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. 2023/24

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Message-oriented communication

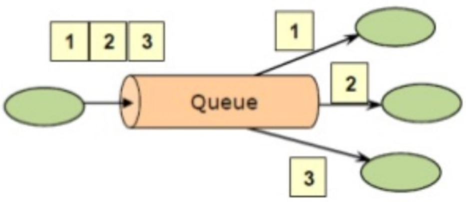
- RPC improves distribution transparency with respect to socket programming
- But still synchrony between interacting entities
  - Over time: caller waits the reply
  - In space: shared data
  - Functionality and communication are coupled
- Which communication models to improve decoupling and flexibility?
- 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

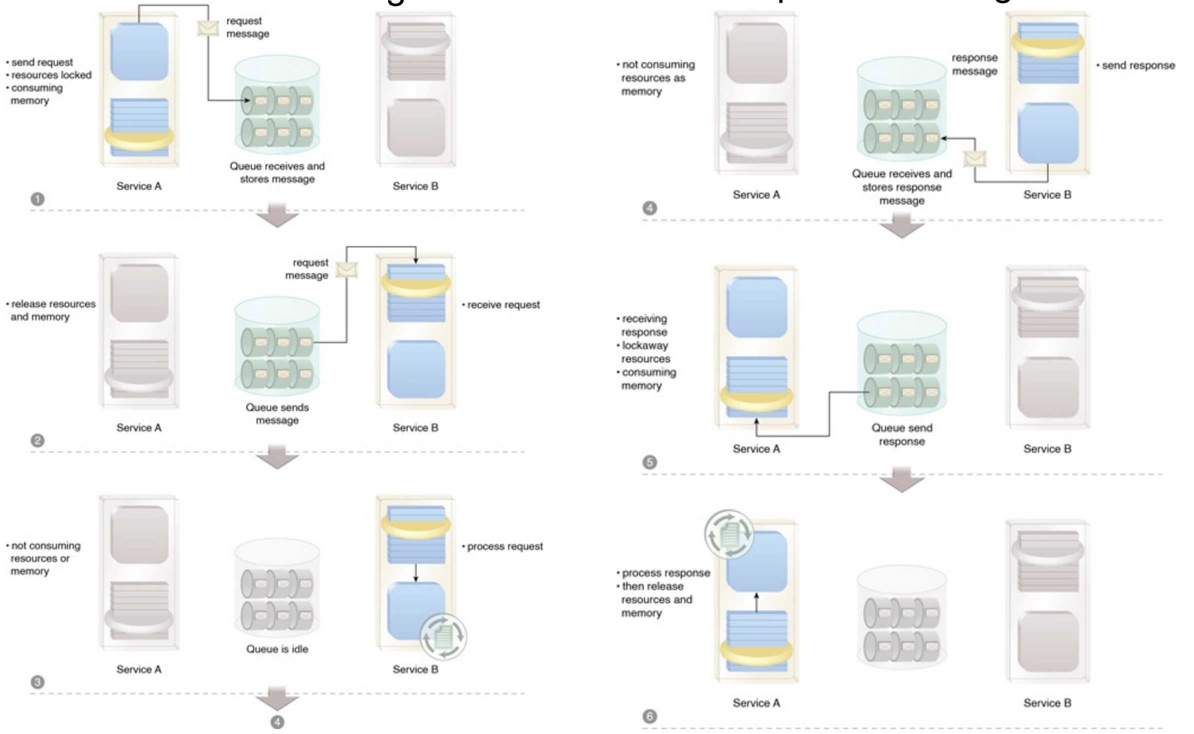
- Messages sent to the queue are stored until they are retrieved by the consumer
- Multiple producers can send messages to queue
- Multiple consumers can receive messages from queue
- But communication is one-to-one: each message from a producer is delivered to a single consumer



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

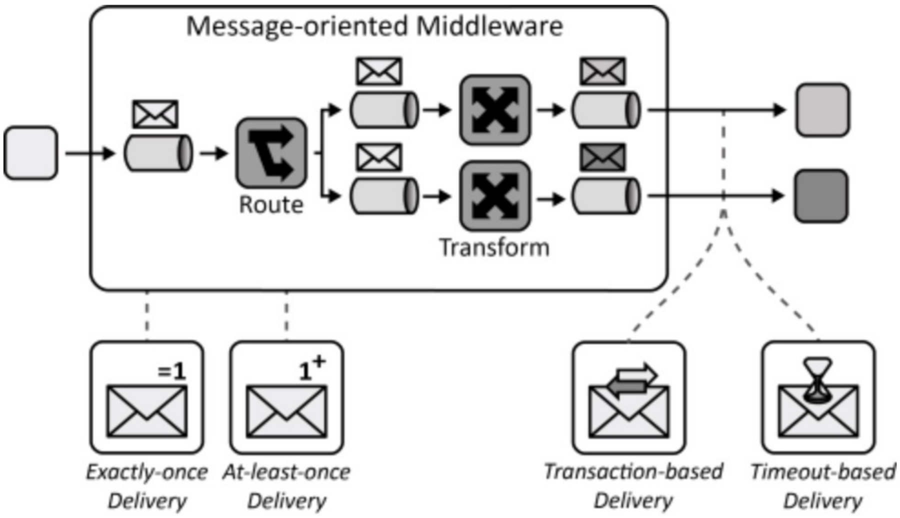
- A sibling of message queue pattern but further generalizes it by delivering a message to multiple consumers
  - Message queue: delivers messages to *only one* receiver, i.e., one-to-one communication
  - Pub/sub channel: delivers messages to *multiple* receivers, i.e., one-to-many communication

Publish/subscribe API

- Calls that capture the core of any pub/sub system:
  - publish(event): to publish an event
    - Events can be of any data type supported by the given implementation languages and may also contain meta-data
  - subscribe(filter expr, notify_cb, expiry) → sub handle: to subscribe to an event
    - Takes a filter expression, a reference to a notify callback for event delivery, and an expiry time for the subscription registration.
    - Returns a subscription handle
  - unsubscribe(sub handle)
  - notify_cb(sub_handle, event): called by the pub/sub system to deliver a matching event

MOM functionalities

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



Cloud Computing Patterns,
www.cloudcomputingpatterns.org/message_oriented_middleware

MOM functionalities

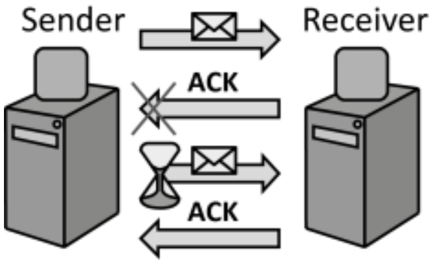
- Let us analyze
  - Delivery semantics
  - Message routing
  - Message transformations

Delivery semantics in MOM

At-least-once delivery

How can MOM ensure that messages are received successfully?

- By sending ack for each retrieved message and resending message if ack is not received
- Design your application to be *idempotent* (not be affected adversely when processing the same message more than once)

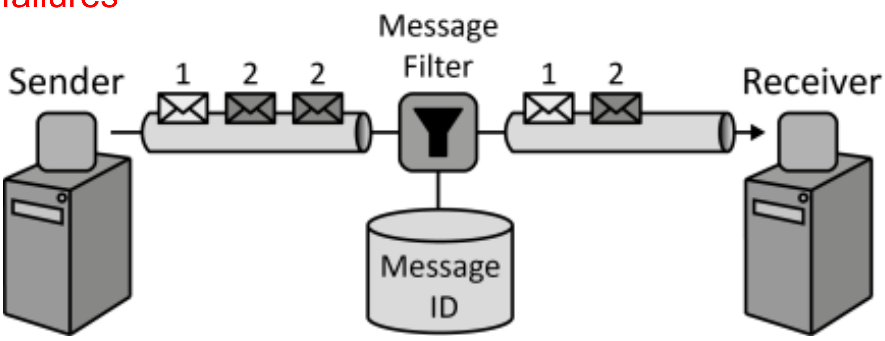


Delivery semantics in MOM

Exactly-once delivery

How can MOM ensure that a message is delivered only exactly once to a receiver?

- By filtering possible message duplicates automatically
  - Upon creation, each message is associated with a unique ID, which is used to filter message duplicates during their traversal from sender to receiver
- In addition, messages must survive MOM components' failures

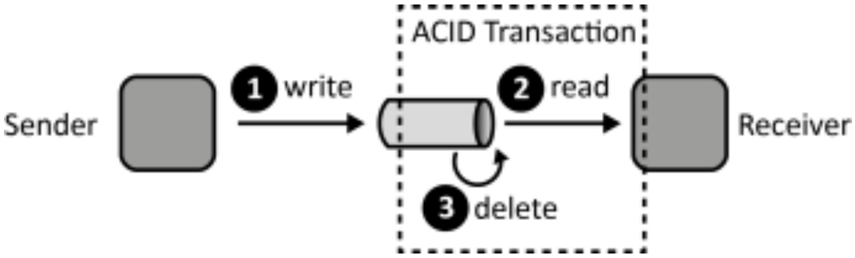


Delivery semantics in MOM

Transaction-based delivery

How can MOM ensure that messages are only deleted from a message queue if they have been received successfully?

