remote node and wait for result (RPC pattern)

https://www.rabbitmq.com/tutorials

RabbitMQ and Go

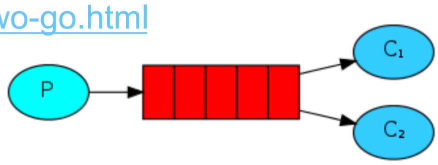
- Let's use RabbitMQ, Go and AMQP (messaging protocol) for:

Ex. 1: Message queue pattern

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

Ex. 2: Work queue pattern

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 on default port https://www.rabbitmq.com/download.html

   `$ rabbitmq-server`

   – RabbitMQ CLI tool: `rabbitmqctl`

   ```
   $ rabbitmqctl status
   $ rabbitmqctl shutdown
   ```

   Some useful commands for `rabbitmqctl`

   ```
   list_channels
   list_consumers
   list_queues
   stop_app
   reset
   ```

   – Also web UI for management and monitoring

2. Install Go AMQP client library

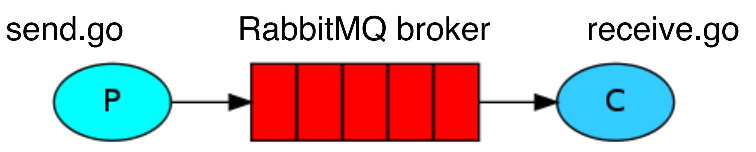
   `$ go get github.com/rabbitmq/amqp091-go`

   See https://pkg.go.dev/github.com/rabbitmq/amqp091-go for details on `ampq`

RabbitMQ and Go: example 1

1. Message queue pattern

- Run single producer/single consumer, multiple producers/multiple consumers
- Note that:
  - Message is delivered to only one consumer
  - Delivery is push-based



RabbitMQ and Go: example 2

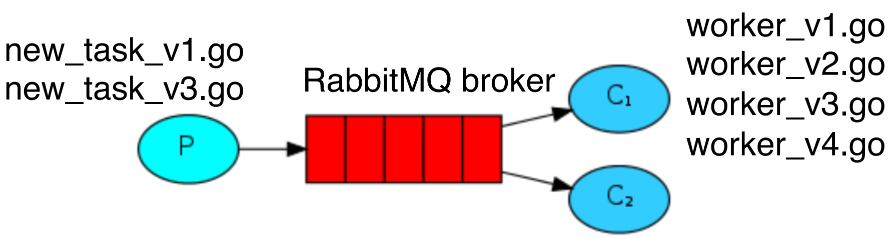
2. Work queue pattern

- Version 1 (new_task_v1.go and worker_v1.go):
  - Use multiple consumers and see how queue allows us to distribute tasks among consumers in *round-robin* fashion
  - If consumer crashes after RabbitMQ delivers message but before completing task, message is lost (i.e., cannot be delivered to another consumer)

    `auto-ack=true`: message is considered to be successfully delivered immediately after it is sent (“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 received, processed and that RabbitMQ can safely discard it
  - Let’s shutdown and restart RabbitMQ: what happens to pending messages?
  - Which delivery semantics with explicit acks?

RabbitMQ and Go: example 2

2. Work queue pattern
   - Version 3 (new_task_v3.go and worker_v3.go):
     - Use durable queue so it is persisted to disk and survives RabbitMQ crash and restart
     - New queue with `durable=true` in `QueueDeclare`
   - Version 4 (new_task_v3.go and worker_v4.go):
     - Improve task distribution among consumers by looking at number of unacknowledged messages for each consumer, so to not dispatch a new message to a consumer until it has processed and acknowledged the previous one
     - Use channel prefetch setting (`Qos`)



Multicast communication

- **Multicast communication**: group communication pattern in which 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, in which data is sent to *all* receivers
- Cannot be implemented as unicast replication (source sends as many copies as receivers number): lack of scalability
  - Solution: replicate only when needed

Types of multicast

- How to realize multicast?
  - **Network**-level multicast (IP-level)
    - Packet replication and routing managed by network routers: IP Multicast
    - ✗ Limited usage
  - **Applicatio**n-level multicast
    - Replication and routing managed by hosts

Application-level multicast

- Basic idea:
  - Organize nodes into overlay network
  - Use overlay network to disseminate data
  - Structured or unstructured
- Structured application-level multicast
  - Explicit communication paths
  - How to build 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

