TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

14/11/22

Communication in Distributed Systems
Part 2

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2022/23

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 — usata per migliorare decoupling e flessibilità
  - Transient
    - Berkeley socket
    - Message Passing Interface (MPI): see "Sistemi di calcolo parallelo e applicazioni" course
  - Persistent
    - **Message Oriented Middleware (MOM)** — Di nostro interesse

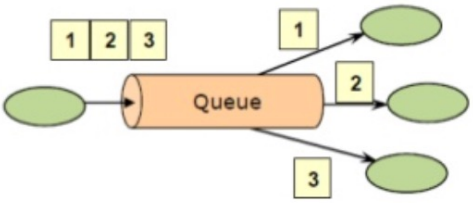
Applicazioni cloud cercano di passare da una struttura monolitica ad una struttura con vari componenti che realizzano le varie funzionalità e sono accoppiate in "modo indipendente" fra loro

Message-oriented middleware

- Communication middleware that supports sending and receiving messages in a persistent way
- Loose coupling among system/application components
  - Decoupling in time and space , i componenti possono essere anonimi.
  - 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 systems:
  - **Message queue system** (MQS)
  - **Pub/sub system**

Queue message pattern

- Messages are stored on the queue until they are processed and deleted
- Multiple consumers can read from the queue
- Each message is delivered only once, to a single consumer



- Example of apps:
  - Task scheduling, load balancing, collaboration

Ho a disposizione una coda che permette di memorizzare in modo persistente dei messaggi e rimangono lì finchè non vengono processati. Dopo definiremo che vuol dire "processato".
Un singolo messaggio è consegnato una volta sola ad un singolo consumatore. (pattern 1-1 point to point).
Possiamo mettere in comunicazione un producer ed un consumer andandoli a disaccoppiare temporalmente (non necessariamente c'e' disaccoppiamento spaziale o di sincronia).

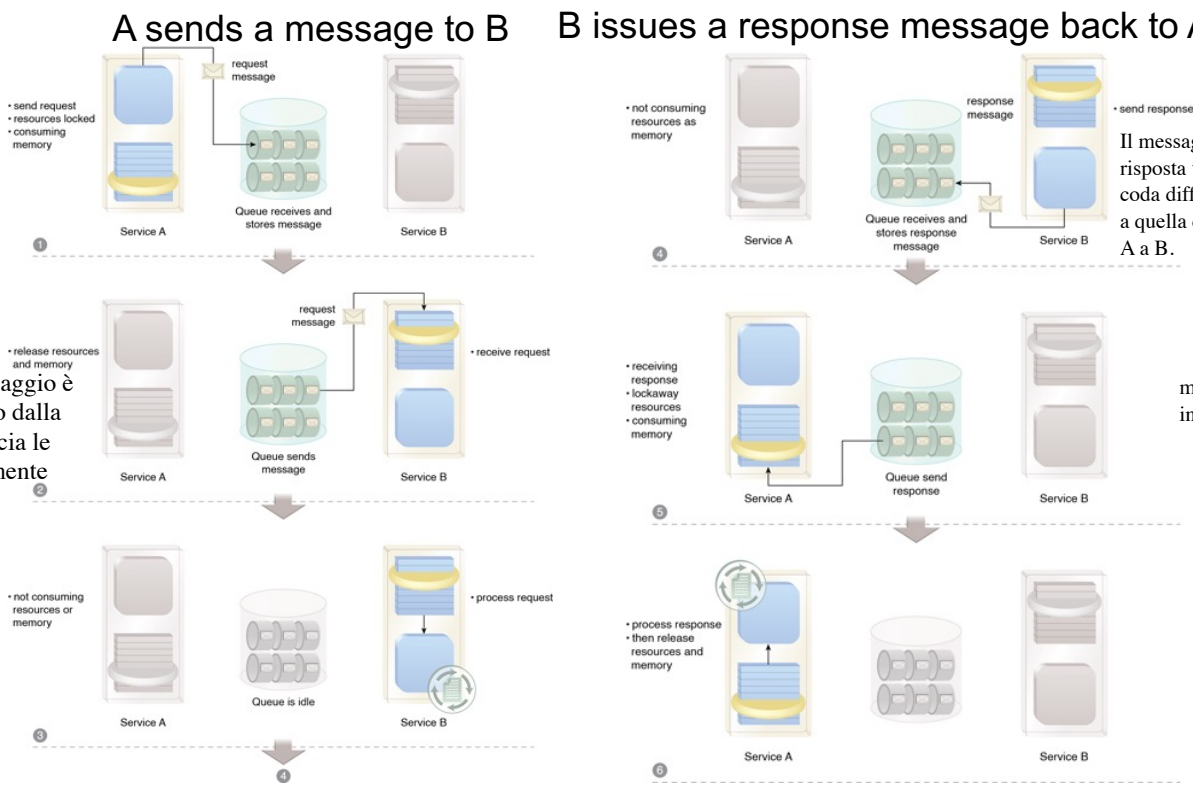
A Produce, B consuma.

Queue message pattern

A manda un messaggio a B, per farlo consumerà risorse e farà locking di aree di memoria. Può anche scollegarsi A, perchè msg resta in coda.

Una volta che messaggio è stato preso in carico dalla coda producer rilascia le risorse precedentemente acquisite.

Consumer prende messaggio, processa e manda risposta.



Il messaggio di risposta viaggia su una coda differente rispetto a quella del messaggio da A a B.

msg memorizzato in coda.

Message queue API

- Basic interface to a queue in a MQS:
  - **put**: nonblocking send
    - Append a message to a specified queue
  - **get**: blocking receive
    - Block until the specified queue is nonempty and remove the first message

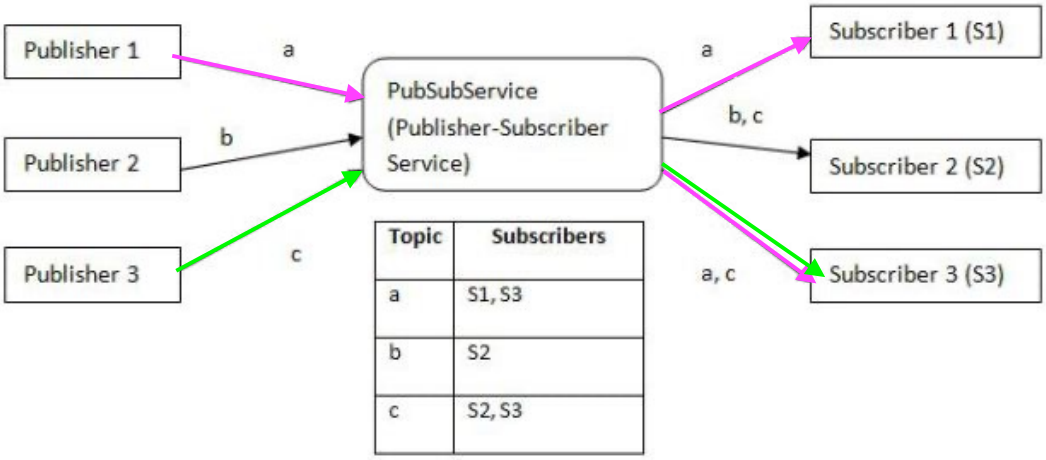
      ricezione bloccante: finchè la coda è vuota io sono bloccato (niente da leggere), appena arriva qualcosa (coda nonempty) "Leggo" il primo msg.
    - Variations: allow searching for a specific message in the queue, e.g., using a matching pattern (get con keyword, non per forza prelevo il prossimo)
  - **poll**: nonblocking receive
    - Check a specified queue for message and remove the first
    - Never block

      viene ricercato messaggio nella coda e viene rimosso o il primo in assoluto o il primo che matcha il pattern di ricerca.
  - **notify**: nonblocking receive (comunicazione asincrona)
    - Install a handler (callback function) to be automatically called when a message is put into the specified queue

Messaggio inviabile uno-a-molti, componenti possono pubblicare messaggi asincroni o essere interessati a ricevere dei messaggi facendo subscription. Subscriber possono essere interessati a topic con o senza filtri, le sottoscrizioni sono raccolte da un eventi dispatcher, responsabile di mandare tutti gli eventi ai subscriber interessati. Alto grado di disaccoppiamento tra i componenti. Qui è 1->N, prima 1-1.

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



Publish/subscribe pattern

- Multiple consumers can subscribe to topic with or without filters Topic: "partite serieA" filtro: "partite AS ROMA"
- 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

Questa è la differenza che abbiamo già evidenziato prima: 1-1 vs 1-N

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

possibile specificare: filtri per gli eventi, scadenza per iscrizione

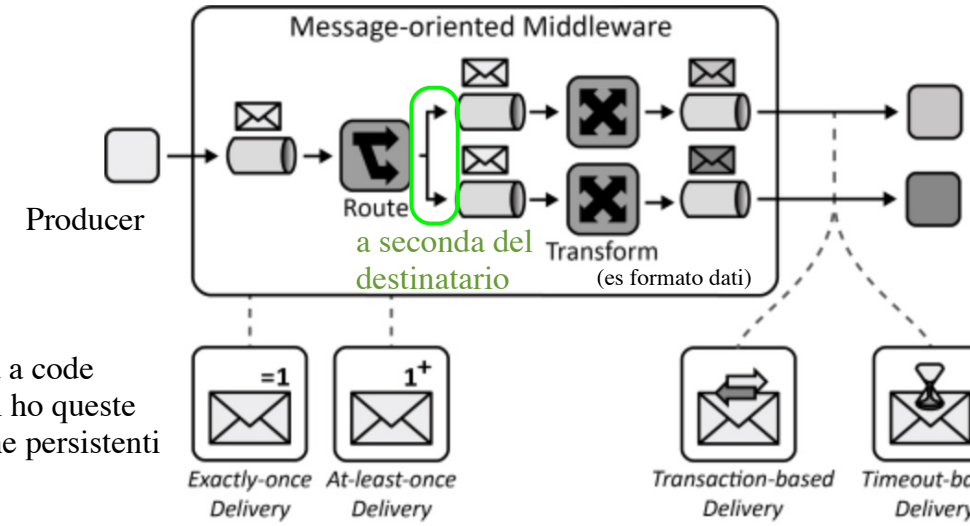
  - unsubscribe(sub handle)
  - notify_cb(sub_handle, event): called by the pub/sub system to deliver a matching event

manda notifiche per un evento corrispondente ai filtri specificati dai subscribe per un determinato evento.

Sia la "coda" che "Publish/Subscribe" necessitano di una componente di routing, la quale deve occuparsi di instradamento ed aspetti più "tecnici".

MOM functionalities

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



Nel sistema a code di messaggi ho queste due politiche persistenti

Qui ho due casi "più specifici". Sono tutte semantiche.

Source: Cloud Computing Patterns
https://www.cloudcomputingpatterns.org/message_oriented_middleware

NB: Queste 4 politiche sono "DELIVERY", ovvero consegnate a livello MIDDLEWARE. Quando parlo di CONSEGNA, invece, intendo la consegna al DESTINATARIO.