- MOM and message receiver participate in a transaction: read and delete operations are performed within a transaction, thus guaranteeing ACID behavior

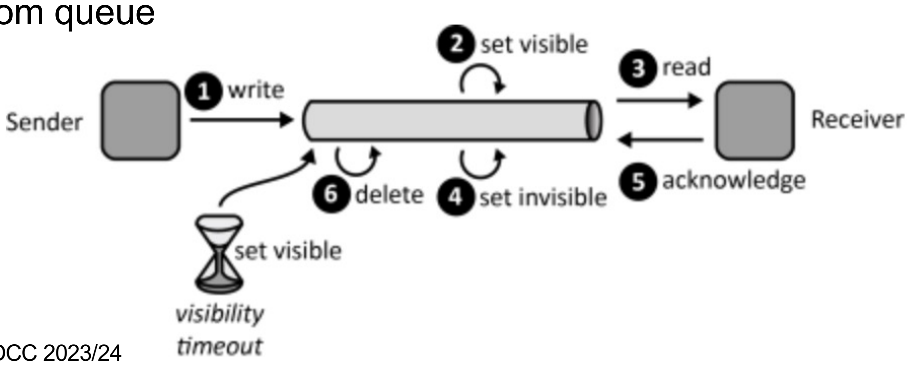


Delivery semantics in MOM

Timeout-based delivery

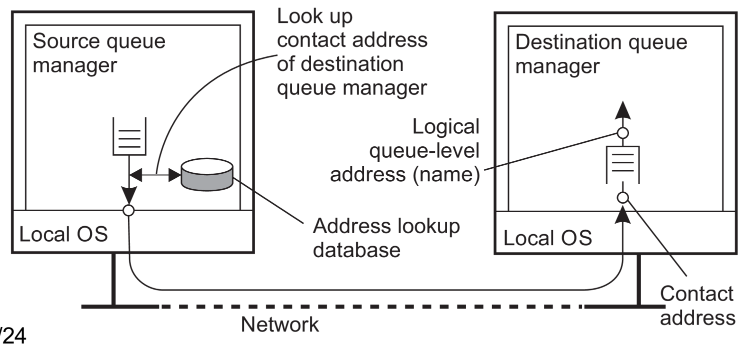
How can MOM ensure that messages are only deleted from a message queue if they have been received successfully at least once?

- Message is not deleted immediately from queue, but marked as being invisible until *visibility timeout* expires
- Invisible message cannot be read by another receiver
- After receiver's ack of message receipt, message is deleted from queue



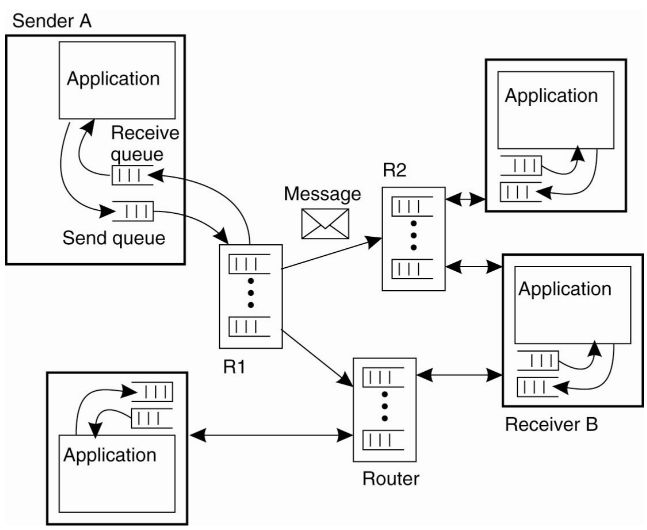
Message routing: general model

- 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 is used 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 dynamically manage mapping between queue names and their location

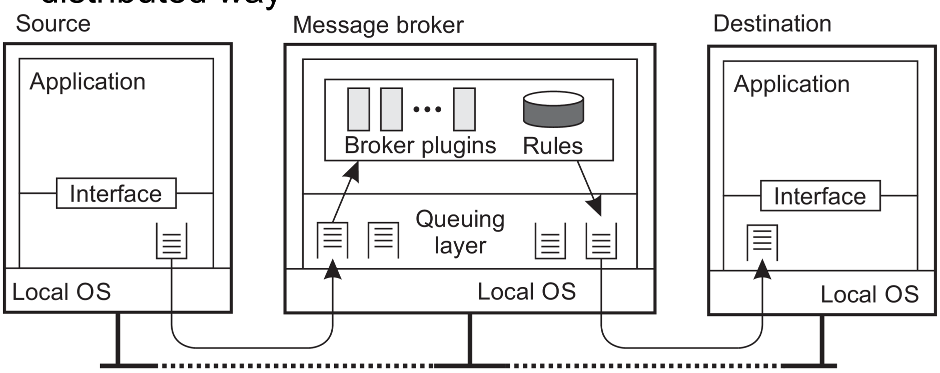


Message transformation: message broker

- New/existing apps that need to be integrated into a single, coherent system rarely agree on a common data format
- How to handle data heterogeneity?
  - We have already examined different solutions in the context of RPC
- Let's focus on ***message broker***
  - Message broker: component that usually takes care of application heterogeneity in a MOM

Message broker: general architecture

- Message broker handles application heterogeneity
  - Converts incoming messages to target format providing access transparency
  - Very often acts as an application gateway
  - Manages a repository of conversion rules and programs to transform a message of one type to another
  - May provide subject-based routing capabilities
  - To be scalable and reliable can be implemented in a distributed way



MOM frameworks

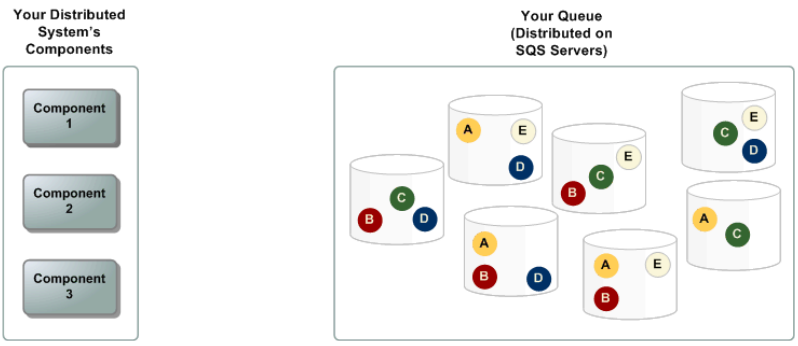
- Main MOM systems and libraries
  - Apache ActiveMQ activemq.apache.org
  - **Apache Kafka**
  - Apache Pulsar pulsar.apache.org
  - IBM MQ
  - NATS nats.io
  - **RabbitMQ**
  - ZeroMQ zeromq.org
- Clear distinction between queue message and pub/sub patterns is often lacking
  - Some frameworks support both (e.g., Kafka, NATS)
  - Others not (e.g., pub/sub in Redis 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

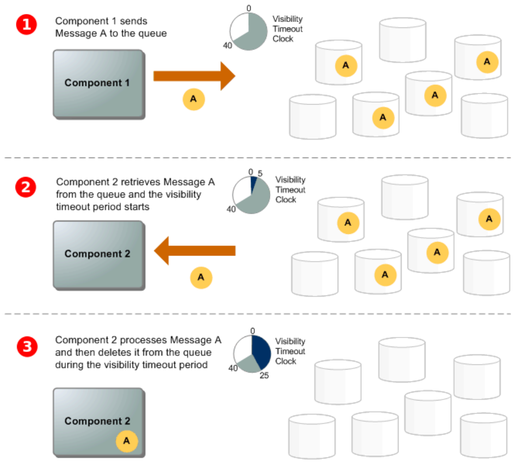
Amazon Simple Queue Service (SQS)

- Reliable, highly-scalable Cloud-based message queue service based on polling model
  - Goal: decouple application components, which can run independently and asynchronously and be developed with different technologies and languages
- Features
  - Message queues are fully managed by AWS
  - SQS servers are replicated within a single region: SQS stores copies of messages on multiple servers for HA



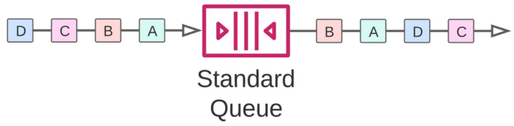
Amazon SQS: Features

- 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 a queue
  - *Short* polling: SQS queries only a subset of servers
  - *Long* polling: SQS queries all servers for messages
- 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



