Indirect Communication

Distributed software systems

CdLM Informatica - Università di Bologna

Agenda

Group communication in distributed systems

Publish-subscribe systems

Message queues

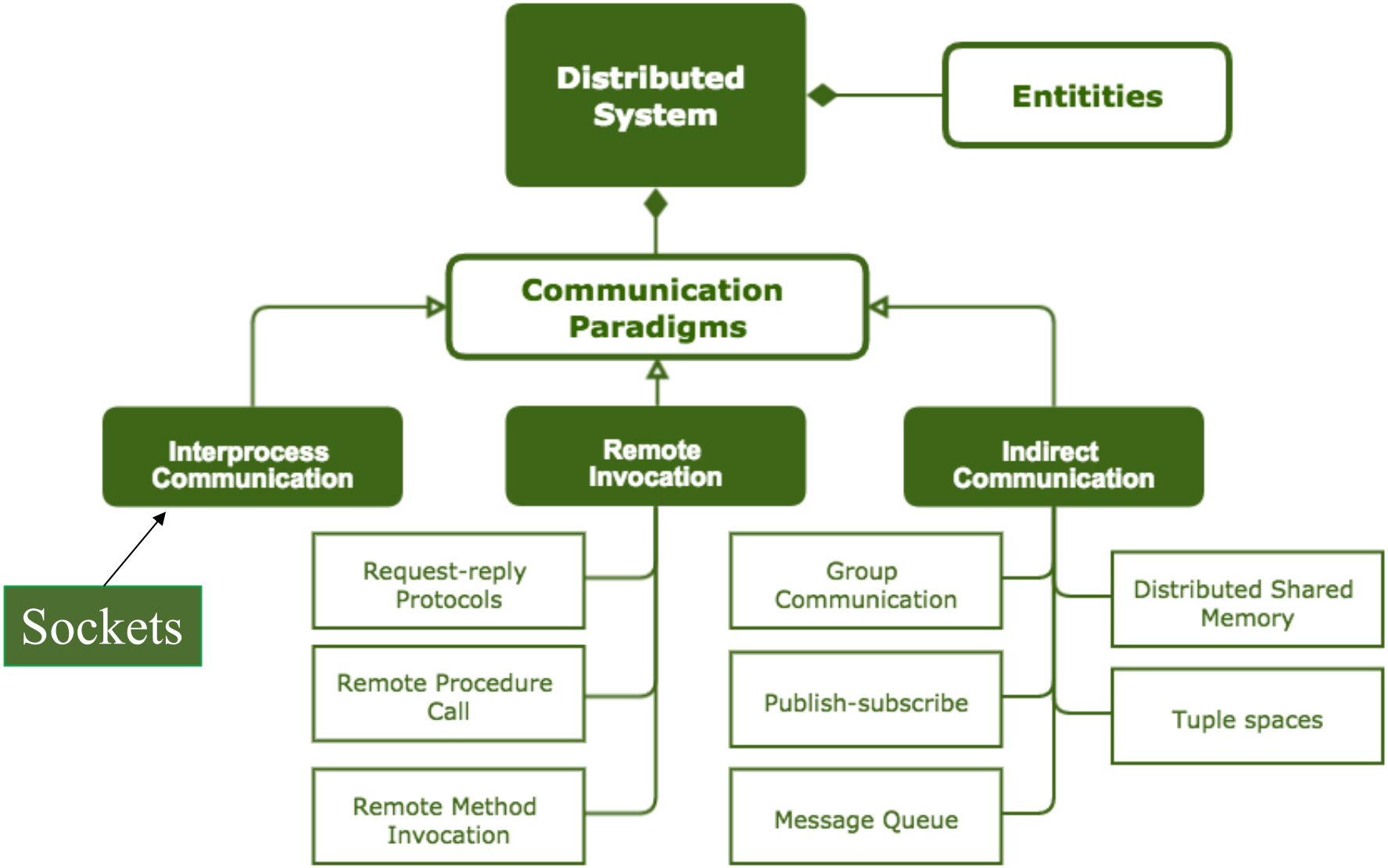
Distributed shared memory

RPC today (a comment from Quora)

RPC was invented a long time ago. While it was popular for a while, it is not the mail model in use today. There are several issues:

- Normal procedure calls (function calls) in high-level languages are protected by strong type checking. How can we ensure type safety?
- Endianness: How do we deal with the differences between big and little endianness efficiently? XDR (https://en.wikipedia.org/wiki/External_Data_Representation) is an entire layer to implement just to deal with this specific issue.
- Liveness: What happens if the callee never returns?
- State: does the callee maintain state? If so, what is the lifetime of that state? How does the callee track the caller to see if the state can be discarded?
- The world has moved on since then. RPC has been replaced by message passing and automated by messaging tools such as Google Protocol Buffers and Thrift.

Communication paradigms in distributed systems



https://dzone.com/articles/message-brokers-in-indirect-communication-paradigm

How can we decouple the sender and the receiver?

A basic weakness of both send/receive messages and remote invocations, i.e. **direct communication mechanisms**, is that they introduce a dependency between – respectively - the sender and the receiver or between the caller and the callee

The dependency is both functional and temporal

How can we get rid of this dependency?