MOM functionalities

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

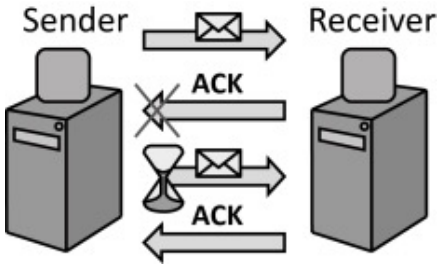
Delivery semantics in MOM

At-least-once delivery

How can MOM ensure that messages are received successfully? Almeno una ricezione al Receiver, quindi posso riceverlo più volte.

- By sending ack for each retrieved message and resending message if ack is not received
- Be careful, app should be tolerant to message duplications, i.e., it should be *idempotent* (not be affected adversely when processing the same message more than once)

Uso ACK: se non ricevo ACK, il sender lo re-invia dopo un tot di tempo.

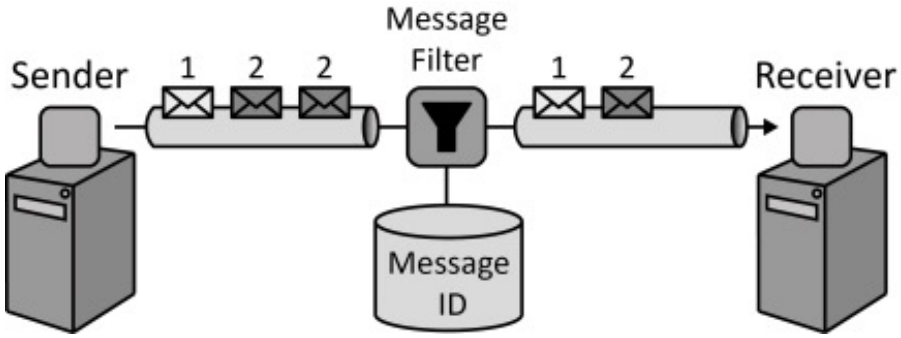


Delivery semantics in MOM

Exactly-once delivery

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

- By filtering possible message duplicates automatically (da solo non basta)
- Upon creation, each message is associated with a unique message ID, which is used to filter message duplicates during their traversal from sender to receiver
- Messages must also survive MOM components' failures



Incapsulo transazione ACID nel messaggio: quando receiver ha ricevuto messaggio, invia "ack" alla coda di messaggi e solo alla ricezione dell'"ack" il messaggio viene cancellato dalla coda.

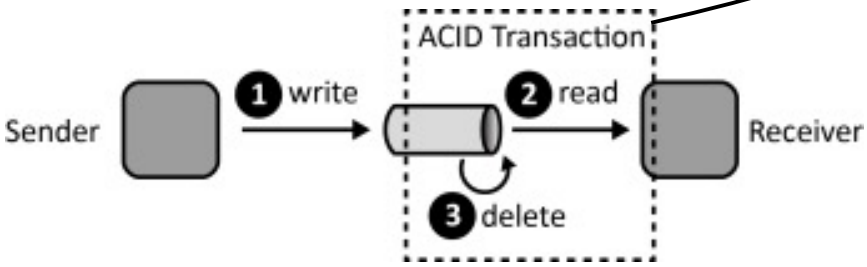
Delivery semantics in MOM

Transaction-based delivery

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

- MOM and the receiver participate in a transaction: all operations involved in the reception of a message are performed under one transactional context guaranteeing ACID behavior

Non ho garanzie su QUANTE VOLTE RICEVO, qui dico solo che viene cancellato quando è ricevuto correttamente.

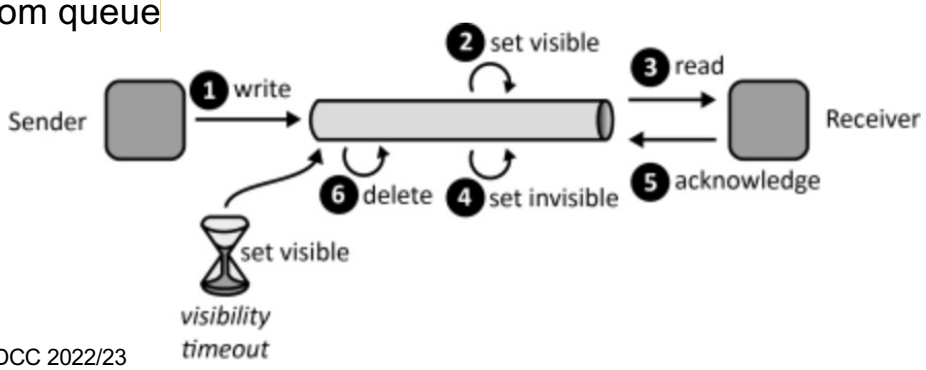


Delivery semantics in MOM

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

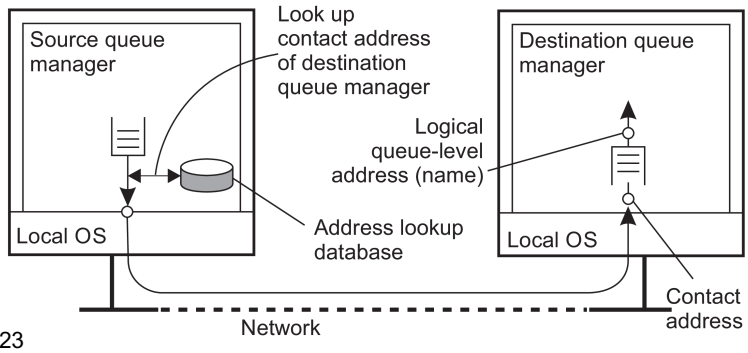


Routing avviene su diverse code, ciascuna gestita da un manager delle code, supporta l'idea di overlay network formata dai gestori delle code.

Message routing: general model

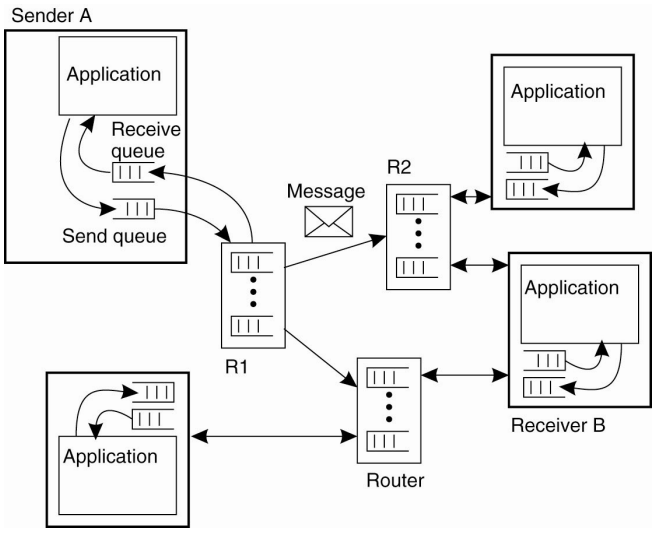
- Queues are managed by queue managers (QMs)
  - An application can put messages only into a local queue
  - Getting a message is possible by extracting it from a local queue only
- QMs need to route messages
  - Work as message-queuing "relays" that interact with distributed applications and each other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

viene usata per fare il routing dei messaggi, usando routing table che sono gestite e salvate dai queue manager



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



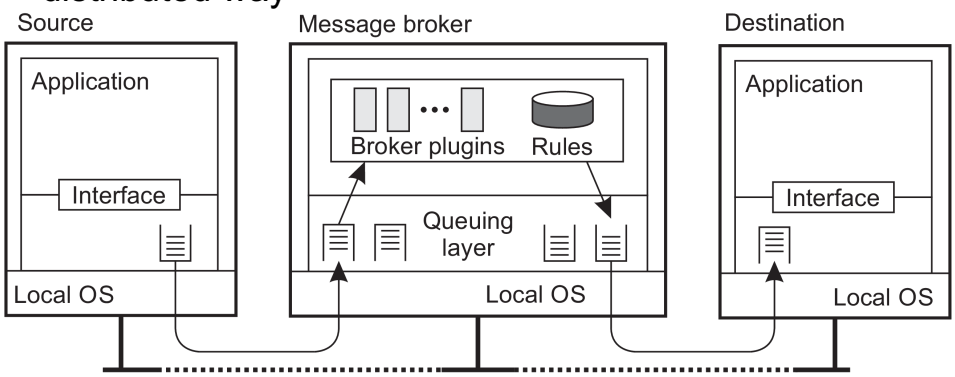
Message transformation: message broker

- 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

Non esaminato in dettaglio.

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

- Examples of MOM systems and libraries
  - Apache ActiveMQ http://activemq.apache.org
  - **Apache Kafka** Sfrutta i LOG)
  - Apache Pulsar https://pulsar.apache.org
  - IBM MQ MQ = Message Queue
  - NATS https://nats.io
  - Open MQ (JMS specification implementation)
  - **RabbitMQ** https://www.rabbitmq.com L'approccio più "semplice"
  - ZeroMQ https://zeromq.org
- Clear distinction between queue message and pub/sub patterns often lacks
  - Some frameworks support both (e.g., Kafka, NATS)
  - Others not (e.g., Redis is pub/sub https://redis.io/topics/pubsub)

MOM frameworks

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

16/11/2022

Più usate per code di msg su singola nodo.

RabbitMQ

- Popular open-source message broker Framework li gestisce.
  https://www.rabbitmq.com

gestore di 'n' code di msg istanziate su stesso server.

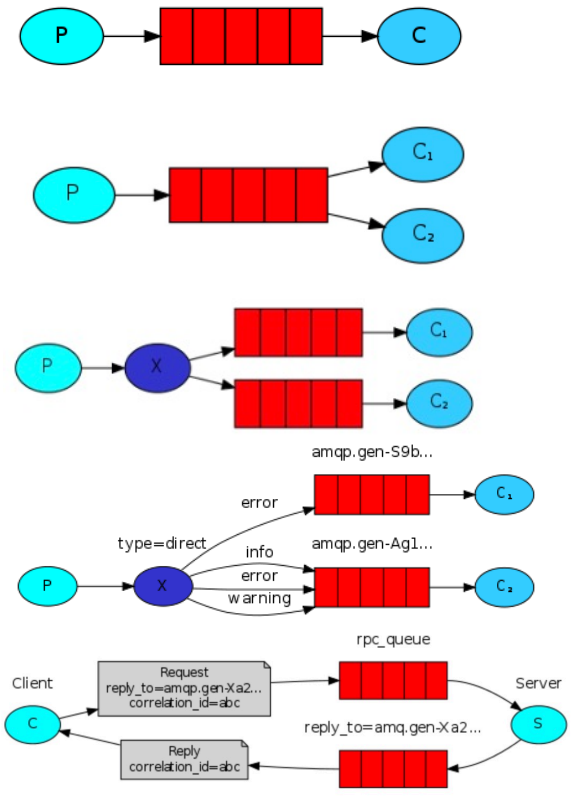
- Supports multiple messaging protocols
  - AMQP, STOMP and MQTT
- FIFO ordering guarantees at queue level
- Installation https://www.rabbitmq.com/download.html
- RabbitMQ CLI tool: `rabbitmqctl`

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

- Also web UI for management and monitoring
- RabbitMQ broker can be distributed, for example forming a cluster https://www.rabbitmq.com/distributed.html

Using message queues: 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)



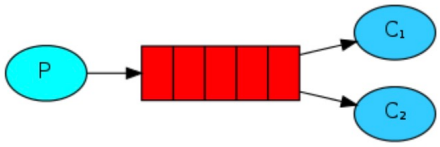
Source: RabbitMQ tutorial http://bit.ly/2zPPMJO

Using message queues: RabbitMQ and Go

- Let's use RabbitMQ, Go and AMQP (messaging protocol, see next slides) to use a message queue for:

1. Message queue pattern Classico, persistente.
   https://www.rabbitmq.com/tutorials/tutorial-one-go.html

2. Competing consumers pattern Molteplici worker, processano il task.
   https://www.rabbitmq.com/tutorials/tutorial-two-go.html



Code available on course site:
rabbitmq_1_hello.zip
rabbitmq_2_worker.zip

Using message queues: RabbitMQ and Go

- Preliminary steps:

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

```
$ rabbitmq-server
```

2. Install Go AMQP client library

```
$ go get github.com/streadway/amqp
```

See https://godoc.org/github.com/streadway/amqp for details on Go package `ampq`

Vediamo l'esempio 2

Using message queues: RabbitMQ and Go

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

- Version 1 (new_task_v1.go and worker_v1.go):
  - Use multiple consumers to see how the queue can be used to distribute tasks among them in a round-robin fashion
  - If the consumer crashes after RabbitMQ server delivers the message but before completing the task, the corresponding message is lost (i.e., cannot be delivered to another consumer)
    - `auto-ack = true`: a 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):
  - Change auto-ack in `Consume` from true to false and add explicit ("manual") ack in consumer to tell RabbitMQ server that a particular message has been received, processed and that RabbitMQ can discard it
  - Shutdown and restart RabbitMQ server: what happens to pending messages in the queue?