- 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

Amazon SQS: API

- `CreateQueue`, `ListQueues`, `DeleteQueue`
  - Create, list, delete queues
- `SendMessage`
  - Add message to the specified 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 in the message
- `ReceiveMessage`
  - Retrieve message from the specified queue
  - Can't specify which messages to retrieve, only maximum number of messages (up to 10)
- `DeleteMessage`
  - Remove the specified message from the specified queue

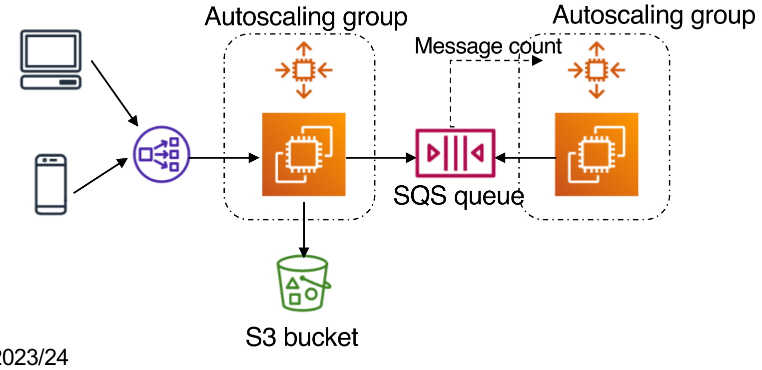
docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/Welcome.html

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 receiver)
  - 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 achieve decoupling between app front-end and back-end, load balancing and fault tolerance
  - App front-end sends to queue a message with S3 link to image
  - A pool of EC2 instances takes a request from queue and resizes image
    - 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



RabbitMQ

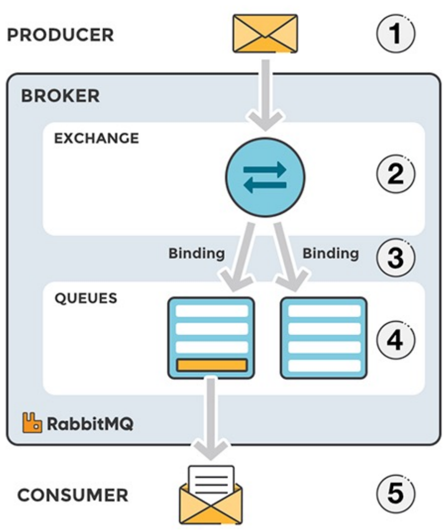
- Popular open-source message broker www.rabbitmq.com

- Uses a push 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 developer tools for most 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

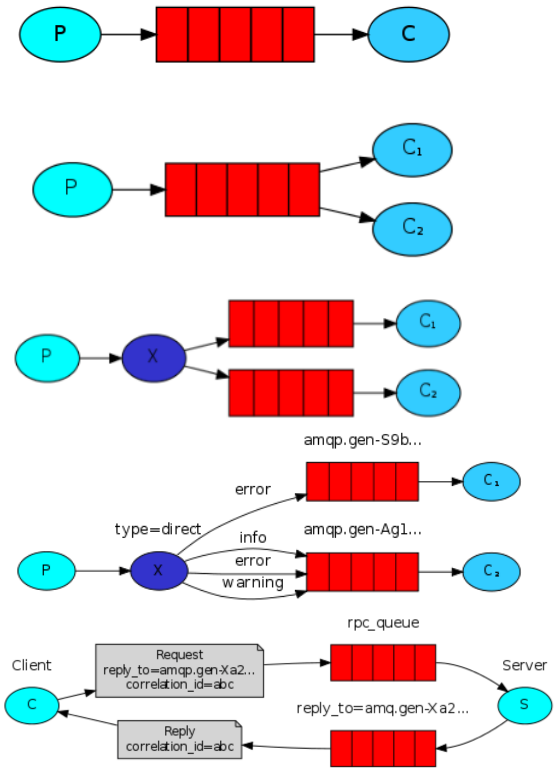
Message flow in RabbitMQ



- RabbitMQ broker can be distributed, e.g., forming a cluster 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 which are sent by a producer and received by a consumer (message queue pattern)
2. Distribute tasks among multiple workers (competing consumers pattern)
3. Deliver messages to many consumers at once (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 the result (request /reply pattern)



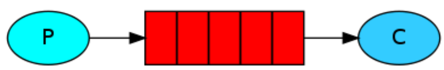
www.rabbitmq.com/getstarted.html

RabbitMQ and Go

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

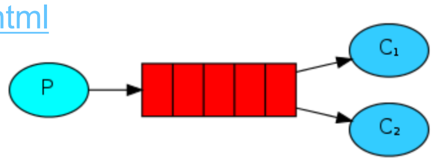
Ex. 1: Message queue pattern

www.rabbitmq.com/tutorials/tutorial-one-go.html



Ex. 2: Competing consumers pattern

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



Code available on course site: rabbitmq-go.zip

RabbitMQ and Go

- Preliminary steps:

1. Install RabbitMQ and start a RabbitMQ server on localhost on default port 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

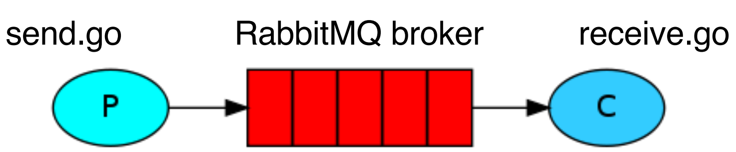
2. Install Go AMQP client library

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

   See pkg.go.dev/github.com/rabbitmq/amqp091-go for details on Go package ampq

RabbitMQ and Go: example 1

1. Message queue pattern
   – Run with 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. Competing consumers (i.e., workers) pattern

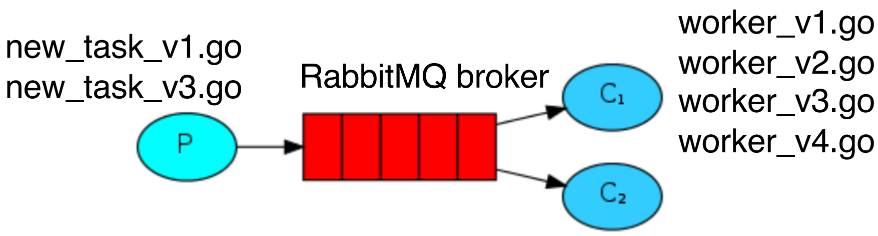
- Version 1 (new_task_v1.go and worker_v1.go):
  - Use multiple consumers to see how queue can be used to distribute tasks among consumers in *round-robin* fashion
  - If consumer crashes after RabbitMQ delivers the message but before completing the task, the message is lost (i.e., cannot be delivered 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 is the delivery semantics with explicit acks?

RabbitMQ and Go: example 2

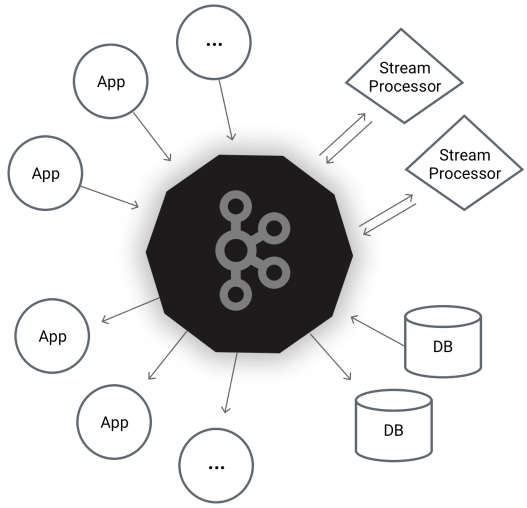
2. Competing consumers (i.e., workers) pattern

- Version 3 (new_task_v3.go and worker_v3.go):
  - Use a durable queue so it is persisted to disk and survives RabbitMQ crash and restart
  - Define a new queue and set `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`)



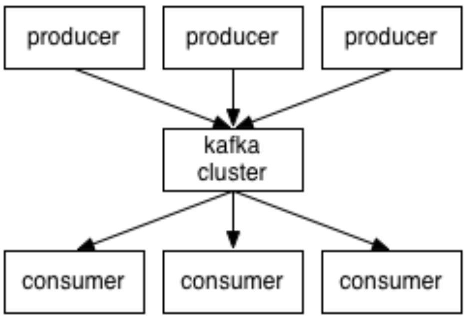
Apache Kafka

- General-purpose, distributed pub/sub system
- Originally developed in 2010 by LinkedIn
- Used at scale by tech giants (Netflix, Uber, LinkedIn, …)
- Written in Scala
- Horizontally scalable
- Fault-tolerant
- High throughput ingestion
  - Billions of messages
- Not only messaging, also data processing
  - We focus on messaging



kafka.apache.org/documentation
Kreps et al., Kafka: A Distributed Messaging System for Log Processing, NetDB'11