Unstructured application-level multicast

- How to realize unstructured application-level multicast?
  - ✓ Flooding
    - Node *P* sends multicast message *m* to all its neighbors
    - In its turn, each neighbor will forward multicast message to all its neighbors (except to *P)* if it had not seen *m* before
  - ✓ Random walk
    - Node *P* sends multicast message *m* to a randomly chosen neighbor
    - In its turn, the neighbor will forward multicast message to a randomly chosen neighbor
  - 👉 **Gossiping**

Gossip-based protocols

- Gossip-based protocols (or algorithms) are probabilistic (aka *epidemic* algorithms)
  - Gossiping effect: information can spread within a group just as it would be in real life
  - 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 node that receives it will send a copy to another subset, also chosen at random, and so on

Origin of gossip-based protocols

- Gossiping protocols proposed in 1987 by Demers et al. in a work on data consistency in replicated databases composed of hundreds of servers
  - Basic idea: assume there are no write conflicts (i.e., independent updates)
  - Update operations are initially performed at one replica server
  - A replica passes its updated state to only a few 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 quickly spread information throughout the S3 system. This allows Amazon S3 to quickly route around failed or unreachable servers, among other things"
- Amazon's Dynamo uses gossiping for failure detection of nodes
- BitTorrent uses a gossip-based information exchange
- Cassandra uses gossiping for group membership and failure detection of nodes
- Gossip dissemination pattern
  https://martinfowler.com/articles/patterns-of-distributed-systems/gossip-dissemination.html

Strategies to spread updates

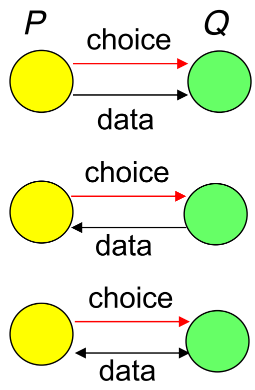
- Let's consider the two principle operations

1. Anti-entropy: a node regularly picks another node randomly and exchanges updates (i.e., state differences), aiming to have identical states at both afterwards

2. Rumor spreading: periodically a node which has new or updated information (i.e., has been contaminated) selects F ($F >= 1$) peers to send updates to (contaminating them); a node that has received an update can stop distributing it

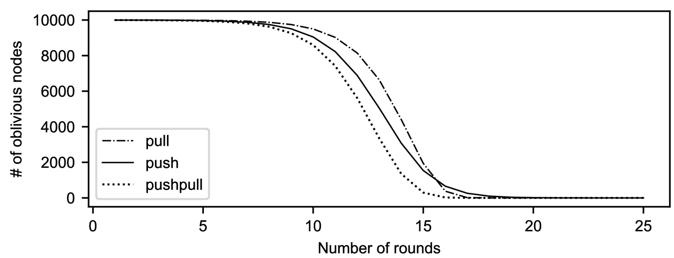
Anti-entropy

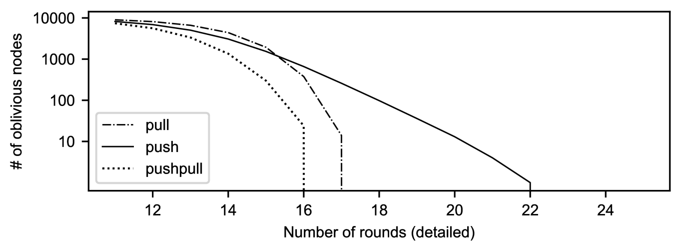
- Goal: increase node state similarity, thus decreasing “disorder” (reason for name!)
- Node *P* selects node *Q* randomly: how does P update *Q*?
- 3 different update strategies:
  - **push**: *P* only pushes its own updates to *Q*
  - **pull**: *P* only pulls in new updates from *Q*
  - **push-pull**: *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 takes the initiative to start an exchange

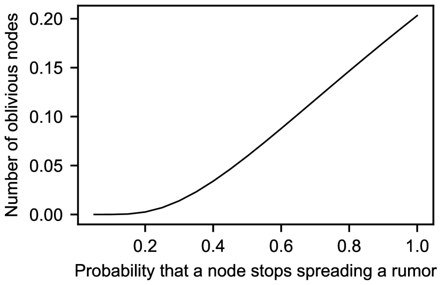




Rumor spreading

- Node P, having an update to report, contacts randomly chosen node *Q* and forwards update message to it
- If *Q* was already updated, *P* may lose interest in spreading update any further and with probability p_{stop} stops contacting other nodes
- Fraction *s* of oblivious nodes (that have not 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