Indirect communication

The essence of indirect communication is to communicate through an intermediary and have no direct coupling between the sender and the receiver(s)

The concepts of *space and time uncoupling* are useful for:

- group communication, in which communication is via a group abstraction with the sender unaware of the identity of the recipients;
- publish-subscribe systems, where publishers are producing events toward multiple recipients through an intermediary;
- message queue systems, wherein messages are directed to the abstraction of a queue with receivers extracting messages from such queues;
- shared memory–based approaches, including 1) distributed shared memory and 2) tuple space approaches, which offer to programmers an abstraction of a logically shared memory

Space and time coupling in distributed systems

	Time coupled	Time uncoupled
Space coupling	Properties: Communication directed to given receivers that must exist at that moment in time Examples: message passing, remote invocation	Properties: Communication directed to given receivers; senders and receivers can have non overlapping lifetimes Example: message queues
Space uncoupling	Properties: Sender does not need to know the identity of the receivers; senders and receivers must exist at that moment in time Examples: broadcast, IP multicast	Properties: Sender does not need to know the identity of the receivers; senders and receivers can have non overlapping lifetimes Examples: group communication

Model	Guarantees	Cost	Scalability
Virtual Synchrony	strong	Very expensive	low
Reliable multicast	Reliable, weak order	high	moderate
Causal consistency	Causality preserved	moderate	good
Eventual consistency	Eventual convergence	Moderate/low	high
RESTful interactions	Eventual consistency	moderate	high
Pub/sub	Loose, async	low	Very high
Gossip protocol	Probabilistic, eventual	Very low	Extremely high
Best effort broadcast	No guarantees	Very low	infinite

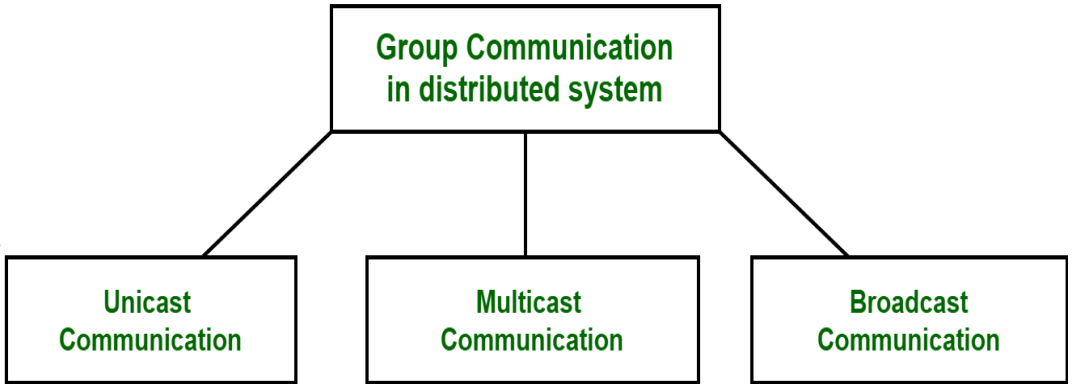
Group communication

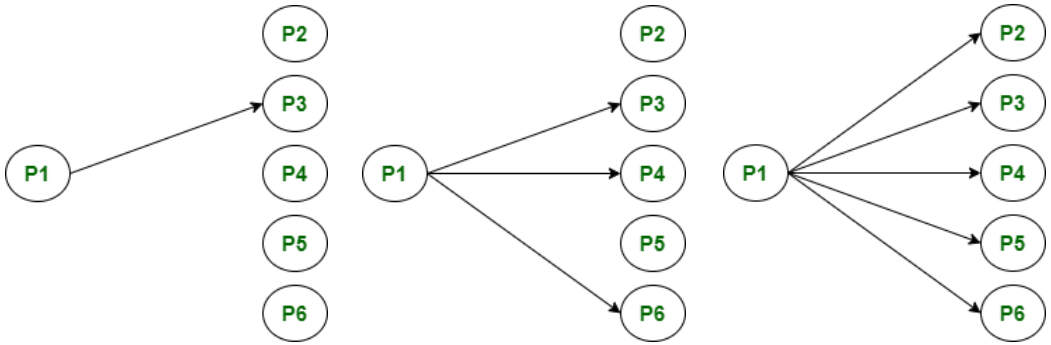
When one source process tries to communicate with multiple processes at once, it is called **Group Communication**.

A group is a collection of abstractly interconnected processes

The abstraction hides the message passing so that the communication looks like a normal procedure call.

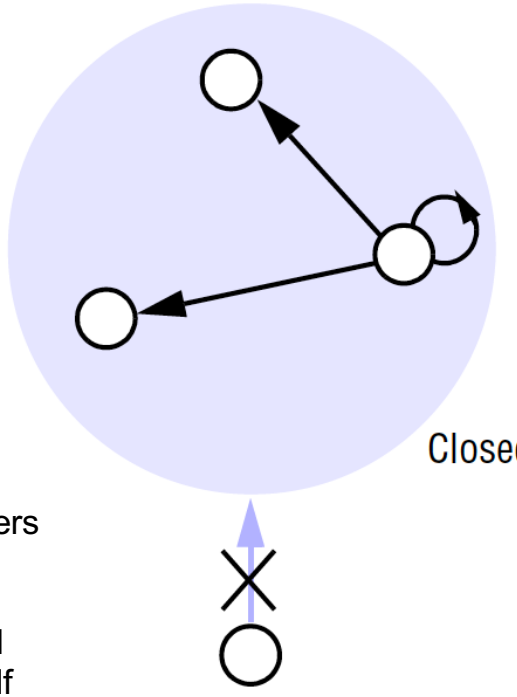
Group communication also helps the processes from different hosts to work together and perform operations in a synchronized manner, therefore increases the overall performance of the system.