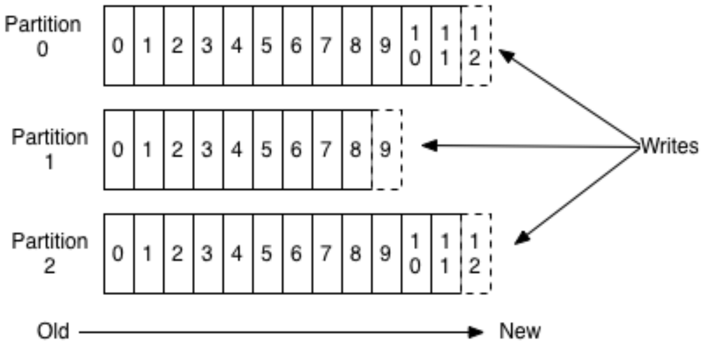
Kafka at a glance



- Kafka stores feeds of messages (or records) in categories called topics
  - A topic can have 0, 1, or many consumers subscribing to data written to it
- Producers: publish messages to a Kafka topic
- Consumers: subscribe to Kafka topics and process the feed of published messages
- Kafka cluster: distributed log of data over servers known as brokers
  - A broker is responsible for receiving and storing published data

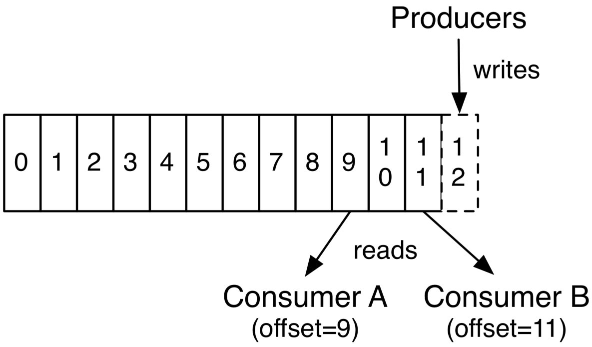
Kafka: topic and partitions

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



Kafka: partitions

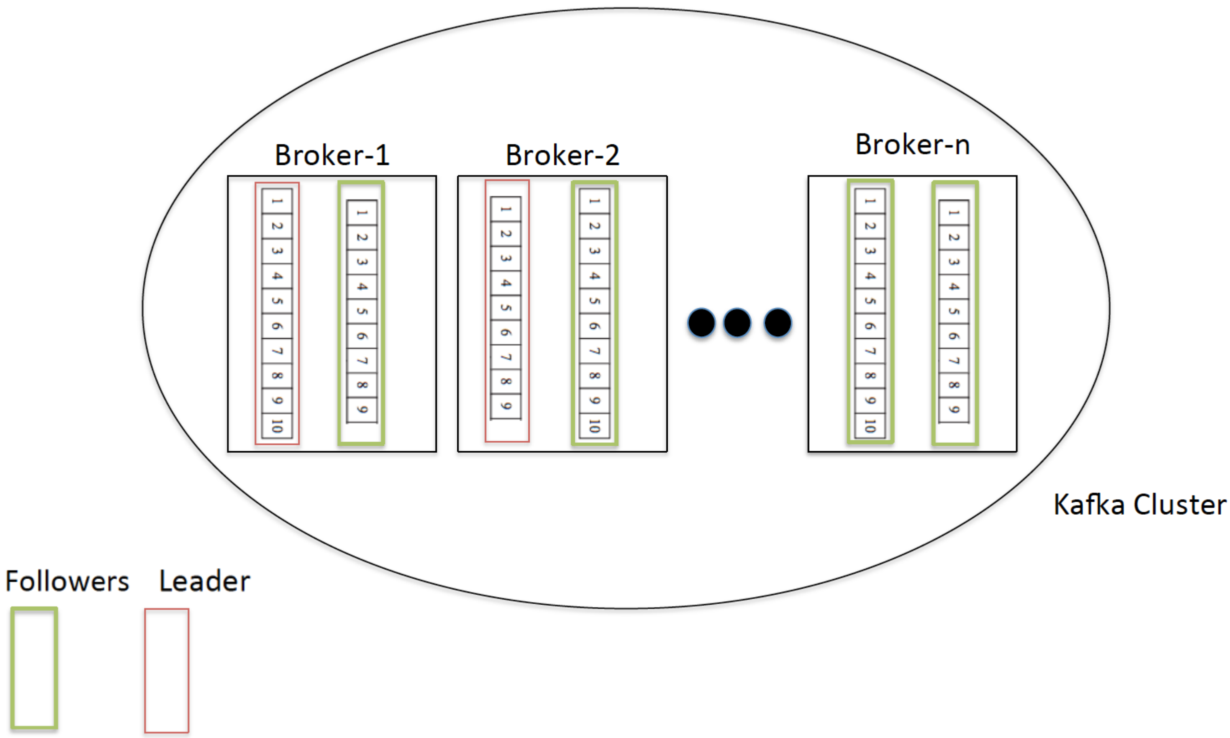
- Producers publish (write) their messages to a topic partition
- Consumers read records published on a topic
- Each partition is an *ordered*, *numbered*, *immutable* sequence of records that is continually *appended to*
  - Like a commit log
- Each record is associated with a monotonically increasing sequence number, called offset



Kafka: partitions and design choices

- To improve scalability: partitions are *distributed* across brokers
  - By distributing partitions on multiple brokers, I/O throughput increases
  - Parallel reads and writes on partitions of the same topic
    - Multiple producers can write in parallel
    - Multiple consumers can read in parallel
- To improve fault tolerance: each topic partition can be *replicated* across a configurable number of brokers
  - Driven by *replication-factor* (equal to total number of replicas including the leader)
  - If *replication-factor* = *N*, up to *N*-1 brokers can fail before losing access to data
  - Each partition has one leader broker and 0 or more followers
    - followers > 0 in case of replication

Kafka: partition leader and followers



Kafka: partitions and design choices

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

- 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

Kafka: producers

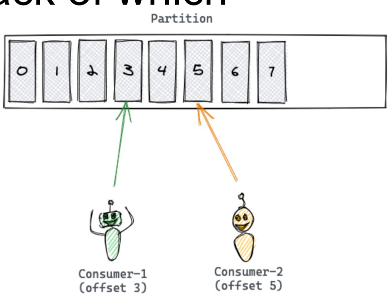
- Producers = data sources
- Publish data to topics of their choice
  - Producer sends data directly (i.e., without any routing tier) to the broker that is the leader for the partition
- Producer is responsible for choosing which message to assign to which partition within the topic: how?
  - *Key-based* partitioned, i.e., the producer uses a partition key to direct messages to a specific partition
    - E.g., if user id is the key, all data for a given user will be published in the same partition
  - Round-robin (default, if key is not specified)
- Multiple producers can write to the same partition

Design choice for consumers

- Push or pull model for consumers?
- **Push** model
  – Broker actively pushes messages to consumers
  – Challenging for broker to deal with different types of consumers as it controls the rate at which data is transferred
  – Need to decide whether to send a message immediately or accumulate more data and then send
- **Pull** model
  – Consumer is in charge of retrieving messages from broker
  – Consumer has to maintain an offset to identify the next message to be transmitted and processed
  ✓ Better scalability (less burden on brokers) and flexibility (different consumers with diverse needs and capabilities)
  ✗ In case broker has no data, consumers may end up busy waiting for data to arrive

Kafka: consumers

- Kafka uses a pull approach for consumers
  kafka.apache.org/documentation.html#design_pull
- Consumer uses the offset to keep track of which messages it has already consumed
- A partition can be consumed by more consumers, each reading at different offsets



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

Kafka: brokers

- Kafka brokers store messages reliably on disk
- Differently from traditional queue message and pub/sub systems, Kafka does not delete messages after delivery
- Topics are configured with a *retention time* that specifies how long messages should be stored on disk
  - Topic retention can also be specified in bytes instead of time

Hands-on Kafka

- Preliminary steps:
  - Download and install Kafka kafka.apache.org/downloads
    - Zookeeper comes included with Kafka
    - Configure Kafka properties in `server.properties` (e.g., `listeners` and `advertised.listeners`)
  - Start Kafka environment