Using message queues: RabbitMQ and Go

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

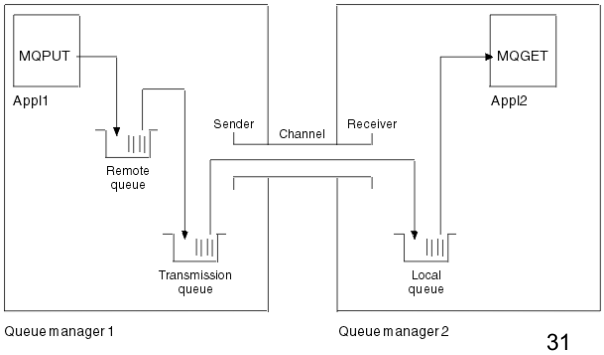
- Version 3 (new_task_v3.go and worker_v3.go):
  - Let's use a durable queue so it is persisted to disk and survives a node restart
  - Change `durable` in `QueueDeclare` from true to false; you also need to use a new queue
- Version 4 (new_task_v3.go and worker_v4.go):
  - Improve task distribution among multiple consumers by looking at the number of unacknowledged messages for each consumer, so to not dispatch a new message to a worker until it has processed and acknowledged the previous one
  - Use channel prefetch setting (`Qos`)

NON VISTA.

IBM MQ

https://www.ibm.com/products/mq

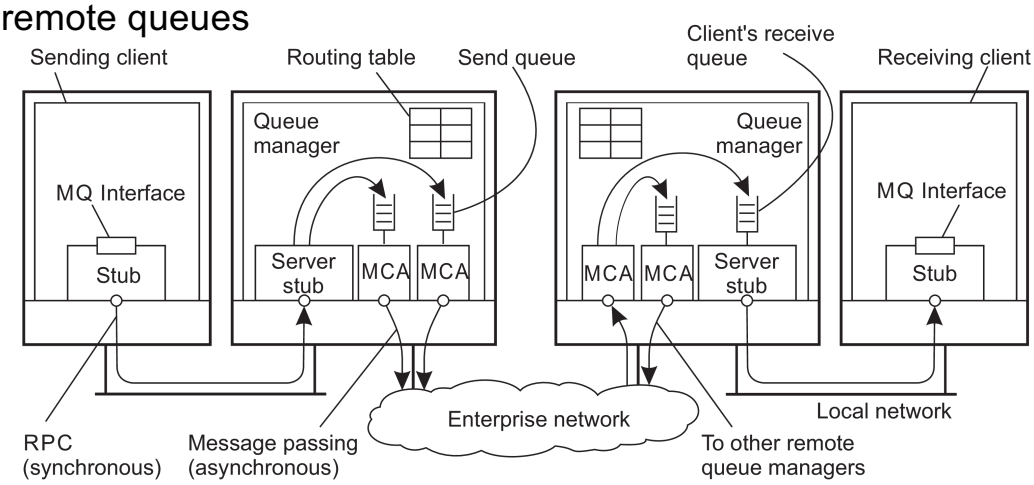
- The first enterprise messaging technology, from 1993
- Basic concepts:
  - App-specific messages are put into and removed from queues
  - Queues reside under the regime of a queue manager (QM)
  - Processes can put messages only in local queues, or in remote queues through an RPC mechanism
- Message transfer
  - Messages are transferred between queues
  - Message transfer between queues requires a channel
  - At each endpoint of channel is a message channel agent (MCA)
- MCAs are responsible for:
  - Setting up channels
  - Sending/receiving messages
  - Encrypting messages



IBM MQ

- Principles of operation:
  - Channels are inherently unidirectional
  - Automatically start MCAs when messages arrive
  - Any network of queue managers can be created
  - Routes are set up manually (system administration)
  - Routing: by using logical names, in combination with name resolution to local queues, it is possible to route message to remote queues

Testo



Amazon Simple Queue Service (SQS)

- Cloud-based message queue service based on polling model
  - Goal: decouple Cloud app components
  - Message queues are fully managed by AWS
  - Messages are stored in queues for a limited period of time
- Application components using SQS can run independently and asynchronously and be developed with different technologies
- Provides timeout-based delivery
  - Messages are only deleted from a message queue if they have been received properly
  - A received message is locked during processing (*visibility timeout*); if processing fails, the lock expires and the message is available again
- Can be combined with Amazon SNS
  - To push a message to multiple SQS queues in parallel

Amazon SQS: API

- **CreateQueue, ListQueues, DeleteQueue**
  - Create, list, delete queues
- **SendMessage, ReceiveMessage**
  - Add/receive messages to/from a specified queue (message size up to 256 KB)
  - Larger message?
    - Put in queue reference to message payload stored in S3

Posso mandare un riferimento all'oggetto, come il link/url di un oggetto presente su server.

- **DeleteMessage**
  - Remove a received message from a specified queue (the component must delete the message after receiving and processing it)

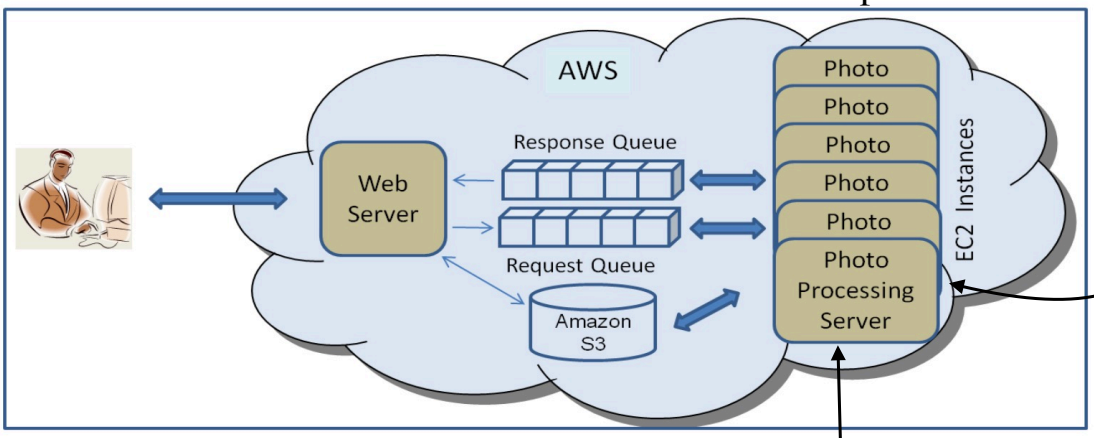
Amazon SQS: API

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

Amazon SQS: example

- Cloud app for online photo processing service
- Let's use SQS to achieve app components decoupling, load balancing and fault tolerance http://bit.ly/2gwJFBw

Due code unidirezionali: richiesta e risposta.



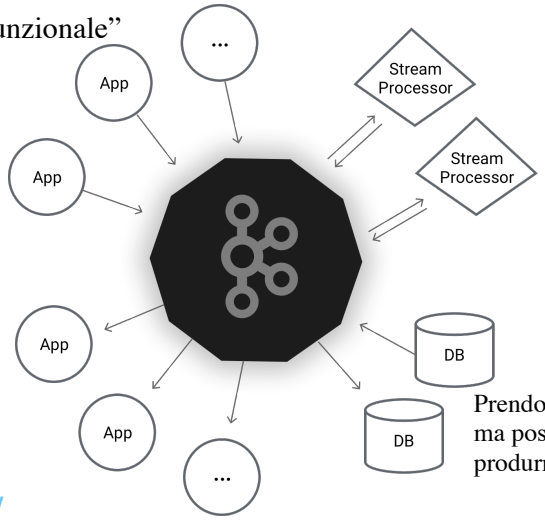
Eseguono la computazione

Se "visibilità" è "ben settata", quando il msg ritorna visibile è probabile che venga preso da un'altra delle copie (l'importante è prenderlo).

Ho varie repliche per aumentare la scalabilità, il funzionamento non viene influenzato da ciò.

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 Linguaggio di tipo "funzionale"
- Horizontally scalable
- Fault-tolerant
- High throughput ingestion
  - Billions of messages
- Not only messaging, also processing of data
  - We focus on messaging



Prendo dati dal DB. ma posso anche produrre dati nel DB

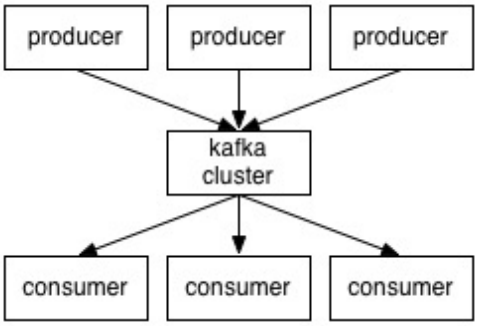
https://kafka.apache.org/documentation/

Kreps et al., Kafka: A Distributed Messaging System for Log Processing, NetDB'11

E' anche collegabile ad altri framework.

Da un punto di vista architetturale, Kafka è composto da un cluster di broker, che mettono a disposizione un log distribuito per la gestione dei dati. Espone a producer/consumer due topic: un topic può avere 1 producer che pubblica su un topic e 0-1-N consumer che si sono sottoscritti al topic e ricevono messaggi da quel topic; i broker gestiscono i topic.

Kafka at a glance



- Kafka maintains feeds of messages in categories called topics (categorie)
  - 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 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

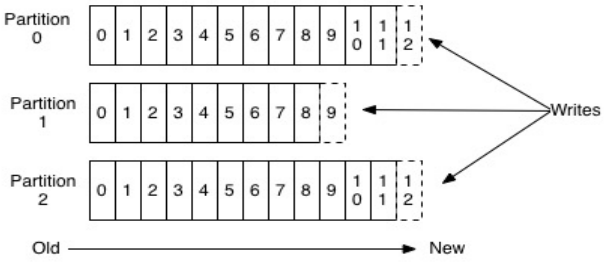
Kafka: topics and partitions

- Topic: category to which the message is published
- For each topic, Kafka cluster maintains a partitioned log
  - **Log** (data structure!): *append-only*, *totally-ordered* sequence of records ordered by time

Devo pensare ai LOG come 'struttura dati', non come file.