General schema of gossiping protocol

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

Active thread (node P):		Passive thread (node Q):
(1) **selectPeer**(&Q);		(1)
(2) **selectToSend**(&bufs);		(2)
(3) sendTo(Q, bufs);	----->	(3) receiveFromAny(&P, &bufr);
(4)		(4) **selectToSend**(&bufs);
(5) receiveFrom(Q, &bufr);	<-----	(5) sendTo(P, bufs);
(6) **selectToKeep**(cache, bufr);		(6) **selectToKeep**(cache, bufr);
(7) processData(cache);		(7) processData(cache)

selectPeer: randomly select a neighbor
selectToSend: select some entries from local cache
selectToKeep: select which received entries to store into local cache; remove repeated entries

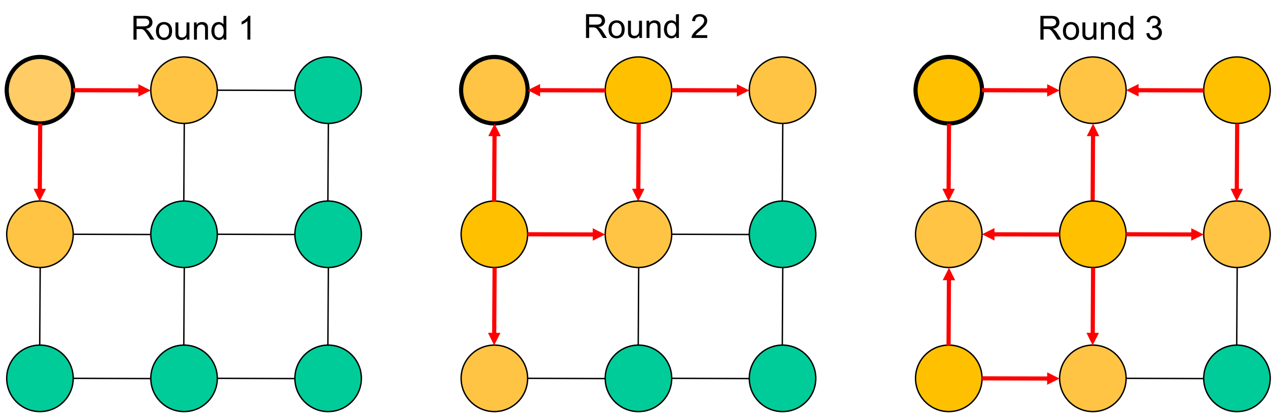
Kermarrec and van Steen, Gossiping in distributed systems, *SIGOPS Oper. Syst. Rev.*, 2007 https://www.distributed-systems.net/my-data/papers/2007.osr.pdf

Framework of gossiping protocols

- Simple? Not quite getting into the details…
- Some crucial aspects
  - Peer selection
    - E.g., *Q* can be uniformly chosen from set of currently available (i.e., alive) nodes
  - Data exchanged
    - Exchange 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 more efficient than flooding
- Flooding-based information dissemination
  - Each node that receives message forwards it to its neighbors (let's consider *all* neighbors, including sender)
  - Message is eventually discarded when TTL=0



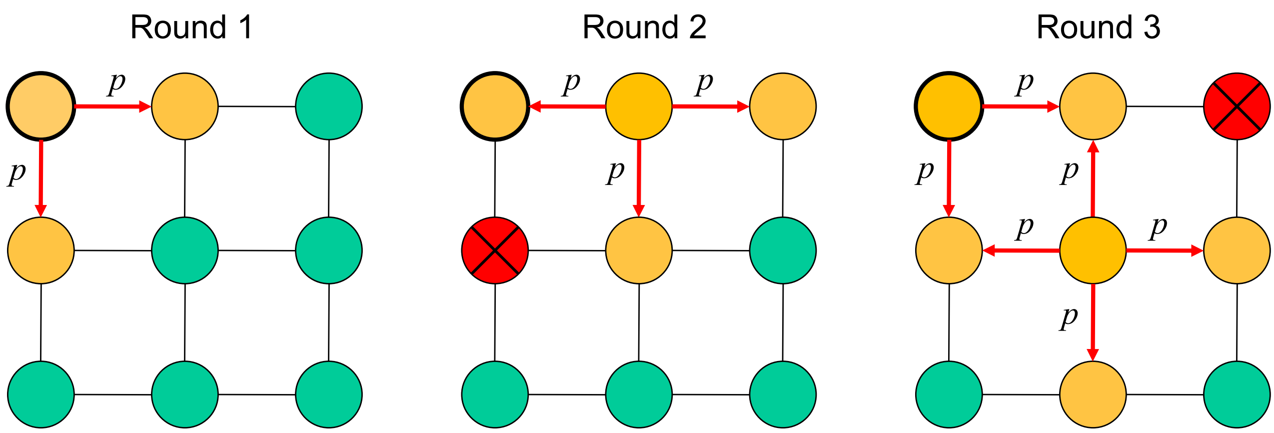
Sent messages: 18
Reached nodes: 8 out of 9