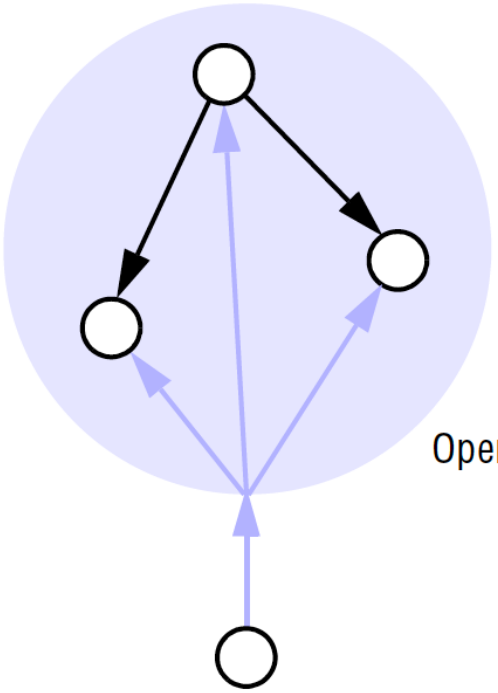


Open, closed and overlapping groups

A group is said to be closed if only members of the group may multicast to it
A process in a closed group delivers to itself any message that it multicasts to the group.

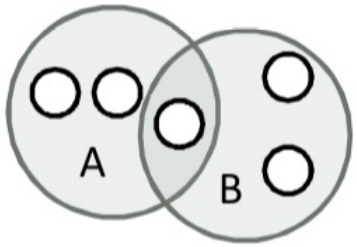


Closed group

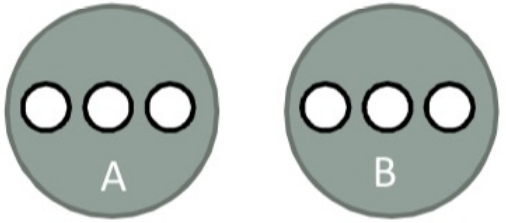


Open group

A group is open if processes outside the group may send to it.



overlapping group



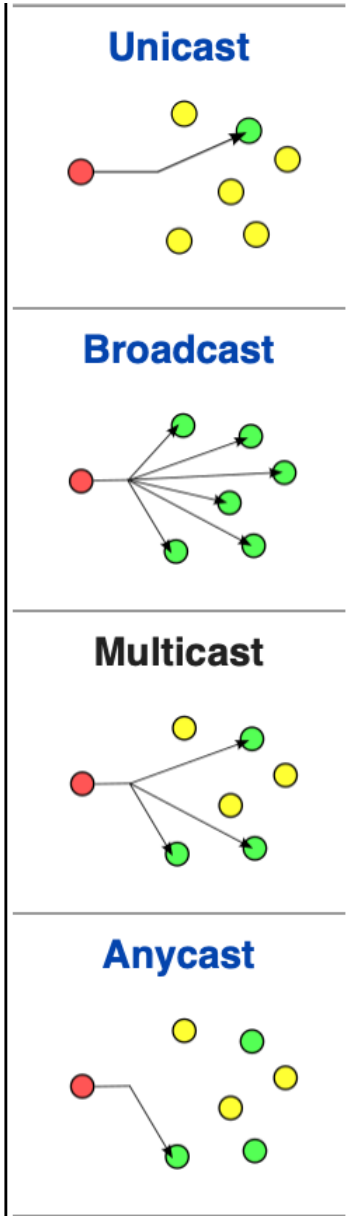
non-overlapping group

Multicast

- Multicast is group communication where data transmission is addressed to a group of destination computers simultaneously.
- Multicast can be one-to-many or many-to-many.
- Multicast should not be confused with physical layer point-to-multipoint communications
- IP multicast is a technique for one-to-many communication over an IP network

We use multicast for

- Fault tolerance based on replicated services
- Discovering services in spontaneous networking
- Improve performance though replicated data
- Propagation of event notifications



A note on IP multicast

- IP multicast allows the sender to transmit a single IP packet to a set of computers that form a multicast group.
- The sender is unaware of the identities of the individual recipients and of the size of the group.
- A multicast group is specified by a Class D Internet address whose first 4 bits are 1110 in IPv4.
- Being a member of a multicast group allows a computer to receive IP packets sent to the group.
- The membership of multicast groups is dynamic, allowing computers to join or leave at any time and to join an arbitrary number of groups.
- It is possible to send datagrams to a multicast group without being a member.
- At the application programming level, IP multicast is available via UDP