- Topic split into a pre-defined number of partitions
  - Partition: unit of parallelism of the topic (allows for parallel access)
- Each partition is replicated with some replication factor

Sennò potrei perdere le singole partizioni



I log mantengono serie di record ordinati (solo su singola partizione) rispetto al tempo.

Il singolo topic è memorizzato da singolo Log.

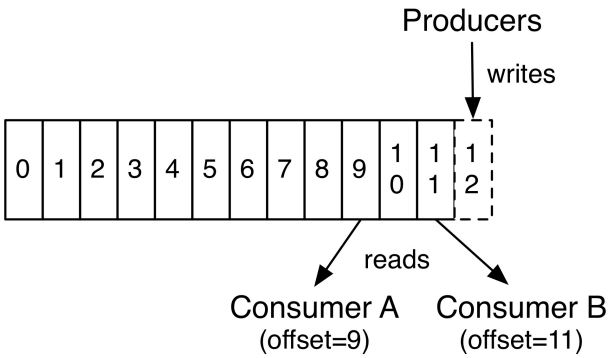
Nelle partizioni ho record, in essi ho gli ID univoci per ogni partizione! Più precisamente sono offset.

Kafka: partitions

- Producers publish their records to partitions of a topic (round-robin or partitioned by keys), and consumers consume published records of that 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

Partizione è sequenza di record ordinata temporalmente rispetto alla scrittura (solo append). Il log è immutabile, poichè posso aggiungere solo in coda, ma non li posso modificare. Posso leggere in parallelo su due punti diversi (es ID 3 ed ID 7)

Kafka divide ciascun topic in partizione



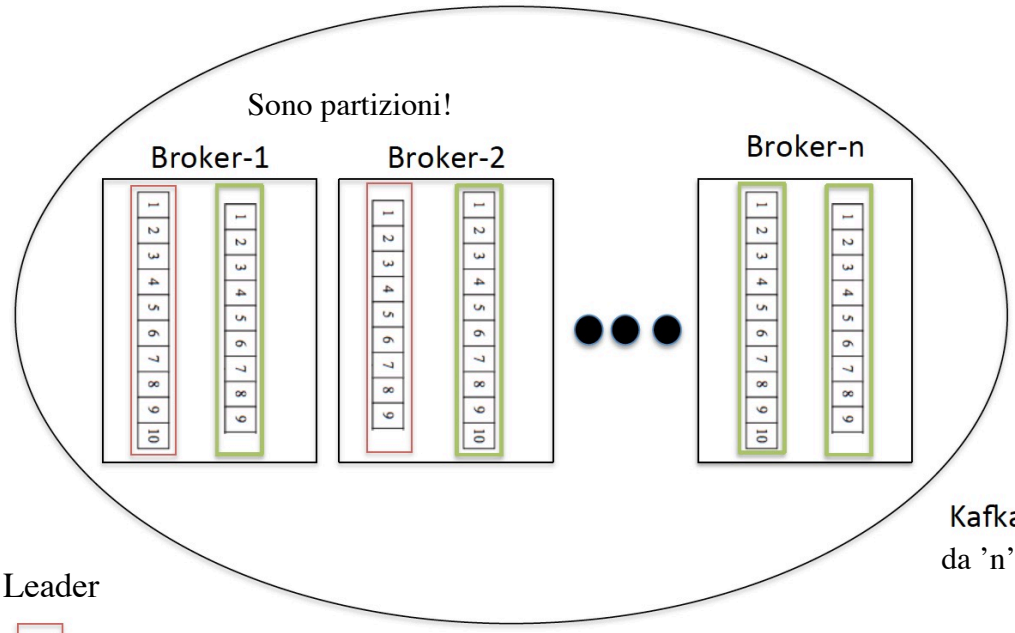
Kafka: partitions

- Partitions are distributed across brokers for scalability
- Each partition is replicated for fault tolerance across a configurable number of brokers
- Each partition has one leader broker and 0 or more followers
- The leader handles read and write requests
  - Read from leader
  - Write to leader
- A follower replicates the leader and acts as a backup
- Each broker is a leader for some of its partitions and a follower for others to load balance (condivisione responsabilità)
  - Brokers rely on Apache Zookeeper for coordination

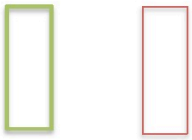
Solo se crasha si passa ad altro leader.
Non esiste un Broker leader di tutto quanto (influencer)

- Versioni differenti dello stesso topic sono gestite da broker diversi.

Kafka: partitions



Follower Leader



Broker 2... broker n sono in sync con la partizione del leader.
In caso di crash del leader, se broker-2 non ha alcune partizioni, tale partizione non è totalmente sincronizzata, non permette di ricostruire la partizione, e non è candidabile a leader.

Kafka: producers

18/11/2022

- Producers = data sources
- Publish data to topics of their choice (topic deve essere specificato)
  - Producer sends data directly (i.e., without any routing tier) to the broker that is the leader for the partition
- Also responsible for choosing which record to assign to which partition within the topic (devo scegliere anche la partizione, di solito round robin)
  - Random or key-based partitioned
  - E.g., if user id is the key, all data for a given user will be sent to the same partition
- Producers can write to the same partition
  == Multiple producers ma i record scritti in "modo ordinato",in base ad offset.

Kafka: 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 send
- **Pull** model
  - Consumer assumes the primary responsibility for retrieving messages from broker
  - Consumer has to maintain an offset that identifies the next message to be transmitted and processed
  - Pros: better scalability (less burden on brokers) and flexibility (different consumers with diverse needs and capabilities)
  - Cons: in case broker has no data, consumers may end up busy waiting for data to arrive

contro:

broker deve adattarsi ai "diversi tipi di dato", e scegliere se inviare subito o aspettare che un buffer si riempa.

Consumer responsabile del recupero dei dati.

Kafka: consumers

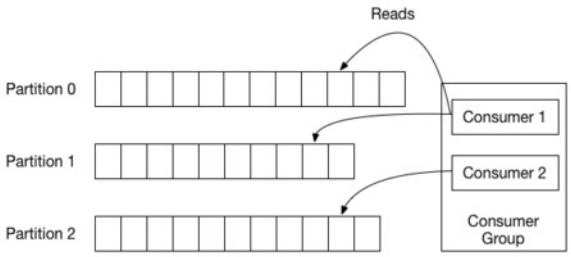
- In Kafka design, **pull** approach for consumers
  http://kafka.apache.org/documentation.html#design_pull
- Consumers use the offset to track which messages have been consumed
  - Messages can be replayed using the offset

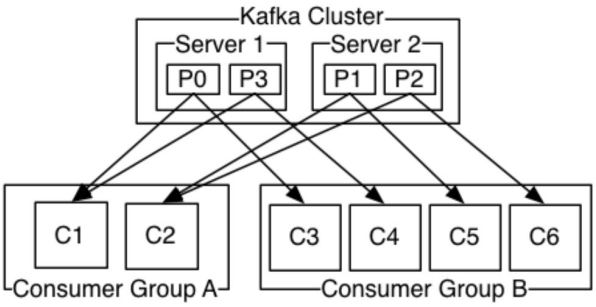
Per ridurre il carico di lavoro sui broker. Inoltre, in questo modo, è il consumer che si adatta ai dati, no i broker. L'unico svantaggio è che il consumer potrebbe rimanere in busy-waiting, cioè cerca continuamente messaggi che potrebbero non arrivare.

Inoltre, quando consumer legge messaggio, salva il suo offset in un topic speciale chiamato __consumer_offset, un topic instanziato automaticamente da Kafka, in cui i consumer salvato i valori dell'offset. Utile per tolleranza ai guasti, poichè se perdo l'offset in memory, posso recuperarlo da questo topic.
Il retrieve da parte del consumer può essere fatto dall'inizio, oppure da un certo offset.

Kafka: consumer group

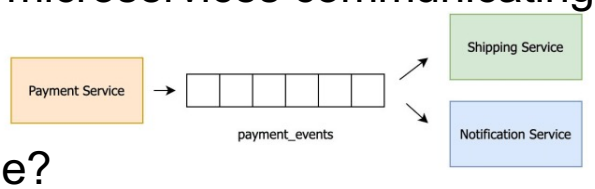
- Consumer Group: set of consumers which cooperate to consume data from some topics and share a group ID
  - A Consumer Group maps to a *logical* subscriber
  - Partitions are divided among consumers in the group for load balancing and can be reassigned in case of consumer join/leave
  - Every message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





Kafka: consumer group

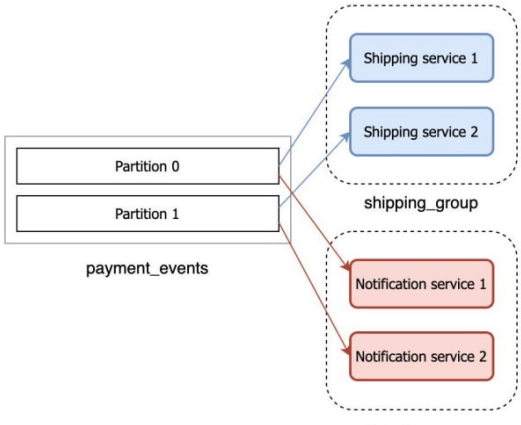
- How to have many consumers reading the same messages from the topic?
  - Using different group IDs
- Example: 2 microservices communicating using Kafka



- How to scale?

Queste due repliche mi permettono di leggere dalle partizioni. Un messaggio P0 può essere letto sia da Shipping1 che Notification1.
Se avessi creato un gruppo: Shipping1,Notification1?

NON POTEVO, due elementi di un gruppo NON POSSONO LEGGERE da stessa partizione.



Ho sequenza di dati, partizionata, cioè suddivisa in diverse partizioni.
Queste partizoni possono essere viste come "sub-topic".

Kafka: ordering guarantees

- Messages sent by a producer to a topic partition will be appended in the order they are sent
- Consumer sees records in the order they are stored in the partition
- Strong guarantees about ordering only within a partition
  Ordinamento garantito solo all'interno della partizione, non totale.
  Nel topic non ho questa garanzia. (troppo oneroso!)
  - Total order over messages within a partition, i.e., *per-partition ordering*
  - But Kafka cannot preserve message order between different partitions in a topic
- However per-partition ordering plus ability to partition data by key among topic partitions, is sufficient for most applications

Tuttavia producer possono scrivere in una partizione usando chiave, quindi il limite dell'ordinamento totale è meno grave del previsto.

Kafka: delivery semantics

- Delivery guarantees supported by Kafka
  - **At-least-once** (default): guarantees no message loss but duplicated messages, possibly out-of-order (se leggo da partizioni differenti)
  - **Exactly-once**: guarantees no loss and no duplicates, but requires expensive end-to-end 2PC two-face-commit
    - But not fully exactly-once NON è proprio la Exactly once che conosciamo, ma ci si avvicina.
    - Support depends on destination system Kafka garantisce, ma necessita di più risorse, con thoughtput R/W ridotto.
- User can also implement at-most-once delivery by disabling retries on producer and committing offsets on consumer prior to processing a message

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

Scrittura avviene sul leader, che chiede di effettuare scrittura su followers, aspetta ack prima di rendere msg visibile ai consumer (i quali leggono solo dal leader). Ciò permette distinzione tra partizioni leader e partizioni follower. Tuttavia msg scritto dal producer potrebbe non essere subito disponibile dai consumer, ma solo dopo che è stato replicato. E' un classico tradeoff.