Gossiping vs flooding: example

- Let's use rumor spreading
  - Message is sent to 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 features
  - Probabilistic
  - Takes a localized decision but results in a global state
  - Lightweight
  - Fault-tolerant
- Flooding has some advantages
  - Universal coverage and minimal state information
  - … but it floods the networks with redundant messages
- Gossiping goals
  - Reduce the number of redundant transmissions that occur with flooding while trying to retain its advantages
  - … but due to its probabilistic nature, gossiping cannot guarantee that all the peers are reached and it requires more time to complete than flooding

Other application domains of gossiping

- Besides information dissemination…
- Peer sampling
  - How to provide every node with a list of peers to exchange information with
- Resource management, including monitoring, in large-scale distributed systems
  - E.g., failure detection
- Distributed computations to *aggregate* data in very large distributed systems (e.g., sensor networks)
  - Computation of aggregates e.g., sum, average, maximum and minimum values
  - E.g., to compute average value
    - Let $v_{0,i}$ and $v_{0,j}$ be the values at time t=0 stored by nodes i and j
    - Upon gossip, i and j exchange their local value v_i and v_j and adjust it to

      $v_{1,i}, v_{1,j} \leftarrow (v_{0,i} + v_{0,j})/2$

Gossiping case studies

1. Blind counter rumor mongering: an example of gossiping protocol
2. Bimodal multicast: multicast protocol that exploits gossiping to achieve reliability

Blind counter rumor mongering

- Why such name?
  - *Rumor mongering* (def: "the act of spreading rumors", also known as gossip): a node with "hot rumor" will periodically infect other nodes
  - *Blind*: loses interest regardless of message recipient (*why*)
  - *Counter*: loses interest after some 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* uniformly randomly chosen neighbors that *p* knows have not yet seen *m*

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

- Performance (*B*=*F*=2) with respect to flooding
  - Lower number of messages (~50%)
  - Not complete coverage (~90%)
  - Slower (~2x)