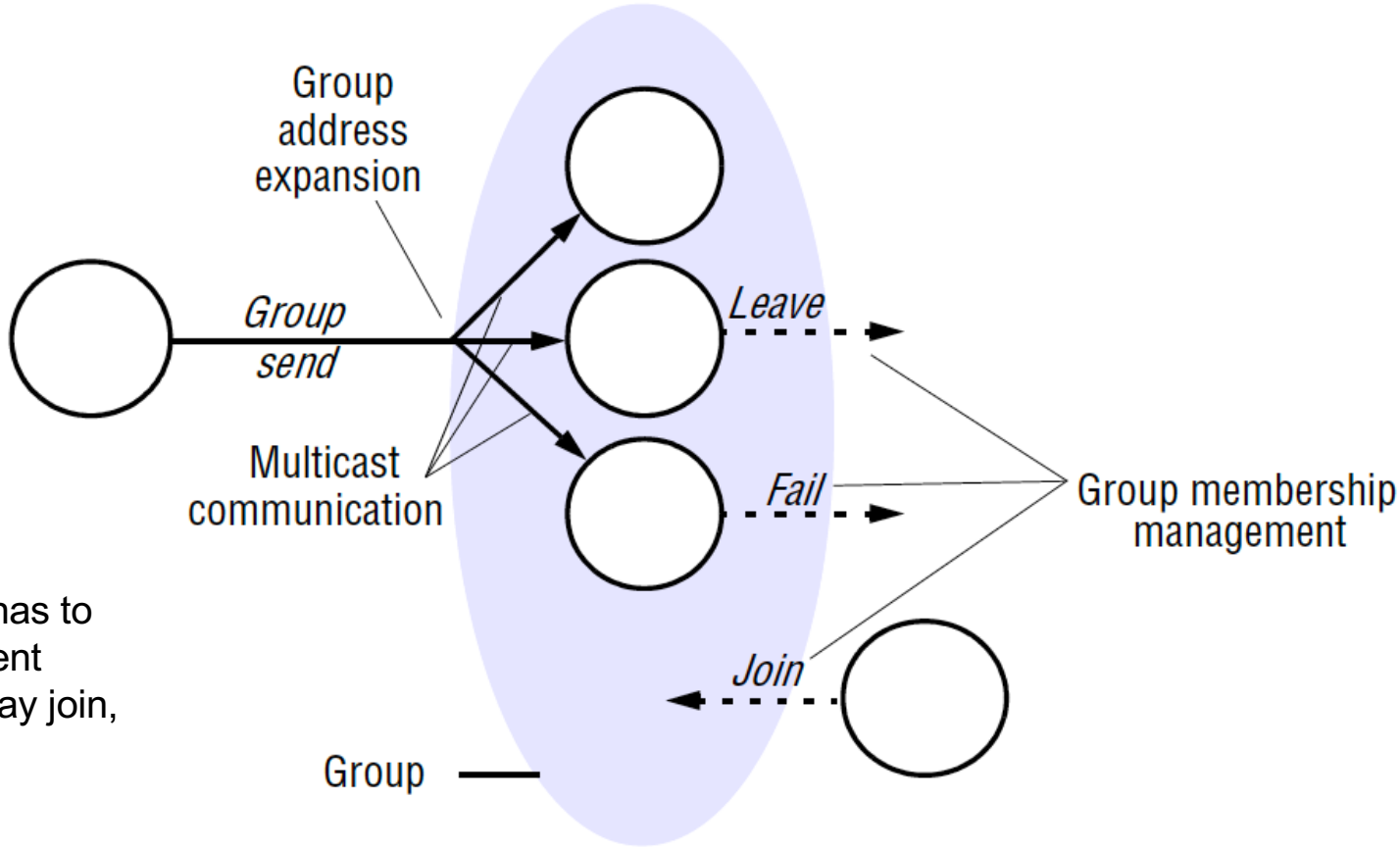
Ordering multicast messages

Ordering is not guaranteed by IPC primitives.

Group communication services offer *ordered multicast*, with the option of one or more of the following properties (with hybrid solutions also possible):

- FIFO ordering : (also referred to as source ordering) preserves the order from the perspective of a sender process: if a process sends one message before another, it will be delivered in this order at all processes in the group.
- Causal ordering : Causal ordering takes into account causal relationships between messages: if a message **happens before** another message in the distributed system this so-called causal relationship will be preserved in the delivery of the associated messages at all processes (we have to define precisely ‘happens before’).
- Total ordering : In total ordering, if a message is delivered before another message at one process, then the same order will be preserved at all processes

The role of group membership management



group membership management has to keep an accurate view of the current membership, given that entities may join, leave or even fail

Open group

Group membership service

A group membership service has four main tasks:

1. Providing an interface for group membership changes : The membership service provides operations to create and destroy process groups and to add or withdraw a process to or from a group. In most systems, a single process may belong to several groups at the same time (overlapping groups). Example: IP multicast.
2. Failure detection : The service monitors the group members not only in case they should crash, but also in case they should become unreachable because of a communication failure. The detector marks processes as Suspected or Unsuspected . The service uses the failure detector to reach a decision about the group's membership: it excludes a process from membership if it is suspected to have failed or to have become unreachable.
3. Notifying members of group membership changes : The service notifies the group's members when a process is added, or when a process is excluded (through failure or when the process is deliberately withdrawn from the group).
4. Performing group address expansion : When a process multicasts a message, it supplies the group identifier rather than a list of processes in the group. The membership management service expands the identifier into the current group

ISIS (by Ken Birman @ Cornell University)

- Birman is author of the Isis Toolkit, which introduced the **virtual synchrony** execution model for multicast communication.
- Birman founded Isis Distributed Systems to commercialize this software, which was used by stock exchanges, for air traffic control, and in factory automation.
- The Isis software operated the New York and Swiss Stock Exchanges, the French air traffic control system and the US Navy AEGIS warship
- ISIS allows distributed systems to automatically adapt themselves when failures or other disruptions occur, to securely share keys and security policy data, and to replicate critical services so that availability can be maintained even while some system components are down. Birman released a version of the Isis technology, Vsync, as an open-source library

The Virtual Synchrony model

- Virtual synchrony is an interprocess message passing (sometimes called ordered, reliable multicast) technology.
- Virtual synchrony systems allow programs running in a network to organize themselves into process groups, and to send messages to groups (as opposed to sending them to specific processes).
- Each message is delivered to all the group members, in the identical order, and this is true even when two messages are transmitted simultaneously by different senders.
- Application design and implementation is greatly simplified by this property: every group member sees the same events (group membership changes and incoming messages) in the same order.