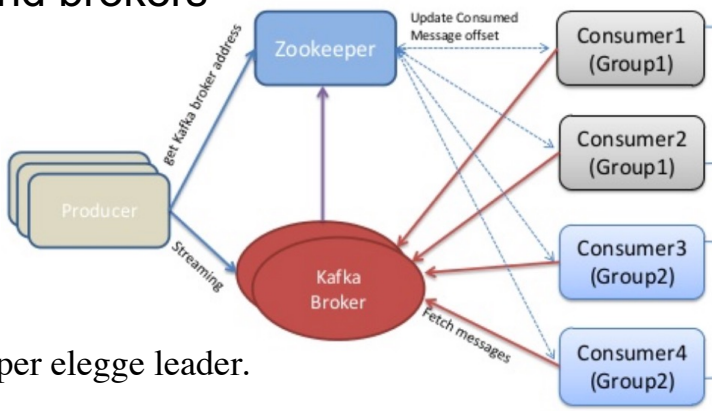
Kafka: fault tolerance

- Kafka replicates partitions for fault tolerance
- Kafka makes a message available for consumption only after all the followers acknowledge to the leader a successful write
  - Implies that a message may not be immediately available for consumption (tradeoff between consistency and availability)
  - This default behavior can be relaxed if such strong guarantee is not required
- Kafka retains messages for a configured period of time
  - Messages can be "replayed" in case a consumer fails

Kafka mantiene per un certo tempo i msg nelle partizioni.
Implementa meccanismo di "compattazione" dei msg nella partizione, sennò crescerebbe a dismisura lo spazio di memorizzazione. Ad un certo punto i msg vengono rimossi. Ciò permette al consumer di fare replay di messaggi già visti.

Kafka and ZooKeeper (progetto Opensource by Apache)

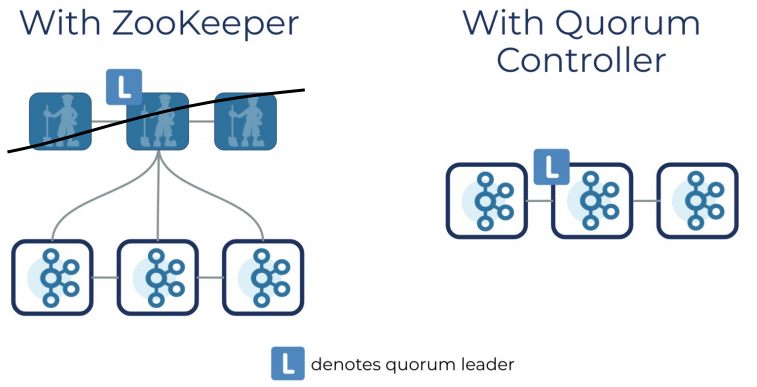
- Zookeeper: hierarchical, distributed key-value store (devo esplorare albero)
  - Coordination and synchronization service for large distributed systems
  - Often used for leader election
  - Used within many open-source distributed systems besides Kafka (Apache Mesos, Storm, …)
- Kafka uses ZooKeeper to coordinate between producers, consumers and brokers
- ZooKeeper stores Kafka metadata
  - List of brokers
  - List of consumers and their offsets
  - List of producers



Zookeeper elegge leader.

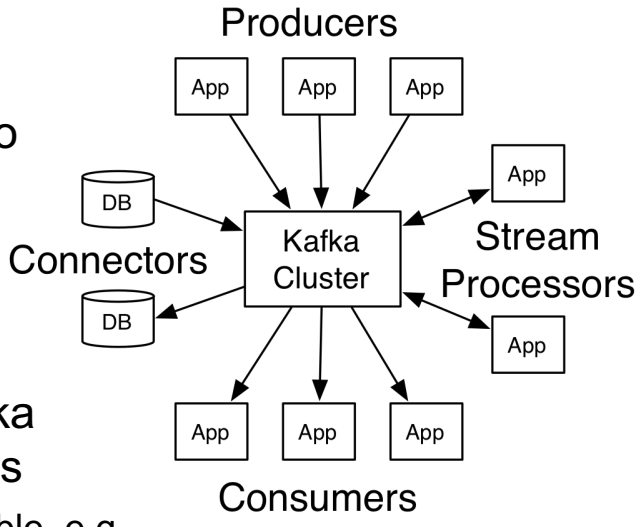
KRaft

- Kafka and Zookeeper: a different system for metadata management and consensus
- New release can use Apache Kafka Raft (KRaft)
  - Consensus protocol to allow metadata management directly in Kafka Protocollo di consenso
  - Event-based variant of Raft



Kafka: APIs

- Four core APIs https://kafka.apache.org/documentation/#api
- Producer API: allows apps to publish records to topics
- Consumer API: allows apps to read records from topics
- Connect API: allows implementing reusable connectors (producers or consumers) that connect Kafka topics to apps or data systems
  - Many connectors already available, e.g., for AWS S3, ActiveMQ, IBM MQ, RabbitMQ, MySQL, Postgres, AWS Lambda



Connect si usa poco, ne esistono già altri.

Kafka: APIs

- Streams API: allows transforming streams of data from input topics to output topics
  - Kafka is not only a pub/sub system but also a real-time streaming platform
    - Use Kafka Streams to process data in pipelines consisting of multiple stages
- Kafka APIs support Java and Scala only

Kafka: client library

- JVM internal client
- Plus rich ecosystem of client library, among which:
  - Go
    https://github.com/Shopify/sarama
    https://github.com/segmentio/kafka-go
  - Python
    https://github.com/confluentinc/confluent-kafka-python/
  - NodeJS
    https://github.com/Blizzard/node-rdkafka

Kafka and Python client library

- Preliminary steps
  1. Install, configure and start Kafka and Zookeeper
  2. Install Python client library https://pypi.org/project/kafka-python/

```
$ pip install kafka-python
```

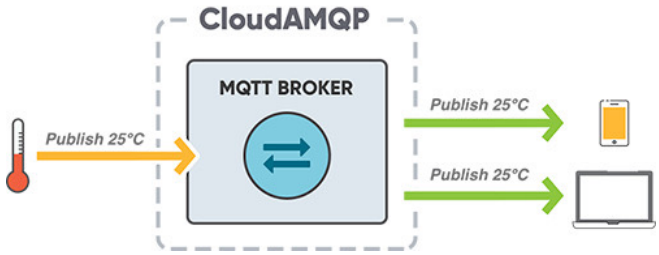
- See `kafka-python_example.py` on course site

Protocols for MOM

- 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 (minore overhead rispetto AMQP)
  - 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

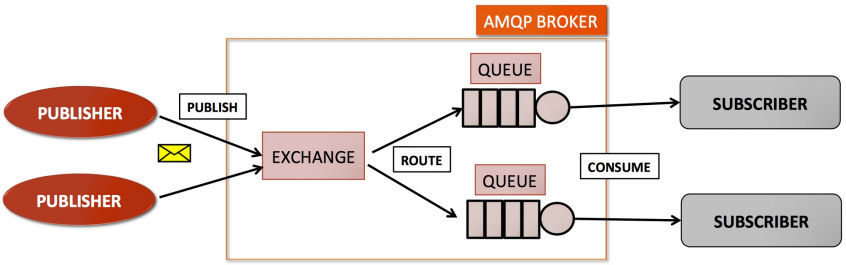
AMQP: characteristics

- Open-standard protocol for MOM, supported by industry standardizzato nel 2014.
  - Current version: 1.0 http://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 (at-most once, at-least once, exactly once delivery)
- Programmable protocol Diverse entità/schemi di routing configurabili
  - Several entities and routing schemes are primarily defined by apps
- Implementations
  - Apache ActiveMQ, RabbitMQ, Apache Qpid, Azure Event Hubs, Pika (Python implementation), …

Qui ho publisher e subscriber, in mezzo il broker, che implementa AMQP. Al suo interno ho 'EXCHANGE' e le code, i messaggi, quando prodotti, vengono pubblicati in exchange (tipo mailbox), che poi vengono instradate nelle code mediante meccanismi configurabili. I messaggi poi possono essere usabili da subscriber in modalità di pull o push.

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

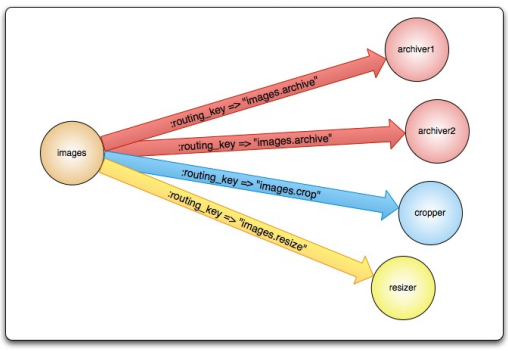
https://bit.ly/2oP683F

AMQP: routing

- Bindings:
  - Direct exchange: delivers messages to queues based on message routing key

    Sulla base di chiave assegnato al msg, smisto verso code. (queue based)

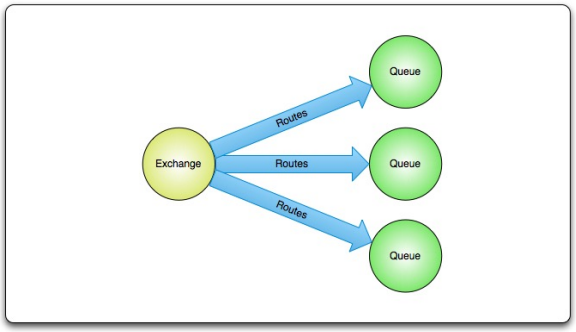
Direct exchange routing



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

    Simil broadcast, su tutte le code possibili.

Fanout exchange routing



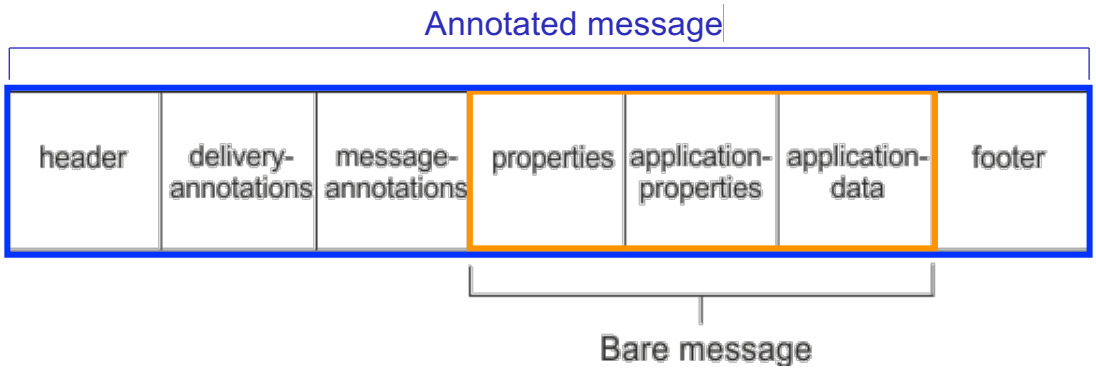
AMQP: routing

- Bindings:
  - Topic Exchange: delivers messages to one or many queues based on topic matching
    - Often used to implement various publish/subscribe pattern variations invio a coda specifica, in base al topic. Usato per multicast.
    - 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 a routing key

      Più flessibili rispetto "direct routing", perchè qui posso specificare diversi parametri del leader del messaggio.