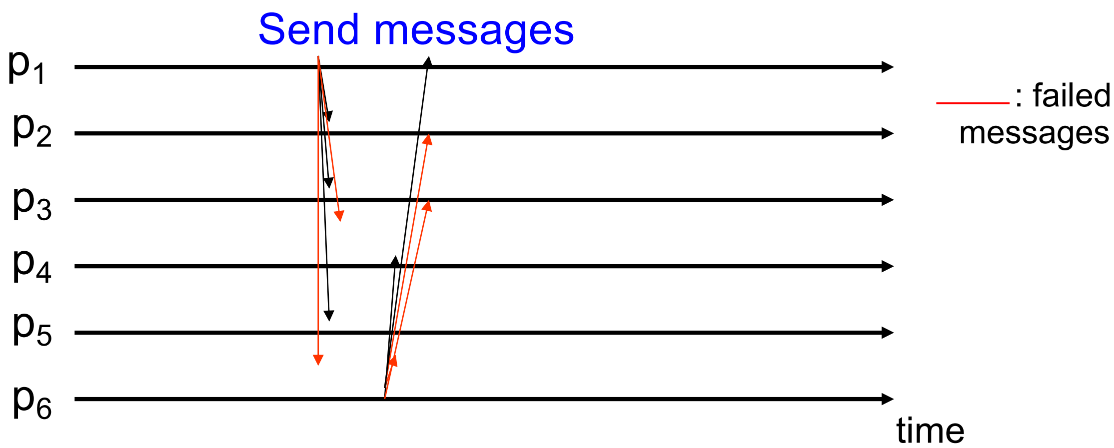
Bimodal multicast

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

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

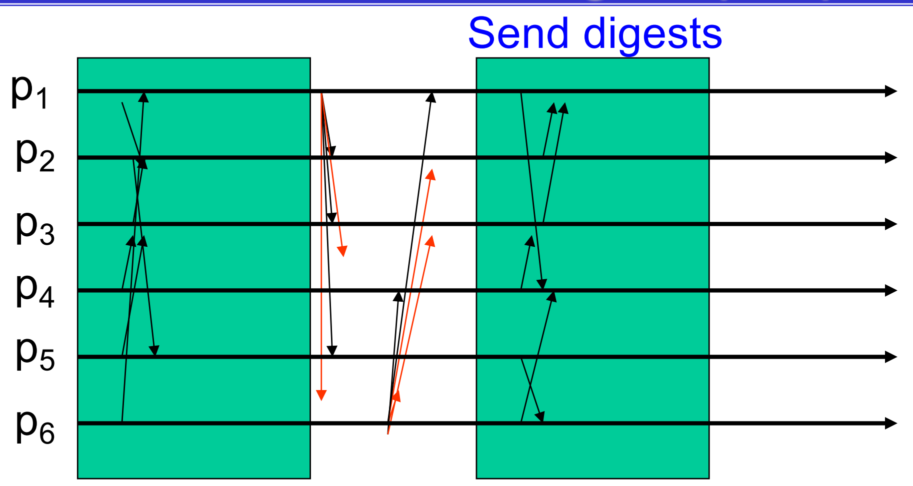
Birman et al., Bimodal multicast, *ACM Trans. Comput. Syst.,* 1999

Bimodal multicast: message distribution



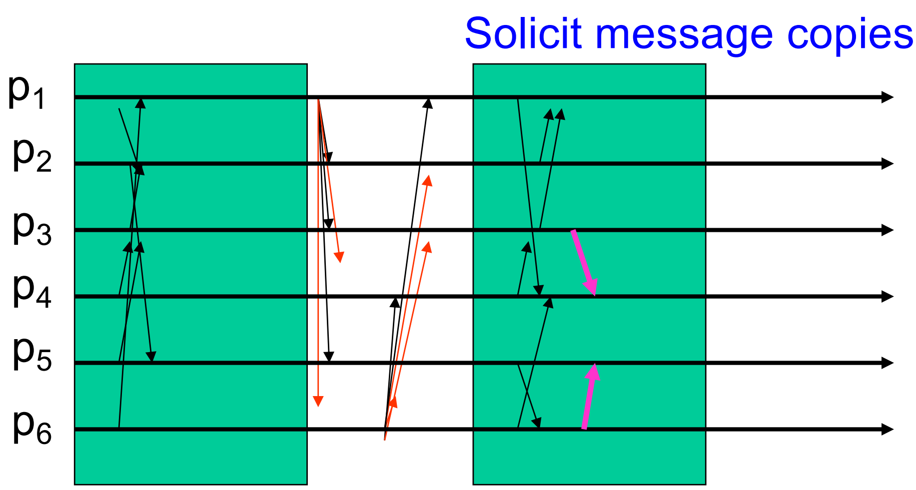
- Start by using *unreliable* multicast to rapidly distribute messages
- Partial distribution of multicast messages may occur
  - Some message may not get through
  - Some process may be faulty

Bimodal multicast: gossip repair



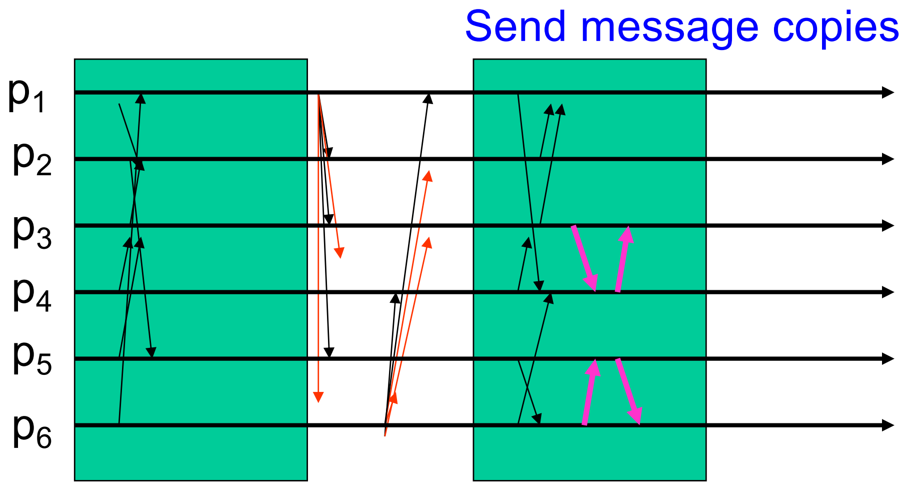
- Periodically (e.g., every 100 ms) each process sends a *digest* describing its state to *some randomly* selected process
- Digest only identifies messages, without including them