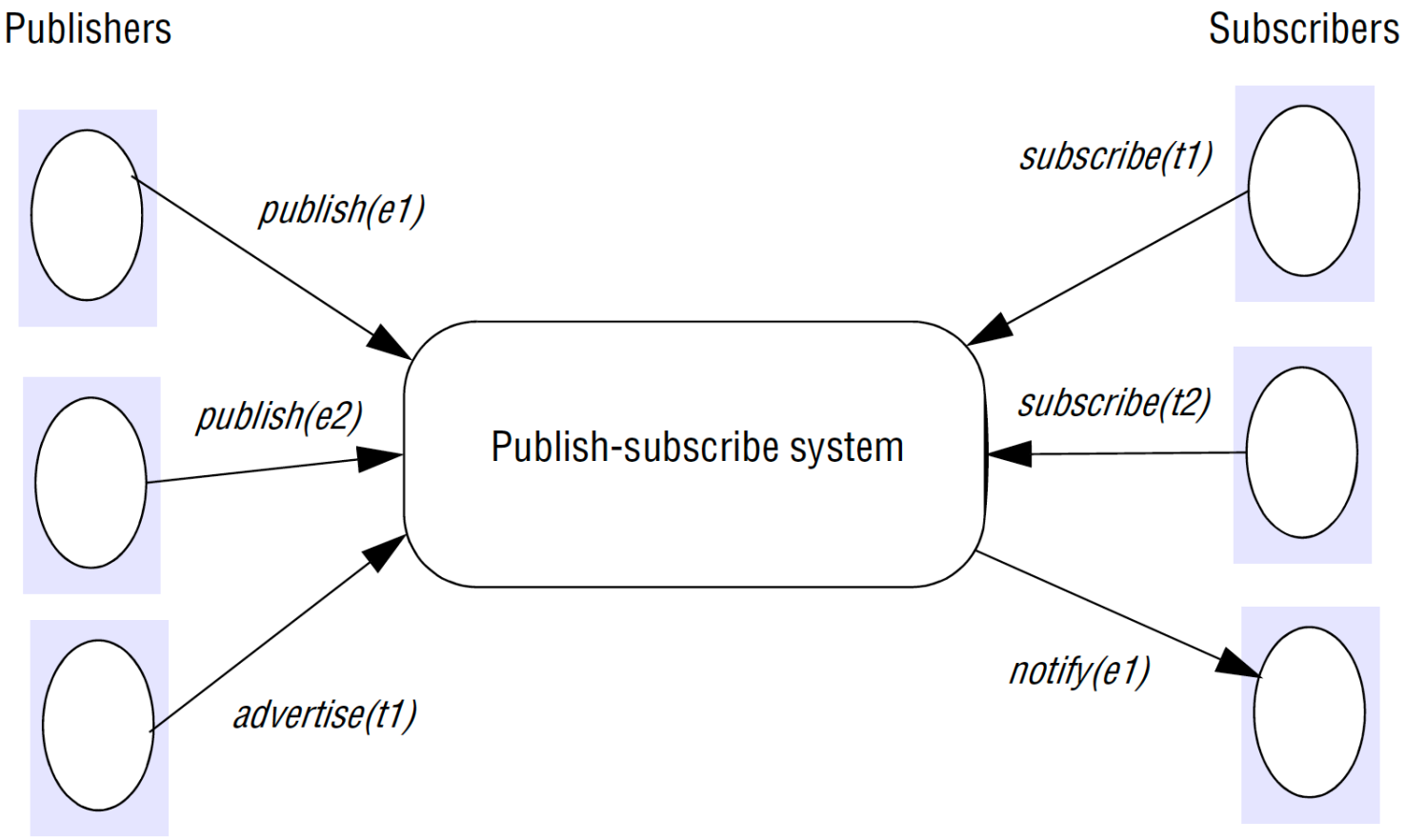
Implementing the Virtual Synchrony model

- A virtually synchronous service is typically implemented using a style of programming called state machine replication, in which a service is first implemented using a single program that receives inputs from clients through some form of remote message passing infrastructure, then enters a new state and responds in a deterministic manner.
- The initial implementation is then transformed so that multiple instances of the program can be launched on different machines, using a virtually synchronous message passing system to replicate the incoming messages over the members.
- The replicas will see the same events in the same order, and are in the same states, hence they will make the same state transitions and remain in a consistent state.

The publish-subscribe architectural pattern

Publish/Subscribe is an interaction pattern between publishing and subscribing clients. Subscribers express interest in receiving messages and publishers produce messages without specifying the recipients for a message.