AMQP: messages

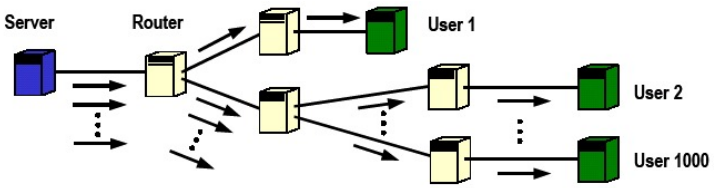
- AMQP defines two types of messages:
  - **Bare messages**, supplied by sender
  - **Annotated messages**, seen at receiver and added by intermediaries during transit
- Message header conveys delivery parameters
  - Including durability requirements, priority, time to live



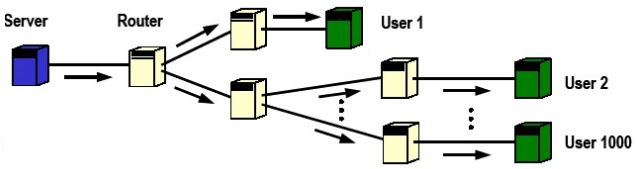
Multicast communication

- **Multicast communication**: communication pattern in which data is sent to *multiple* receivers at once
  - Broadcast communication*:* special case of multicast, in which data are sent to all the possible 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

- Traditional unicast communication does not scale



Unicast of video to 1000 users



Multicast of video to 1000 users

Sia a destra che a sinistra devo contattare 1000 utenti, la differenza è che tramite UNICAST parlo già con 1000 copie del messaggio, in MULTICAST parto da un messaggio, il quale si sdoppia quando trova una ramificazione.

Types of multicast

- How to realize multicast?
  - **Network**-level multicast
  - **Application**-level multicast

A livello network ho "IP Multicast IPMC", generalizzazione di UDP, con un comportamento 1-N poichè si basa sui gruppi. Infatti devo definire il gruppo di multicast, identificato tramite questo IPMC unico per tutti i membri del gruppo. Gruppi formati da host "interessati" a condividere dati sfruttando il multicast. Tramite Internet Group Management protocol ci si unisce a questi gruppi.

Network-level multicast

- Packet replication and routing managed by routers
- IP Multicast (IPMC) based on groups
  - IPMC generalizes UDP with one-to-many behavior
  - Receivers use a special form of IP address that can be shared among multiple processes
  - *Group*: set of hosts interested in the same multicast app; they are identified by the same IP address
    - Group IP address from 224.0.0.0 to 239.255.255.255
  - IGMP (Internet Group Management Protocol) to join the group
- Limited use:

Il supporto a livello di rete "multicast" è limitato poichè soggetto "broadcast storm", che satura la rete.

  - Lack of large-scale support (only ~5% of ASs)
  - Difficult to keep track of group membership
  - Banned in some contexts, e.g., in Cloud data centers because of *broadcast storm* problem (exponential increase of network traffic with possible saturation)

Application-level multicast

- Packet replication and routing managed by end hosts
- Basic idea:
  - Organize nodes into an overlay network
  - Use that overlay network to disseminate data
- Application-level multicast:

Le realizzo in due modi

  - *Structured*
    - Explicit communication paths

Già viste nei p2p

    - How to build a structured overlay network?
      - Tree: only one path between each node pair
      - Mesh: multiple paths between each node pair
  - *Unstructured*
    - Based on flooding
    - Based on gossiping

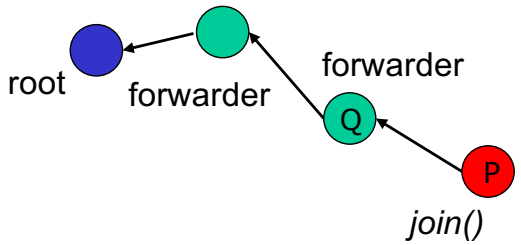
Nel modello "TREE" si fa uso di DHT per realizzare l'albero in multicast. Noi assuiamo che la DHT venga realizzata tramite Chord. Devo sfruttare meccanismo di routing fornito da overlay network.

Structured application-level multicast: tree

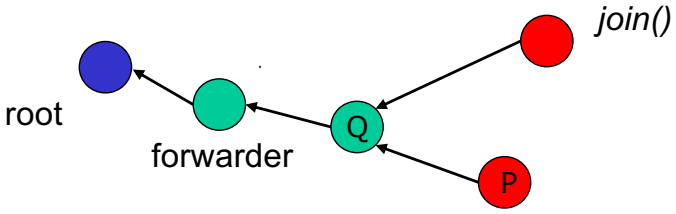
- Let's consider how to build the application-level multicast tree in Scribe
  - Scribe: pub/sub system with decentralized architecture and based on Pastry DHT
  1. Initiator node generates a multicast identifier *mid*
  2. Initiator lookups *succ*(*mid*) using DHT
  3. Request is routed to *succ*(*mid*), which becomes the root of the multicast tree
  4. If node *P* wants to join the multicast, it sends a join request to the root
  5. When request arrives at *Q*:
     - *Q* has not seen a join request for *mid* before ⇒ *Q* becomes forwarder, *P* becomes child of Q; join request continues to be forwarded by *Q*
     - *Q* knows about tree ⇒ *P* becomes child of *Q*; no need to forward join request anymore

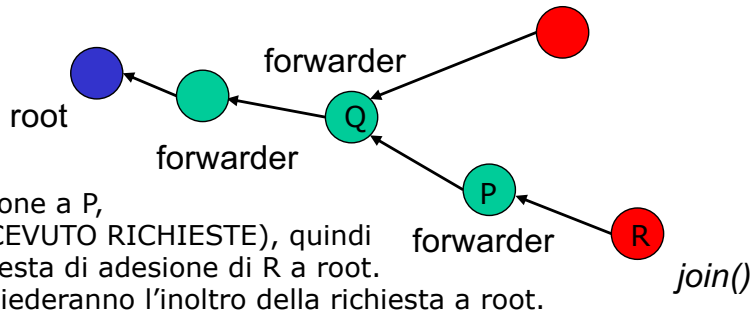
Castro et al., Scribe: A large-scale and decentralised application-level multicast infrastructure, *IEEE JSAC*, 2002

Structured application-level multicast: tree



In questo caso : Q è già forwarder.
Arriva P e si attacca a Q, richiesta non propagata, P è figlio di Q ma non è forwarder.





Qui arriva ad R, fa richiesta di adesione a P,
P non è forwarder (NON HA MAI RICEVUTO RICHIESTE), quindi P diventa forwarder e inoltra la richiesta di adesione di R a root.
Eventuali altre richieste a P non richiederanno l'inoltro della richiesta a root.

Non-structured application-level multicast

- How to realize non-structured application-level multicast?
  - Flooding: already studied
    - Node *P* sends the multicast message *m* to all its neighbors
    - In its turn, each neighbor will forward that message, except to *P*, and only if it had not seen *m* before
  - Random walk: already studied
    - With respect to flooding, *m* is sent only to one randomly chosen node
  - **Gossiping**

Inoltro, a oltranza, posso usare TTL per evitare loop.

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 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
- They allow information dissemination in large-scale networks through the random choice of successive receivers among those known to the 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

E' come se l'informazione fosse un pettegolezzo che si diffonde.

Sostanzialmente, come il gossip reale, seleziono tra i nodi del sistema, normalmente tra i vicini, e in modo randomico a qui inviare la mia informazione. Successivamente, ciascun nodo che l'ha ricevuta farà lo stesso.

Lo scopo originale era gestire un database replicato a larga scala, composto da qualche centinaio di nodi. Le repliche erano limitate, gli aggiornamenti non immediati (lazy). Questa "lentezza" mi porta a dire che prima o poi le informazioni diventeranno consistenti (informazione diffusa), ma non so quando!

The origin

- Gossiping protocols defined in 1987 by Demers et al. in a work about 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 on one or a few replicas
  - 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

Scelte casualmente, non progago a tutti, solo a qualcuno.

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 the overall system size
  - Reliability and robustness: thanks to message redundancy

Il servizio di storage S3 di Amazon dovrebbe ancora usare protocollo di Gossiping (informazione riservata), per diffondere informazioni in S3 in merito all'individuazione dei nodi che hanno fallimenti.
In S3 abbiamo tantissimi nodi coinvolti, e quando un nodo P scopre che un altro è down, il nodo P informa solo un sottoinsieme di nodi, NON TUTTI, a cui propagare l'informazione.

Who uses gossiping?

- 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" http://amzn.to/1MgDVsl
- 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 https://martinfowler.com/articles/patterns-of-distributed-systems/gossip-dissemination.html

Da quello che stiamo apprendendo, capiamo che il gossiping può avere numerosi casi d'uso, all'inizio è stato visto per aggiornare i dati (consistenza), S3 lo usa come failure detection [domanda d'esame].

Propagation models

- Let's consider the two principle operations
  - Pure gossiping and anti-entropy

- Pure gossiping (or rumor spreading): a node which has just been updated (i.e., has been contaminated), tells a number of other nodes about its update DICO IL MIO UPDATE (contaminating them as well)

- Anti-entropy: each node regularly chooses another node at random, and exchanges state differences, SCAMBIO UPDATE leading to identical states at both afterwards

Vediamo i due principi applicabili in un algoritmo di gossiping:
pure gossiping(gossiping semplice) e anti-entropy.
Nel gossiping semplice, un nodo che ha ricevuto una nuova informazione, comunica ad un certo numero di altri nodi il suo update (lo contamina).
Nell'anti entropia voglio andare "contro il disordine" (che per me è quando i messaggi si propagano).
Raggiungo bassa entropia quando tutti i nodi della rete hanno questo aggiornamento e sono consistenti.
Come la raggiungo? scambiando lo stato col nodo con cui sto comunicando (bidirezionale).

Suppongo che un nodo P abbia una informazione da diffondere. Tra i nodi che conosce, ne contatta randomicamente uno, supponiamo Q. Lo scegliere un solo nodo comporta un "fanout = 1". Q riceve il messaggio, definiamo allora la Prob(Q non propaga il messaggio perchè conosce già l'informazione) = 1/k
Altrimenti, Q contatta un altro nodo, similmente a come fatto da Q.

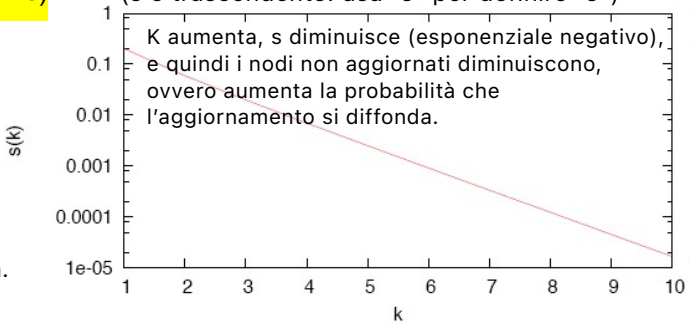
Pure gossiping

- A node *P* having an update to report, contacts another node Q randomly chosen
- If *Q* has already knows the update (it is already contaminated), Q loses interest in spreading the gossip and with probability $1/k$ stops contacting other nodes
- Otherwise, *P* contacts another (randomly selected) node
- If *s* is the fraction of ignorant nodes (i.e., which are unaware of the update), it can be shown that with many nodes

$s = e^{-(k+1)(1-s)}$

(s è trascendente: usa "s" per definire "s")

As k increases, the probability that the update will spread increases



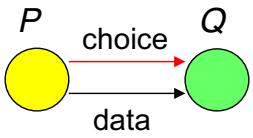
Sull'asse Y la scala è logaritmica.