    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
```

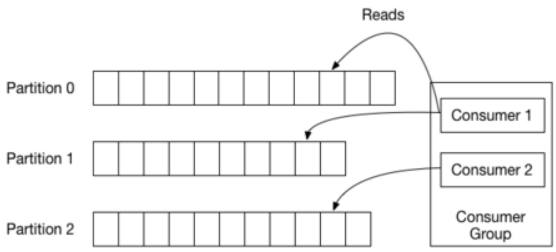
- Stop Kafka and Zookeeper

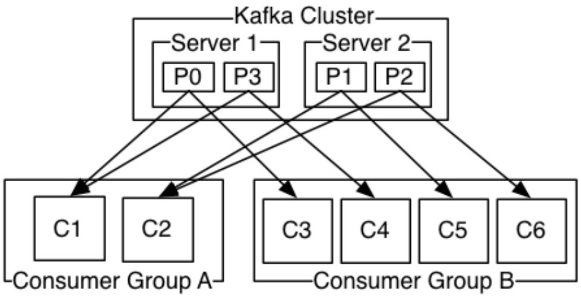
```
$ kafka-server-stop
$ zookeeper-server-stop
```

  Alternatively `$ zKserver stop`

Kafka: consumer group

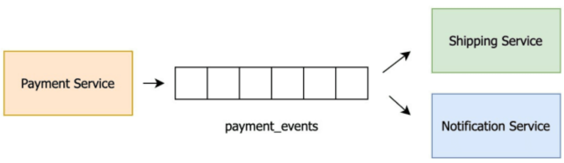
- Consumer Group: set of consumers which cooperate to consume data from some topic and share a group ID
  - A Consumer Group maps to a *logical* subscriber
  - Topic partitions are divided among consumers in the group for load balancing and can be reassigned in case of consumer join/leave
  - Every message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



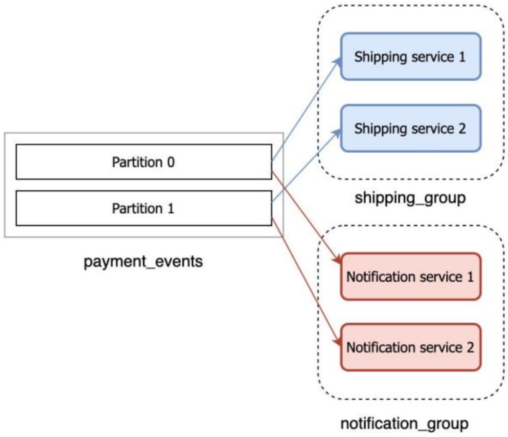


Kafka: consumer group

- How to have many consumers reading the same messages from the topic?
  - Need to use different group IDs
- Example: microservices communicate using Kafka



- How to scale?

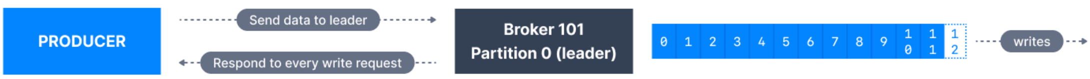


Kafka: ordering guarantees

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

Kafka: delivery semantics

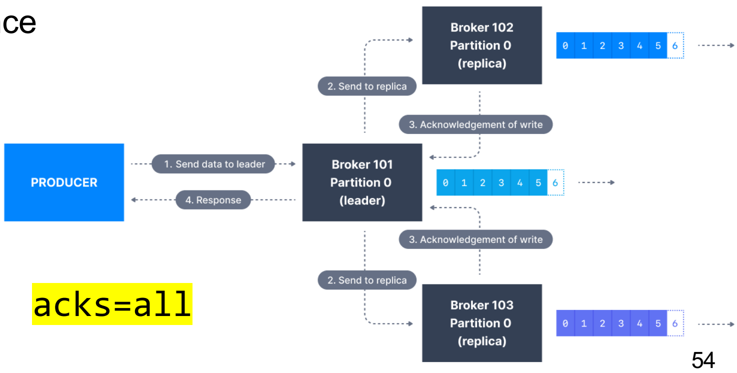
- Delivery guarantees supported by Kafka
  - **At-least-once** (default): guarantees no message loss, but messages may be duplicated and out-of-order (with respect to producer)
    - Producer: wait for ack from partition leader; if none, retry
    - How? Set `acks=1`
    - Consumer: commit offset after processing the message



acks=1

Kafka: delivery semantics

- Delivery guarantees supported by Kafka
  - **Exactly-once**: guarantees no message loss, no duplicates and partition-level ordering, at the cost of 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 message sequence number in each message 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

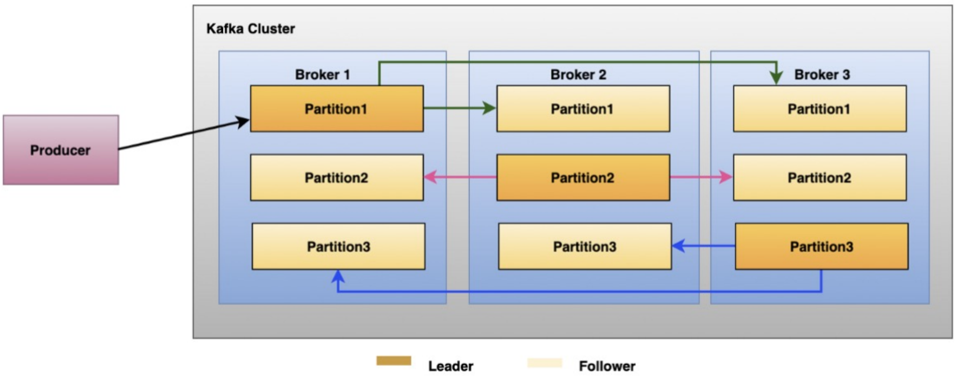
- Delivery guarantees supported by Kafka
  - User can also implement **at-most-once**: messages may be lost but are never re-delivered
    - Producer: disable retries (i.e., `acks=0`)
    - Consumer: commit offset before processing the message

- Take-away message: you need to choose the semantic that makes sense for your application context

See kafka.apache.org/documentation/#semantics

Kafka: fault tolerance

- Kafka replicates topic partitions for fault tolerance
  - Leader coordinates to update followers when new messages arrive
  - The set of in-sync replicas is known as ISR



- In case of leader crash, a follower can be elected as new leader with the help of Zookeeper or KRaft

Kafka: fault tolerance

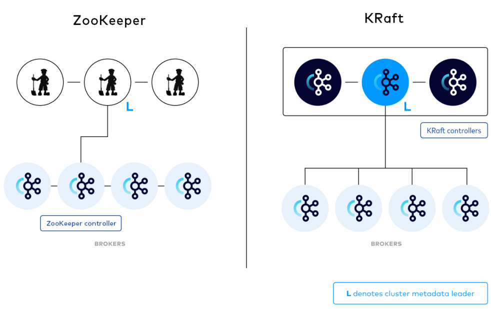
- Kafka makes a message available for consumption only after all replicas in the ISR for that partition have applied it to their log
  - Messages may not be immediately available for consumption: tradeoff between consistency and availability
- Producers have the option of either waiting for the message to be committed or not (by setting `acks`)
  - Tradeoff between latency and durability
- Kafka retains messages for a configured period of time
  - To free up disk space, messages have a retention time; upon expiry, messages are marked for deletion
  - Alternatively, retention can be based on message size

Kafka and ZooKeeper

- Zookeeper: hierarchical, distributed key-value store
  zookeeper.apache.org
  – Coordination service for distributed systems, which provides facilities for supporting various coordination tasks, including locking, leader election, monitoring
  – ZooKeeper maintains a namespace, organized as a tree
  – Simple operations on the tree: creating and deleting nodes, as well as reading and updating the data contained in a node
  – Used within many open-source distributed systems
- Kafka uses ZooKeeper for metadata management
  - List of brokers in Kafka cluster
  - Configuration for topics and permissions
  - Leader election: to determine the leader of a given partition
  - Zookeeper allows Kafka to know about changes (e.g., new topic, deleted topic, broker crashes, broker restarts)