IMPORTANT: the publish/subscribe message exchange is decoupled and anonymous

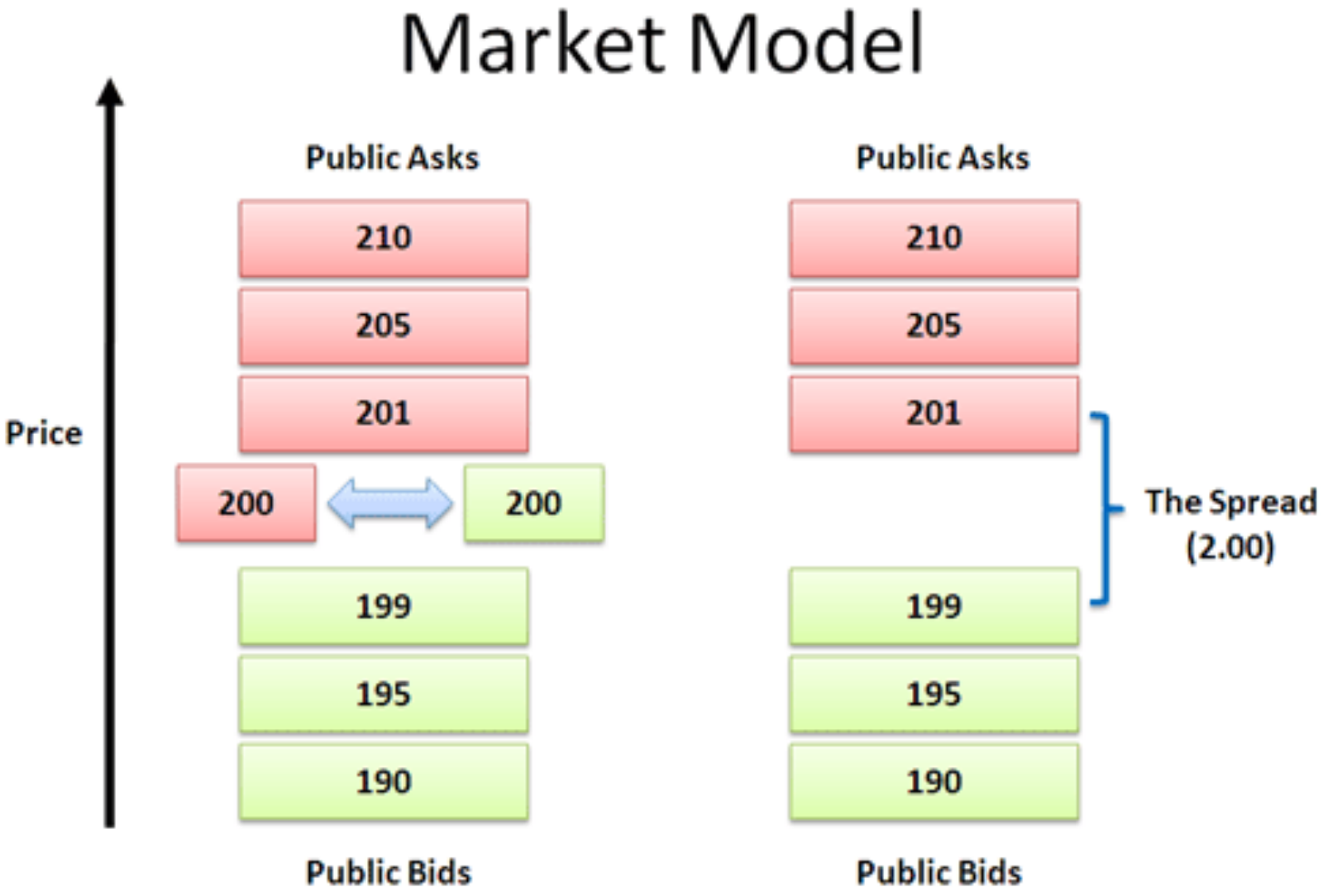


Publish subscribe systems

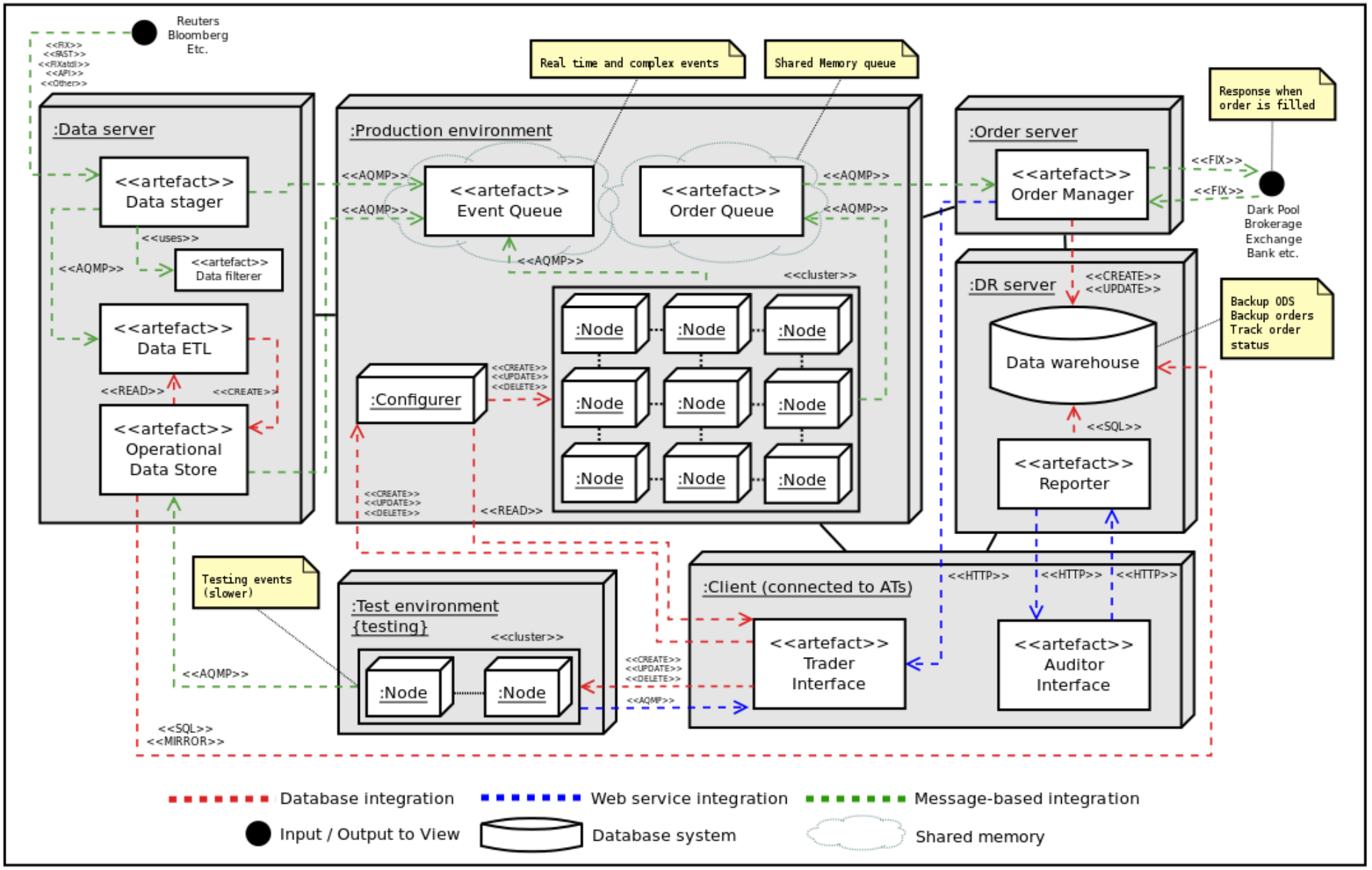
Publish-subscribe systems have three main features:

- Heterogeneity: sw components designed independently may work together
- Asynchronicity: publishers and subscribers are time decoupled
- A variety of different delivery guarantees can be provided for notifications - the one that is chosen should depend on the application requirements

Stock exchange: an example of distributed system



Algorithmic Trading system (ATs) High Level Deployment View

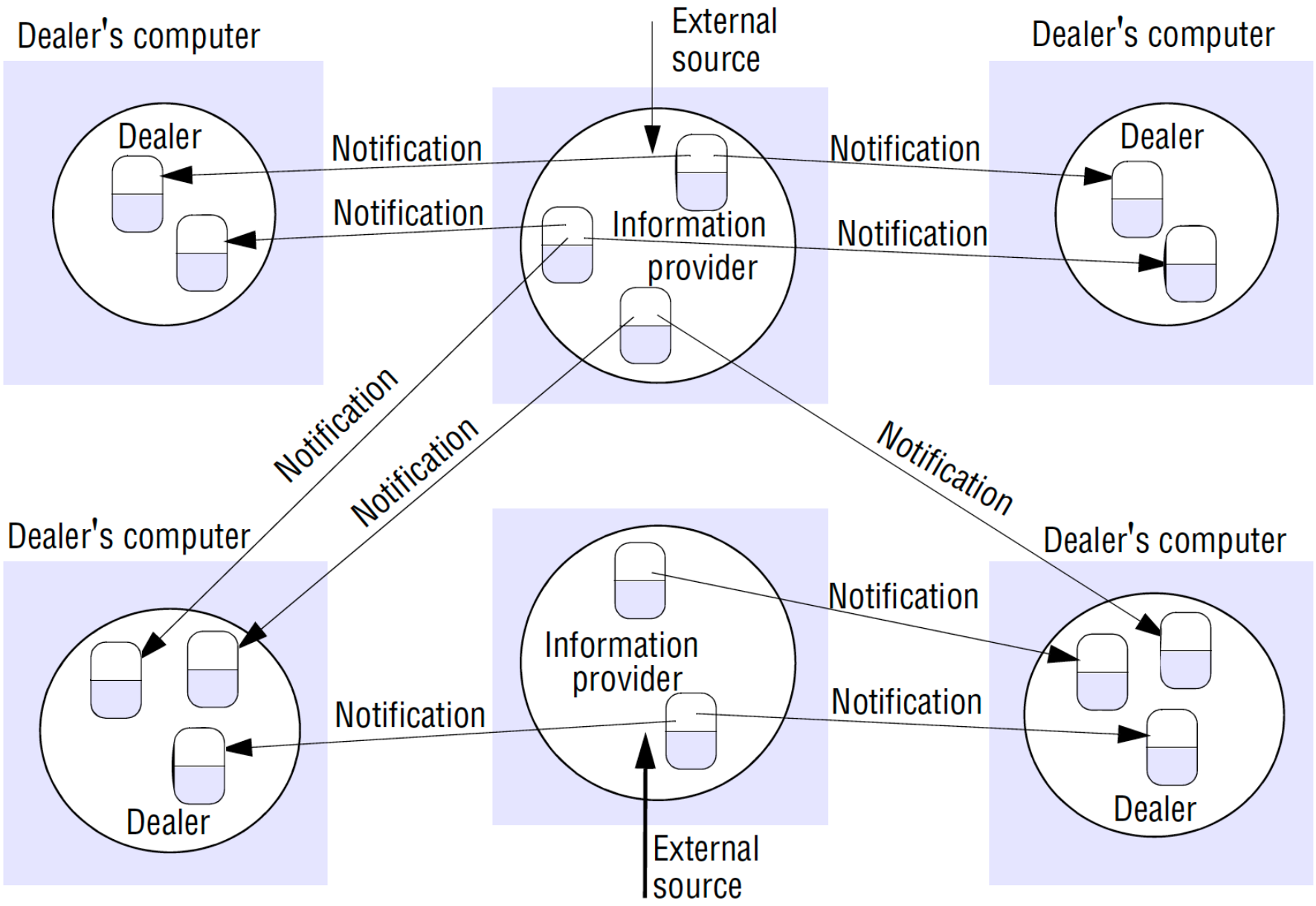


Dealing room system

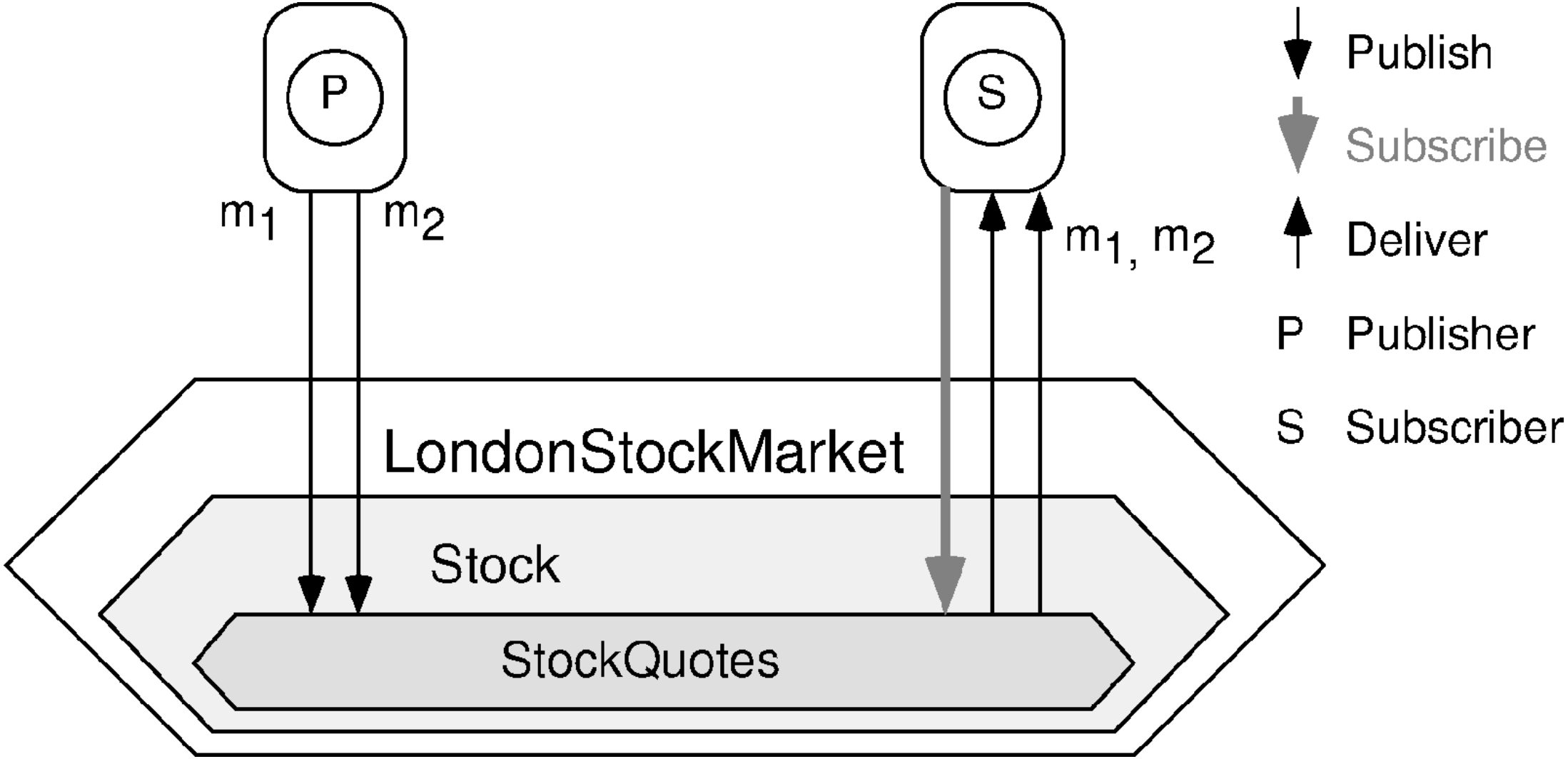
a simple dealing room system whose task is to allow dealers to see the latest data about the market prices of the stocks they deal in

The market price for a single named stock is represented by an associated object

The information arrives in the dealing room from several different external sources in the form of updates to some or all of the objects representing the stocks



Example: stock exchange as Pub/Sub