Consider 10,000 nodes		
k	*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

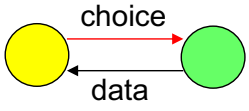
- If we really have to ensure that all nodes are eventually updated, pure gossiping alone is not enough, but we need to combine it with anti-entropy

Anti-entropy

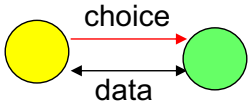
- Goal: increase node state similarity, thus decreasing the "disorder" (the reason for the name!) Come scambiano informazioni?
- A node *P* selects another node *Q* from the system at random; how does P update *Q*?
- Three different update strategies can be followed:



– **push**: *P* only pushes its own updates to *Q*
P sceglie Q e gli comunica le novità. Dati da P->Q.



– **pull**: *P* only pulls in new updates from *Q*
P contatta Q, e gli chiede le novità. Dati verso P <- Q

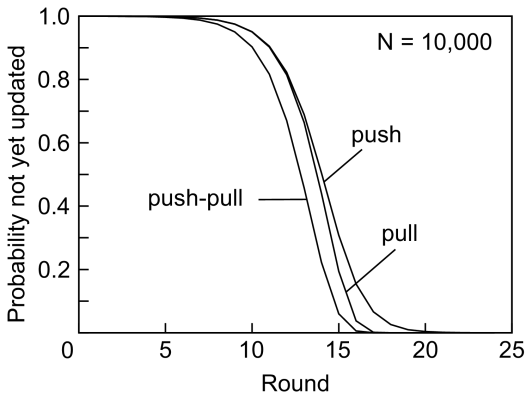


– **push-pull**: *P* and *Q* send updates to each other, i.e., *P* and *Q* exchange updates
Lo scambio dei dati è bidirezionale, questa è la strategia migliore perchè converge più velocemente verso un insieme di nodi aggiornato.

Definisco ROUND come l'intervallo di tempo in cui ogni peer ha preso almeno una volta l'iniziativa di scambiare aggiornamenti. Per N grande, O(log N) si comporta bene. Tale risultato è dato da gossiping + anti-entropia, perchè usando solo la strategia push-pull c'è bisogno di vari round.

Anti-entropy: performance

- Push-pull
  - The fastest strategy
  - Takes O(log(*N*)) rounds to disseminate updates to all *N* nodes
  - Round (or gossip cycle): time interval in which every node has taken the initiative to start an exchange



Le scelte in rosso dipendono dagli algoritmi di gossiping. Suppongo di avere nodi P e Q. P diffonde informazione verso Q. Nel punto (1), P seleziona Q mediante strategia SelectPeer. Esistono varie implementazioni, da una semplice scelta random all'introduzione di pesi per scegliere nodi che hanno meno compiti. Nel punto (2) scelgo cosa inviare, anche questo dipende dall'algoritmo di gossiping (possono servire dati diversi).

General organization of gossiping protocol

- Two nodes *P* e *Q*, where *P* has just selected *Q* to exchange information with; *P* is run once at each new 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*

In (3) invio da una parte e ricevo dall'altra. In (4), se strategia è PUSH-PULL, Q risponde, facendo quindi le stesse operazioni viste nei primi punti. Nello step (6), sia P che Q scelgono che dati mantenere dall'insieme di informazioni appena ottenute. Devono farlo perchè la cache in cui li mettono è limitata, e non è detto che tutti i dati ricevuti mi servano. P e Q concludono processandoli.

Framework of gossip-based protocol

- Simple? Not quite getting into the details…
- Some crucial aspects
  - Peer selection
    - E.g., *Q* can be chosen uniformly from the complete set of currently available (i.e., alive) nodes
  - Data exchanged
    - Exchange is highly application-dependent
  - Data processing
    - Again, highly application-dependent

Fino ad ora, abbiamo sorvolato sulla modalità con cui P contattasse Q. Non ci siamo chiesti:
- come P conosca Q.
- come "dar vantaggio" a nodi più vicini, migliorando le prestazioni.
- come seleziono le informazioni "più interessanti" per altri nodi.
- come gestisco le informazioni in cache (una cache diversa da quella che conosciamo, ma che comunque è uno spazio di memorizzazione temporanea.

(questo è quello che ho appena appuntato)

Implementing a gossiping protocol

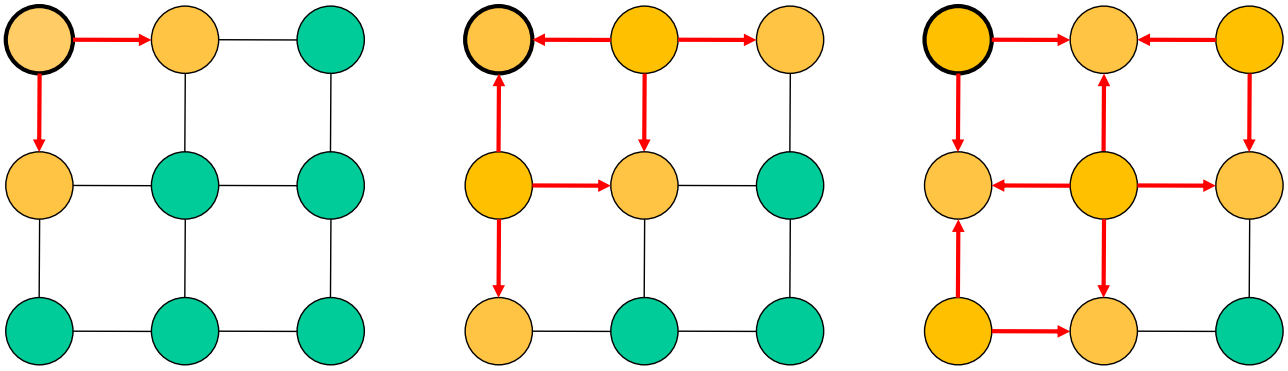
What specific problems need to be addressed when implementing a gossiping protocol?

- Membership: How node can get to know each other and how many acquaintances to have
- Network awareness: How to make node links reflect network topology for satisfactory performance
- Cache management: what information to discard when node's cache is full
- Message filtering: how to consider nodes' interest in the message and reduce the likelihood that they will receive information they are not interested in

Gossiping vs flooding: example

- Information dissemination is the classic and most popular application of gossip protocols in DSs
  - Gossiping is more efficient than flooding
- Flooding-based information dissemination
  - Each node that receives the message propagates it to its neighbors (let's consider all neighbors, including the sender)
  - Message is eventually discarded when TTL=0

Round 1 | Round 2 | Round 3



Sent messages: 18
Reached nodes: 8 out of 9

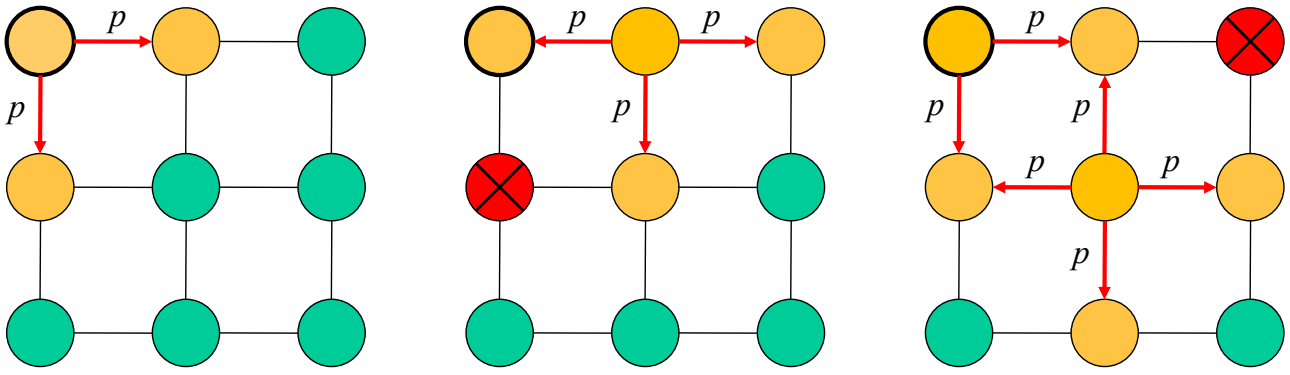
Gossiping vs flooding: example

- Let's use only pure gossiping
  - Message is sent to neighbors with probability *p*

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

se p > rand.value(0,1) -> invia messaggio

Round 1 | Round 2 | Round 3



Gossiping ha raggiunto un nodo in meno, ma ho anche risparmiato 7 messaggi!

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

Più lento a parità di round, rispetto a flooding.

- 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

In che altri contesti può essere utile il gossiping?

Other application domains of gossiping

- Besides information dissemination…
- Peer sampling
  - How to provide every node with a list of peers to exchange information with

Nei sistemi di bootstrap/p2p, è necessario fornire ad un nuovo nodo una lista di nodi contattabili

- Resource management, including monitoring, in large-scale distributed systems
  - E.g., failure detection

Monitoraggio/gestione risorse.
Come abbiamo anche visto prima, posso usarlo per diffondere info su nodi down.

- Distributed computations to *aggregate* data, in particular in large sensor networks
  - Computation of aggregates, such as sums, averages, and maximum and minimum values
  - E.g., to compute an 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

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

In questo terzo caso, esaminiamo un contesto in cui i dati sono posseduti in modo distribuito da nodi del sistema.
Un classico esempio sono le reti di sensori, i quali hanno energia limitata e voglio minimizzare scambio di informazioni.

Nell'esempio del valore medio, i valori sono posseduti da nodi diversi.
Tramite push-pull, i nodi si scambiano i dati, cosi ciascuno può calcolare il valore medio della rete (e fare anche tante altre operazioni).

Two gossiping protocols

- Let's now examine two examples of gossiping protocols Fino ad ora abbiamo visto algoritmi, ma non chi li implementasse.
  - **Blind counter rumor mongering**
  - **Bimodal multicast**

Un nodo che diffonde l'informazione perde interesse a diffonderla, non importa quanto sia rilevante.

Dopo tot contatti, smette la trasmissione dell'info.

diffondo il pettegolezzo

Blind counter rumor mongering

- Why that name for this gossip 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 the recipient (*why*)
  - *Counter*: loses interest after some contacts (*when*)

- Two parameters for this gossip protocol
  - *B*: how many nodes to pick up as gossip targets
  - *F*: how many nodes to contact before losing interesting aumenta tolleranza

Nel gossiping visto prima si parlava di fanout = 1, qui invece la cardinalità dei nodi contattati è "B", mentre "F" è il counter prima di "stancarsi".