From ZooKeeper to KRaft

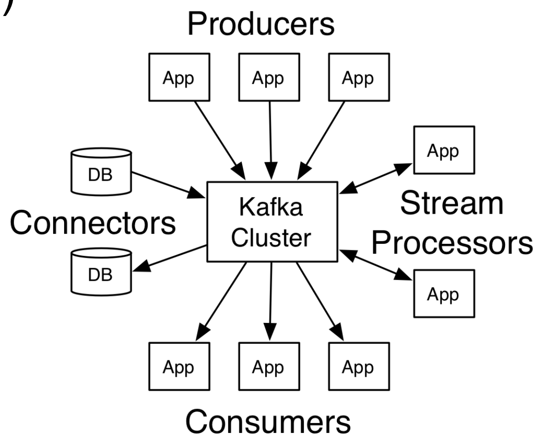
- Zookeeper cons
  ✗ Different system for metadata management and consensus
  ✗ Can become bottleneck as Kafka cluster grows
- New release: ~~Zookeeper~~ Apache Kafka Raft (KRaft)
  – Kafka cluster metadata is stored in Kafka cluster itself
  ✓ Simpler architecture
  ✓ Faster and more scalable metadata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



Kafka: APIs

kafka.apache.org/documentation/#api

- 5 core APIs (Java and Scala only)
- Producer API: to publish data to Kafka topics
- Consumer API: to read data from Kafka topics
- Kafka Connect API: to build and run reusable connectors (producers or consumers) that connect Kafka topics to apps or external systems (source or sink)
  - Many pre-built connectors you can directly use: AWS S3, RabbitMQ, MySQL, Postgres, AWS Lambda, …



Kafka: APIs

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

Kafka: client library

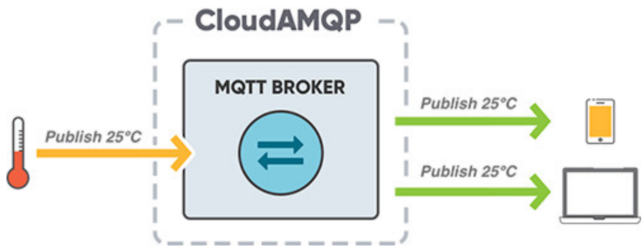
- Kafka officially provides only SDK for Java
- For other languages, implementations of client library provided by community, including
  - Go
    github.com/confluentinc/confluent-kafka-go
    github.com/segmentio/kafka-go
  - Python
    github.com/confluentinc/confluent-kafka-python

Messaging protocols

- Not only systems but also open standard protocols for message queues
  - **AMQP** Advanced Message Queueing Protocol
    - Binary protocol
  - MQTT Message Queue Telemetry Transport
    - Binary protocol
  - STOMP Simple (or Streaming) Text Oriented Messaging Protocol
    - Text-based protocol
- Goals:
  - Platform- and vendor-agnostic
  - Provide interoperability between different MOMs

Messaging protocols and IoT

- Often used in Internet of Things (IoT)
  - Use message queueing protocol to send data from sensors to services that process those data



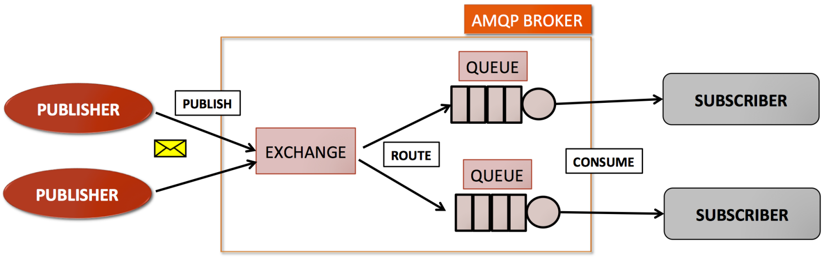
  - Exploit all MOM advantages seen so far:
    - Decoupling
    - Resiliency: MOM provides a temporary message storage
    - Traffic spikes handling: data will be persisted in MOM and processed eventually

AMQP: characteristics

- Open-standard protocol for MOM, supported by industry
  - Current version: 1.0 docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf
  - Approved in 2014 as ISO and IEC International Standard
- Binary, application-level protocol
  - Based on TCP protocol 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



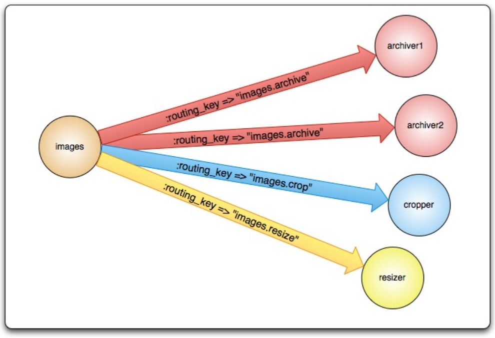
- AMQP entities (within broker): queues, exchanges and bindings
  - Messages are published to *exchanges* (like post offices or mailboxes)
  - Exchanges distribute message copies to *queues* using rules called *bindings*
  - AMQP brokers either push messages to consumers subscribed to queues, or consumers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

www.rabbitmq.com/tutorials/amqp-concepts.html

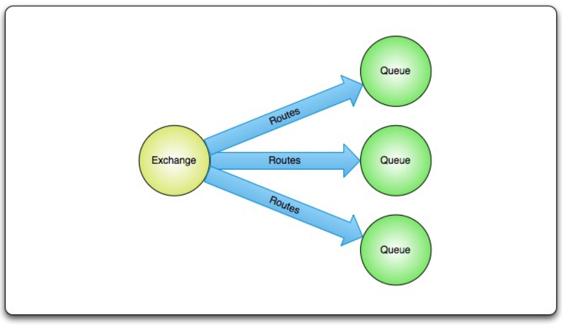
AMQP: routing

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

Direct exchange routing



Fanout exchange routing

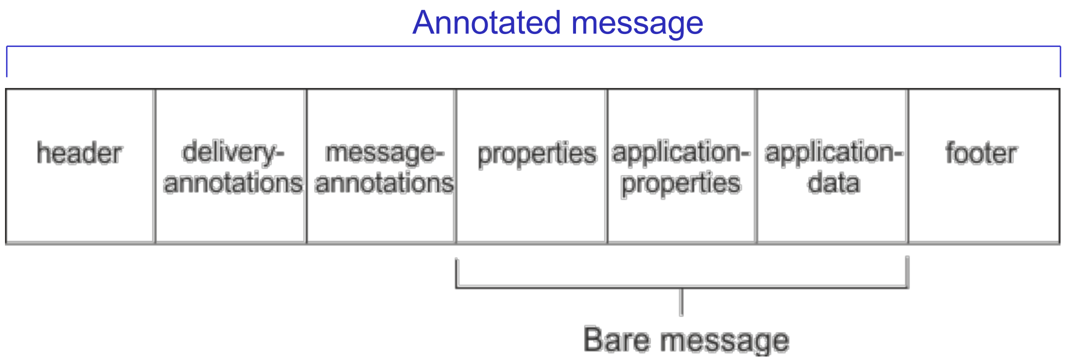


AMQP: routing

- 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 various 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



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
  - Broadcast communication*:* special case of multicast, in which data is sent to *all* receivers
  - Examples of one-to-many multicast apps: video/audio resource distribution, file distribution
  - Examples of many-to-many multicast apps : conferencing tools, multiplayer games, interactive distributed simulations
- Cannot be implemented as unicast replication (source sends as many copies as the number of receivers): 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
  - **Application**-level multicast
    - Replication and routing managed by hosts

Application-level multicast

- Basic idea:
  - Organize nodes into an overlay network
  - Use overlay network to disseminate data