Bimodal multicast: gossip repair



- Recipient checks gossip digest against its own history and *solicits* a copy of any missing message from the process that sent the gossip

Bimodal multicast: gossip repair



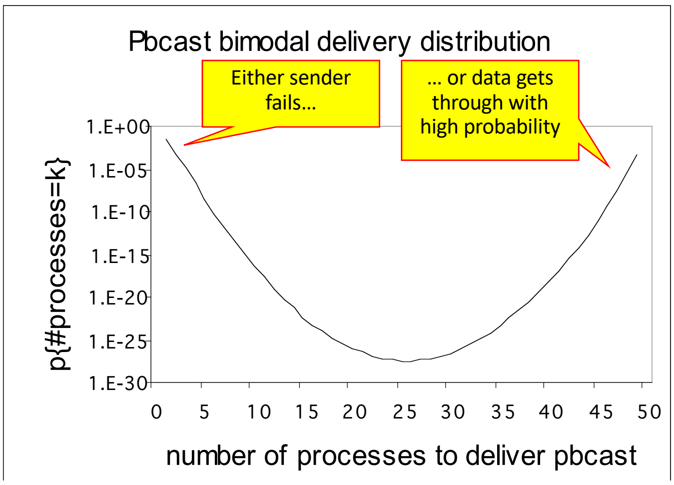
- Processes reply to solicitations received during a gossip round by retransmitting the requested message
- Some optimizations (not examined)

Bimodal multicast: why "bimodal"?

- Are there two phases?
- Nope; description of dual "modes" of result

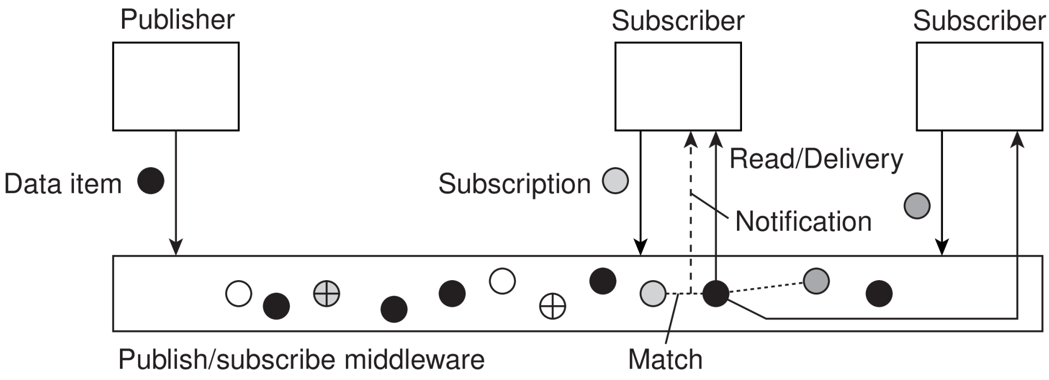
1. pbcast is almost always delivered to most or to few processes and almost never to some processes

   Atomicity = almost all or almost none



2. A second bimodal characteristic is due to delivery latencies, with one distribution of very low latencies (messages that arrive without loss in the first phase) and a second distribution with higher latencies (messages that had to be repaired in the second phase)

Publish-subscribe: subscription



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

Event matching: centralized architecture

- Naive solution: centralized architecture
  - Single server handles all subscriptions and notifications
- Server:
  - 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

- How to achieve matching scalability?
- Simple solution: partition subscriptions; how?

1. Hierarchical architecture: master distributes matching across multiple workers
   - Each worker stores and handles a subset of subscriptions
   - Master receives events and distribute them among workers for matching
   - *How* to partition?
     - Topic-based pub/sub: hash on topics' names to map subscriptions and events to workers

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

Event matching: distributed architecture

- Other solutions: decentralized servers organized into overlay network
- How to route notifications to subscribers?

1. Unstructured overlay: flooding or gossiping to disseminate event notifications
   - Store a subscription only at one server, while disseminating notifications to all servers: in this way, matching is distributed across servers
   - Selective routing helps to avoid disseminating notifications to all servers: install filters that effectively ignore paths toward nodes that are not interested in what is being published
2. Structured overlay: DHT to disseminate event notifications

References

- Chapter 4 and Section 5.6 of van Steen & Tanenbaum book
- RabbitMQ https://www.rabbitmq.com/
https://www.rabbitmq.com/tutorials
- Kafka doc. https://kafka.apache.org/documentation/
- 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, Springer, 2018
- 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