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

"p" invia tramite BROADCAST un messaggio "m" a "B" vicini, scelti randomicamente.
Quando "p" riceve un messaggio 'm' dal nodo 'q':
Se 'p' ha ricevuto questo messaggio un numero di volte < F, 'p' inoltra questo msg a B vicini scelti random, escludendo quelli che hanno già visto il msg stesso. Come li riconosco? O già li ho contattati per inviare 'm', oppure loro hanno contattato me!

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*

Blind counter rumor mongering: performance

- Difficult to obtain analytical expressions to describe the behavior of gossiping protocols, even for relatively simple topologies ⇒ simulation analysis
- Assume Barabási network topology
  - 1000 nodes with an average node degree of 6

	Flooding				Rumor Mongering (F=2, B=2)			
	mean	min	max	std dev	mean	min	max	std dev
Cost	4990	4990	4990	0	2555.42	2471	2638	25.02
Reach	1000	1000	1000	0	918.44	894	945	8.52
Time	8.94	8	10	0.36	21.33	19	30	1.32

  - Rumor mongering vs flooding scalability (F=2, B=2)

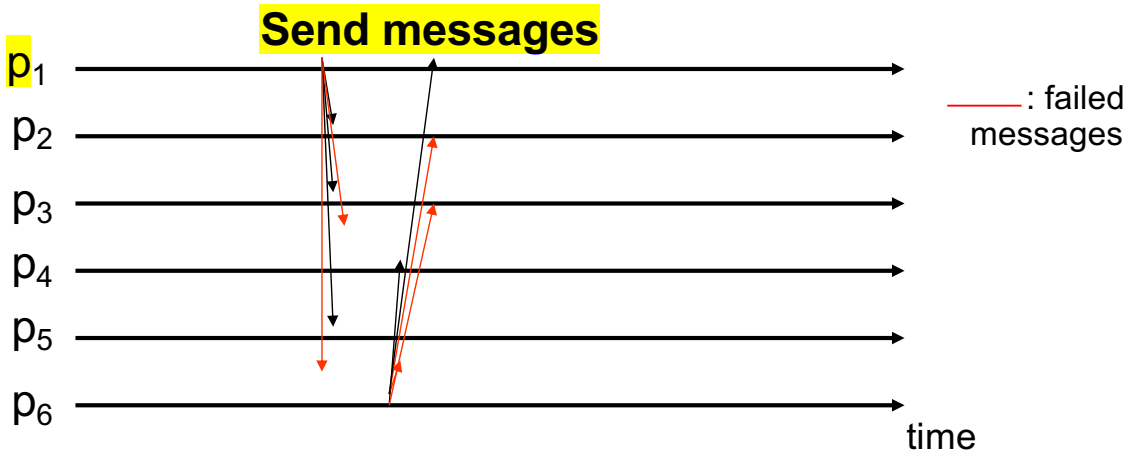
N	cost	reach	time
100	52.60%	93.70%	198%
1000	51.20%	91.10%	240%
10'000	50.00%	92.30%	290%

Bimodal multicast

- Also called pbcast (probabilistic broadcast)
- Composed by two phases:
  1. *Message distribution* phase: a process sends a multicast with no particular reliability guarantees
     - IP multicast if available, otherwise some application-level multicast (e.g., Scribe trees)
  2. *Gossip repair* phase: 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

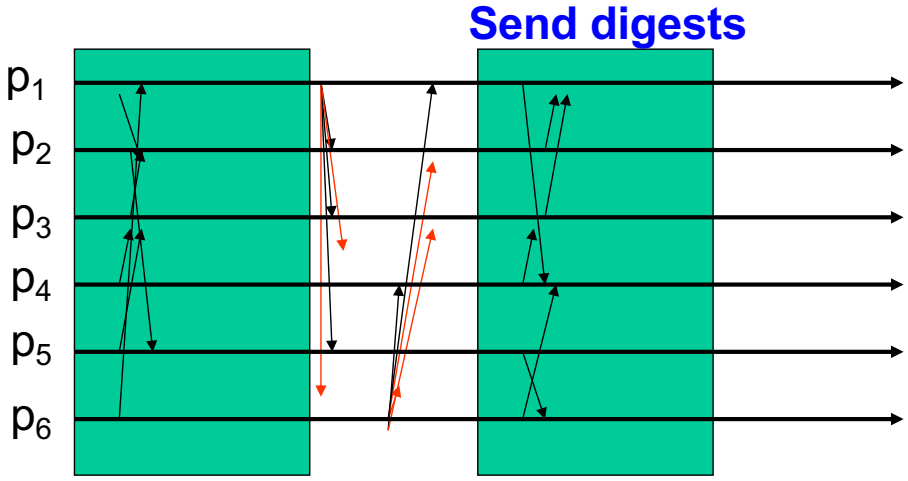
Birman et al., Bimodal multicast, *ACM Trans. Comput. Syst., 1999*

Bimodal multicast: message distribution



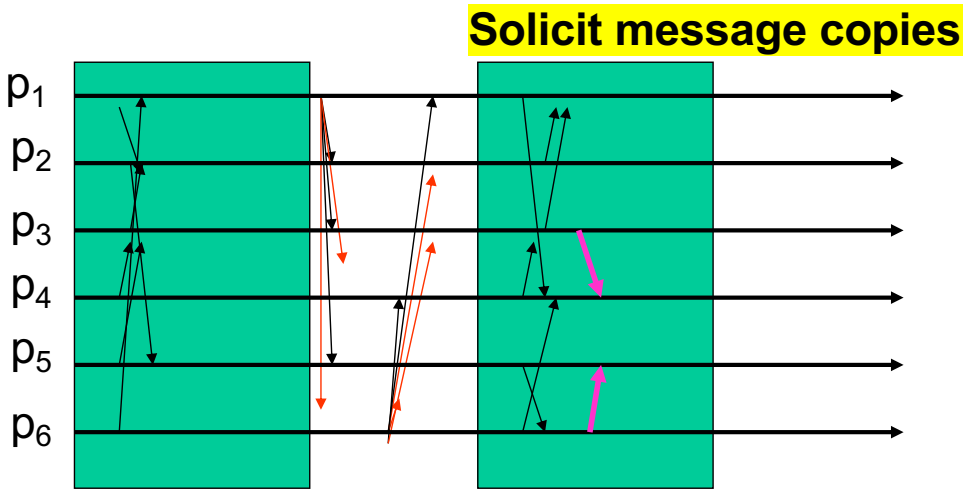
- Start by using *unreliable* multicast to rapidly distribute the message
- But some messages may not get through, and some processes may be faulty
- So initial state involves partial distribution of multicast(s)

Bimodal multicast: gossip repair



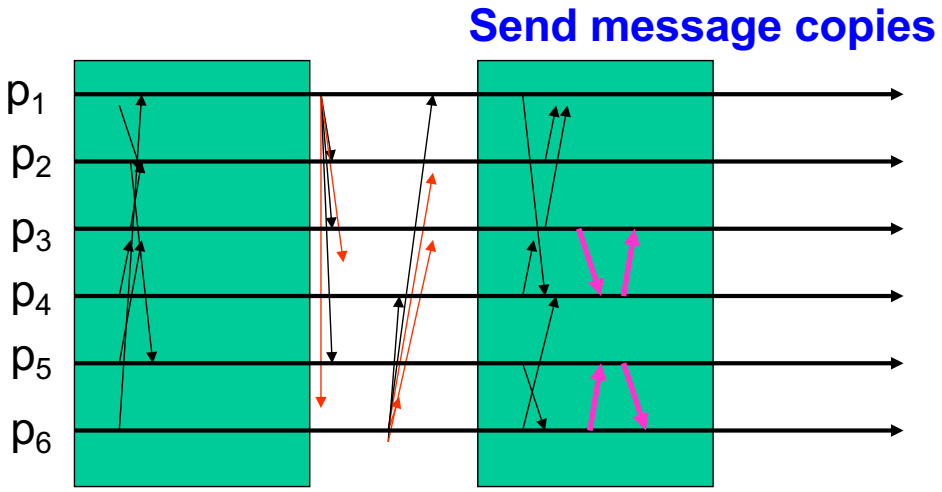
- Periodically (e.g., every 100 ms) each process sends a *digest* describing its state to some randomly selected process
- The digest identifies messages: it does not include them

Bimodal multicast: gossip repair



- Recipient checks the 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 respond to solicitations received during a round of gossip by retransmitting the requested message
- Various optimizations (not examined)

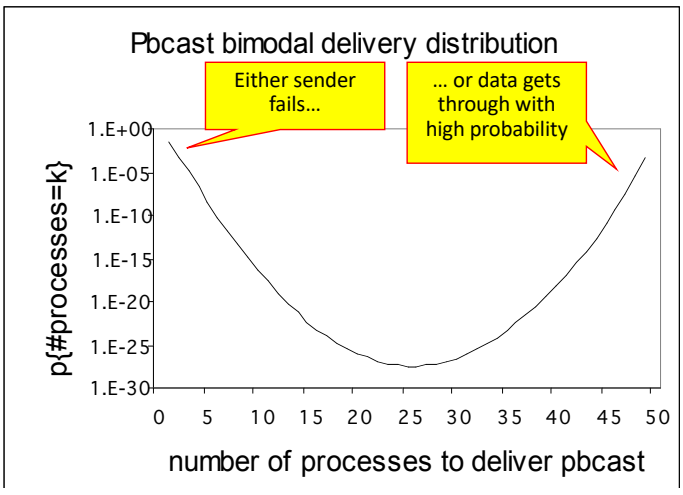
Bimodal multicast: why "bimodal"?

- 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

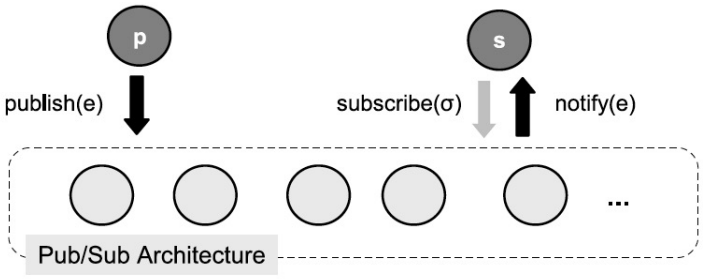
2. A second bimodal characteristic is due to delivery latencies, with one distribution of very low latencies (messages that arrive without loss in the first phase) and a second distribution with higher latencies (messages that had to be repaired in the second phase)



Distributed event matching

- **Event matching** (aka notification filtering): core functionality of pub/sub systems

- Event matching requires:
  - Checking subscriptions against events
  - Notifying a subscriber in case of match



Distributed event matching: centralized architecture

- First solution: centralized architecture
- Centralized server handles all subscriptions and notifications
- Centralized server:
  - Handles subscriptions from subscribers
  - Receives events from publishers
  - Checks every and each event 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 divides work among 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 (called brokers) among which work is divided
  - Organized in a flat architecture, hashing can still be used
  - Example: Kafka

Distributed event matching: distributed architecture

- Let's examine other alternatives: fully distributed (decentralized) solutions
  - Rely on P2P overlay networks
- P2P unstructured overlay: use flooding or gossiping as information dissemination mechanisms
  - Trivial solution for flooding: propagate each event from publisher to all the nodes in the P2P system
  - To reduce message overhead of flooding, selective event routing can be used
- Alternatively, can also rely on a P2P structured overlay
  - Example: Scribe