  - Can be structured or unstructured
- Structured application-level multicast
  - Explicit communication paths
  - How to build structured overlay network?
    - Tree: only one path between each pair of nodes
    - Mesh: multiple paths between each pair of nodes
- Unstructured application-level multicast
  - Based on flooding or random walk
  - Based on gossiping

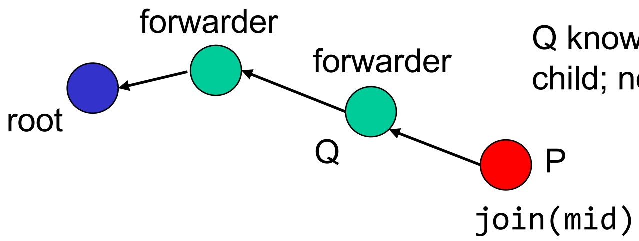
Structured application-level multicast: tree

- Let's consider how to build an application-level multicast tree in Scribe
  - Scribe: pub/sub system with decentralized architecture and based on Pastry (but we use Chord as DHT)
  - Assume a node wants to start a multicast session

1. Multicast initiator node generates multicast identifier `mid`
2. Initiator lookups `succ(mid)` using DHT
3. Request is routed to `succ(mid)`, which becomes root of multicast tree
4. If node *P* wants to join the tree, it executes `lookup(mid)`
5. When request arrives at *Q*:
   - *Q* has not seen a join request for *mid* before ⇒ *Q* becomes *forwarder*, *P* becomes Q's child; join request is forwarded by *Q*
   - *Q* knows about tree ⇒ *P* becomes Q's child; no need to forward join request anymore

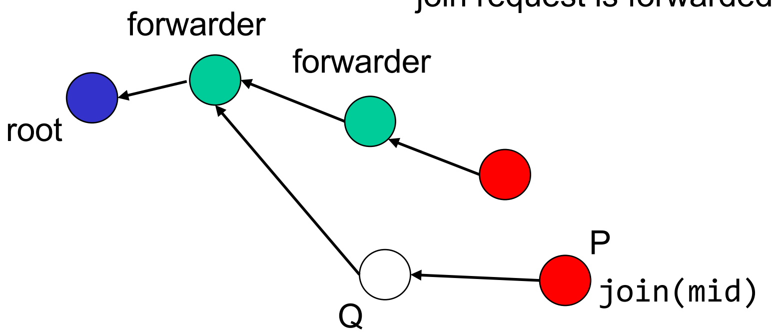
Castro et al., Scribe: A large-scale and decentralised application-level multicast infrastructure, *IEEE JSAC*, 2002

Structured application-level multicast: tree



Q knows about tree ⇒ P becomes Q's child; no need to forward join request

Q has not seen a join request for mid before ⇒ Q becomes forwarder, P becomes Q's child; join request is forwarded by Q



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 that messag (except to *P)* and only if it had not seen *m* before
  - ✓ Random walk
    - With respect to flooding, *m* is sent only to one randomly chosen node
  - 👉 **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, *Proc. of 6th Symp. on Principles of Distributed Computing*, 1987.

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 a gossip-based failure detection service
- BitTorrent uses a gossip-based basic information exchange
- Cassandra uses gossip protocol for group membership and failure detection of cluster nodes
- See gossip dissemination pattern martinfowler.com/articles/patterns-of-distributed-systems/gossip-dissemination.html

Strategies to spread updates

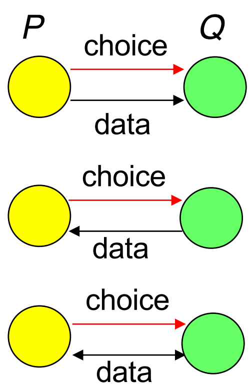
- Let's consider the two principle operations

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) other peers to send updates to (contaminating them)

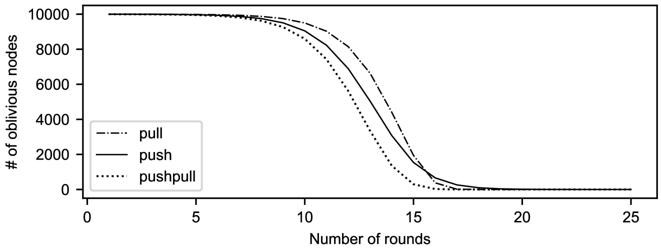
Anti-entropy

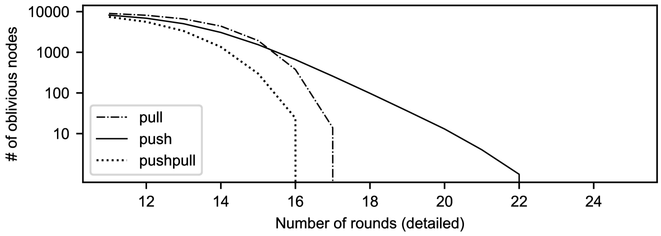
- Goal: increase node state similarity, thus decreasing "disorder" (reason for name!)
- Node *P* selects node *Q* randomly: how does P update *Q*?
- 3 different update strategies:
  - **push**: *P* only pushes its own updates to *Q*
  - **pull**: *P* only pulls in new updates from *Q*
  - **push-pull**: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



Anti-entropy: performance

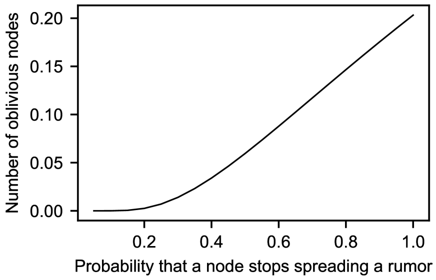
- Push-pull
  - Fastest strategy: takes $O(\log_2 N)$ rounds to disseminate updates to N nodes
  - *Round* (or *gossip cycle*): time interval in which every node takes the initiative to start an exchange





Rumor spreading

- A node P, having an update to report, contacts a randomly chosen node *Q* and forwards the update message to it
- If *Q* was already updated, *P* may lose interest in spreading the update any further and with probability p_{stop} stops contacting other nodes
- The fraction *s* of oblivious nodes (that have not been updated) is $s = e^{-(1/p_{stop}+1)(1-s)}$



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

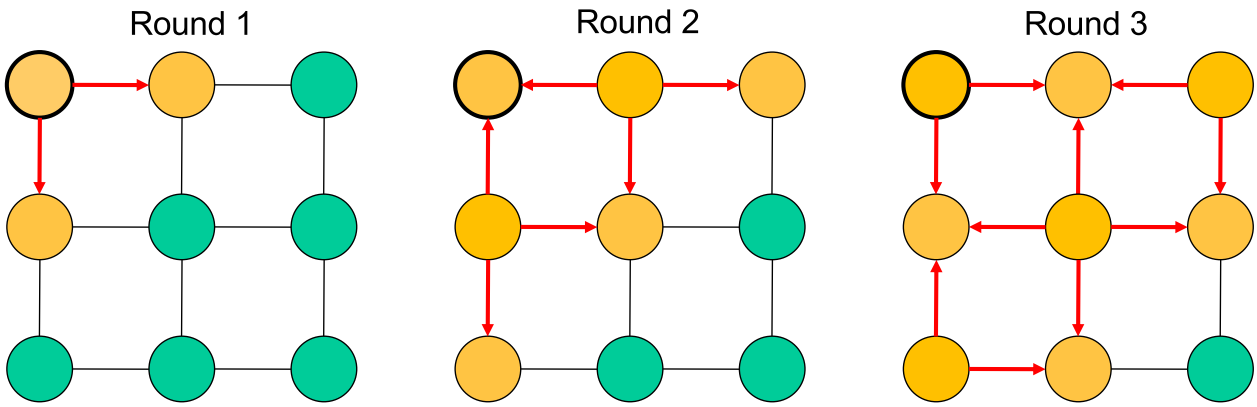
Kermarrec and van Steen, Gossiping in Distributed Systems, *ACM Operating System Review,* 2007

Framework of gossip-based 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 the sender)
  - Message is eventually discarded when TTL=0



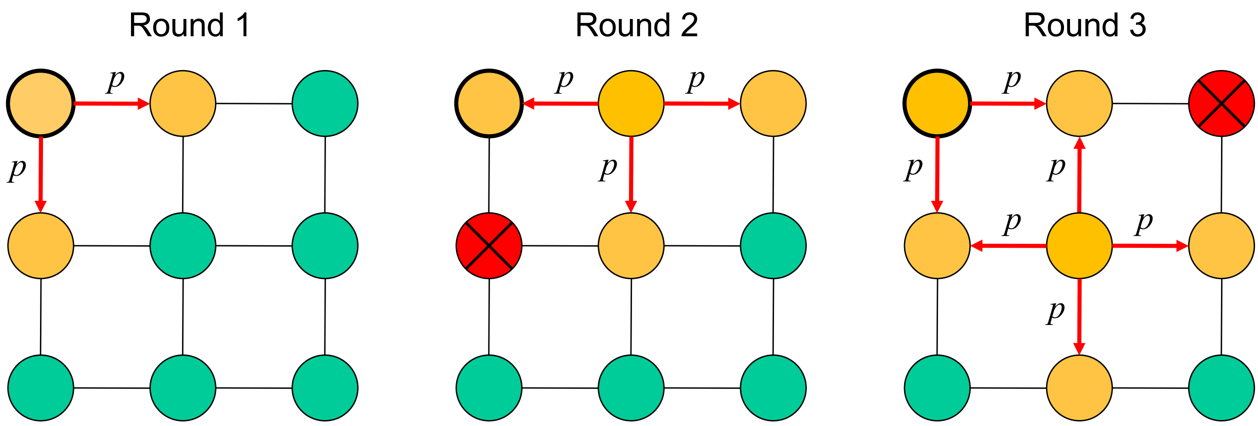
Sent messages: 18
Reached nodes: 8 out of 9

Gossiping vs flooding: example

- Let's use only 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

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

Two algorithms

- Let's consider a gossiping protocol

Blind counter rumor mongering

- And a reliable multicast protocol that exploits gossiping to achieve reliability

Bimodal multicast

Blind counter rumor mongering

- Why such name for this gossiping protocol?
  - *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

- Gossip 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 the message *m* only if *p* has sent it to *r* previously, or if *p* received the message from *r*

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

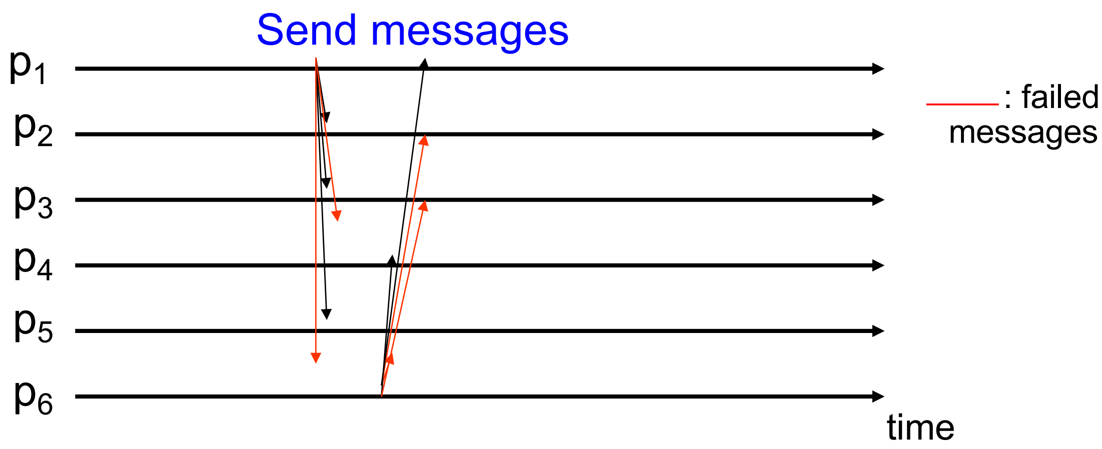
Bimodal multicast

- 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

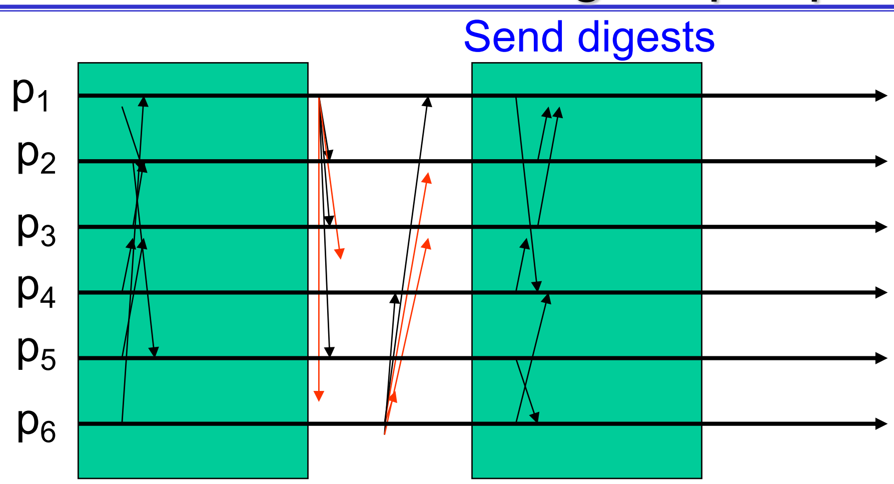
Birman et al., Bimodal multicast, *ACM Trans. Comput. Syst., 1999*

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

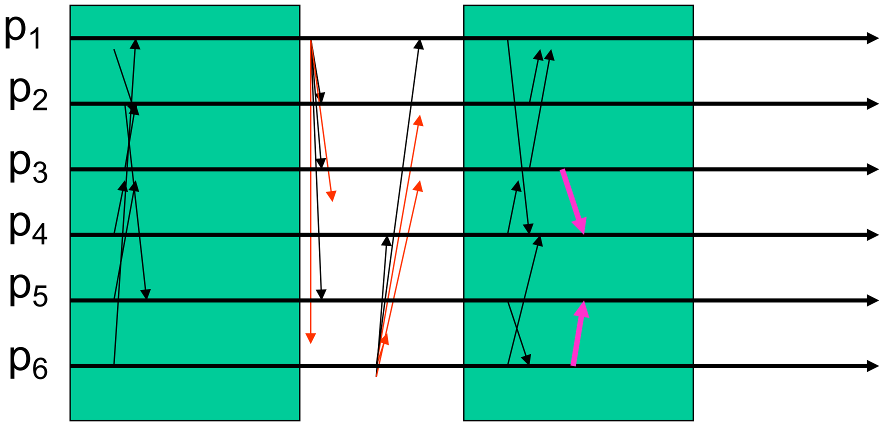
Bimodal multicast: gossip repair



- 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

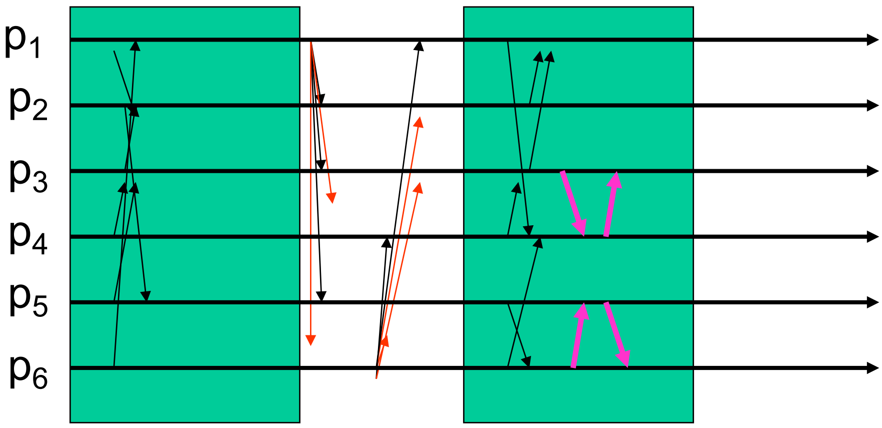
Solicit message copies



- 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

Send message copies



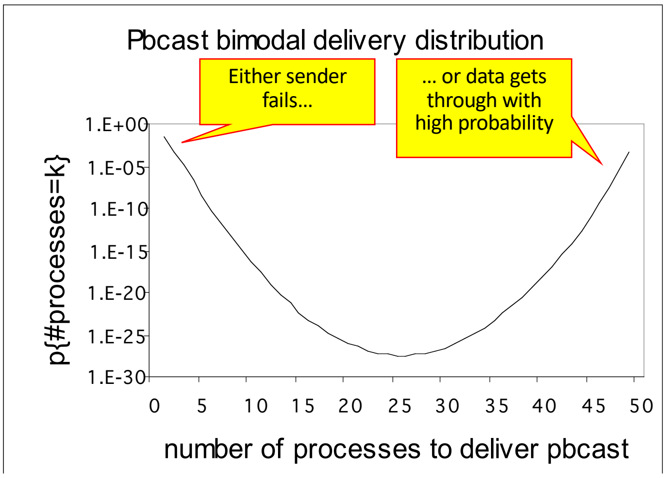
- Processes reply to solicitations received during a gossip round by retransmitting the requested message
- Some optimizations (not examined)

Bimodal multicast: why “bimodal”?

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

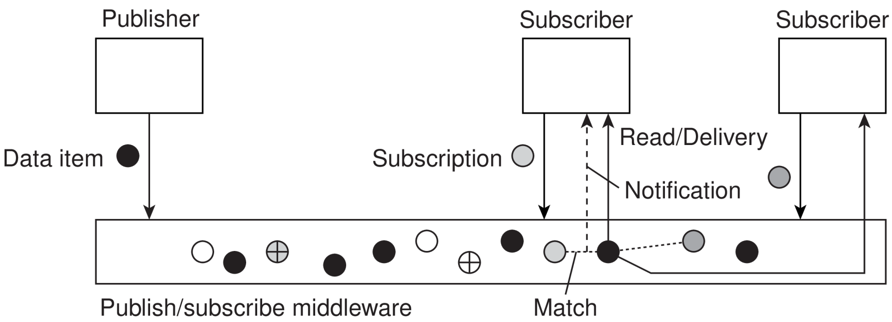
1. pbcast is almost always delivered to most or to few processes and almost never to some processes

   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



- A subscriber specifies in which events it is interested (subscription *S*)
- When a publisher publishes a notification *N* we need to see whether *S* matches *N*
- Challenge: implement the match function in a scalable manner

Distributed event matching: centralized architecture

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

✓ Simple to realize and feasible for small-scale deployments

✗ Scalability

✗ SPOF

Distributed event matching: distributed architecture

- How can we address scalability through distribution?
- Simple solution: partitioning
- Master/worker pattern (i.e., hierarchical architecture): master distributes matching across multiple workers
  – Each worker stores and handles a subset of subscriptions
  – *How* to partition?
    - Simple for topic-based pub/sub: use hashing on topics' names for mapping subscriptions and events to workers

  ✗ Single master
- Alternatively, avoid single master and use a set of distributed servers among which work is spread
  – Organized in a flat architecture, hashing can still be used
  – Example: Kafka

Distributed event matching: distributed architecture

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

1. Unstructured overlay: use flooding or gossiping to disseminate notifications
   - Store a subscription only at one server, while disseminating notifications to all servers: in this way, matching is distributed across the servers
   - Selective routing may help to avoid disseminating notifications to all servers
2. Structured overlay
   - Example: Scribe

References

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

- RabbitMQ, www.rabbitmq.com
- RabbitMQ tutorials, www.rabbitmq.com/tutorials

- Apache Kafka documentation, kafka.apache.org/documentation
- Kreps et al., Kafka: A Distributed Messaging System for Log Processing, NetDB'11
- Sax, Apache Kafka, Encyclopedia of Big Data Technologies, Springer, 2018

- Eugster et al., From epidemics to distributed computing, IEEE Computer, 2004
- Birman et al., Bimodal multicast, ACM TCS 1999
- Portmann and Seneviratne, The cost of application-level broadcast in a fully decentralized peer-to-peer network, ISCC 2002