From: Eugster et alii, The many faces of Publish/Subscribe, ACM Comp. Surv., 35:2, 2003

Variants of publish subscribe schemes

Channel-based:

In this approach, publishers publish events to named channels and subscribers then subscribe to one of these named channels to receive all events sent to that channel.

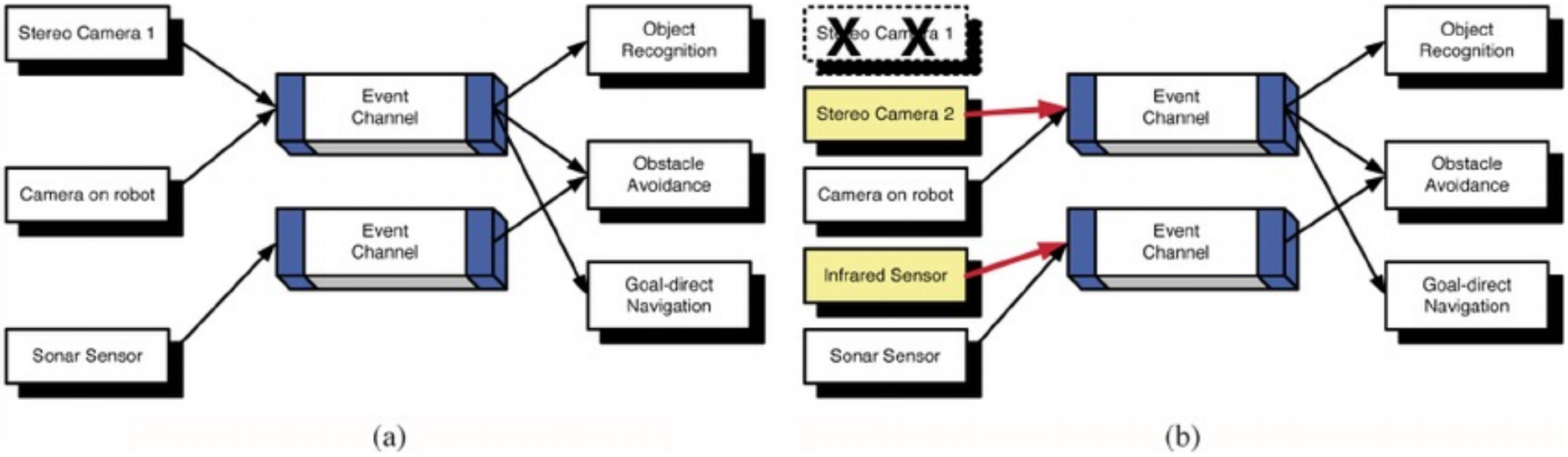
Example: CORBA event service

Example: CORBA Event Service

An example of using CORBA Event Service in robot navigation.

(a) Robot navigation using event channel.

(b) In case of the change of configuration, we just need to reconfigure the event channel connection. We can change the setup by reconfiguring the event channel in case of adding new components (Stereo Camera and Infrared Sensor) and deleting component (Stereo Camera 1)



Source: Song, CONCORD: A Control Framework for Distributed Real-Time Systems, IEEE Sensors journal, 2007

Variants of publish subscribe schemes

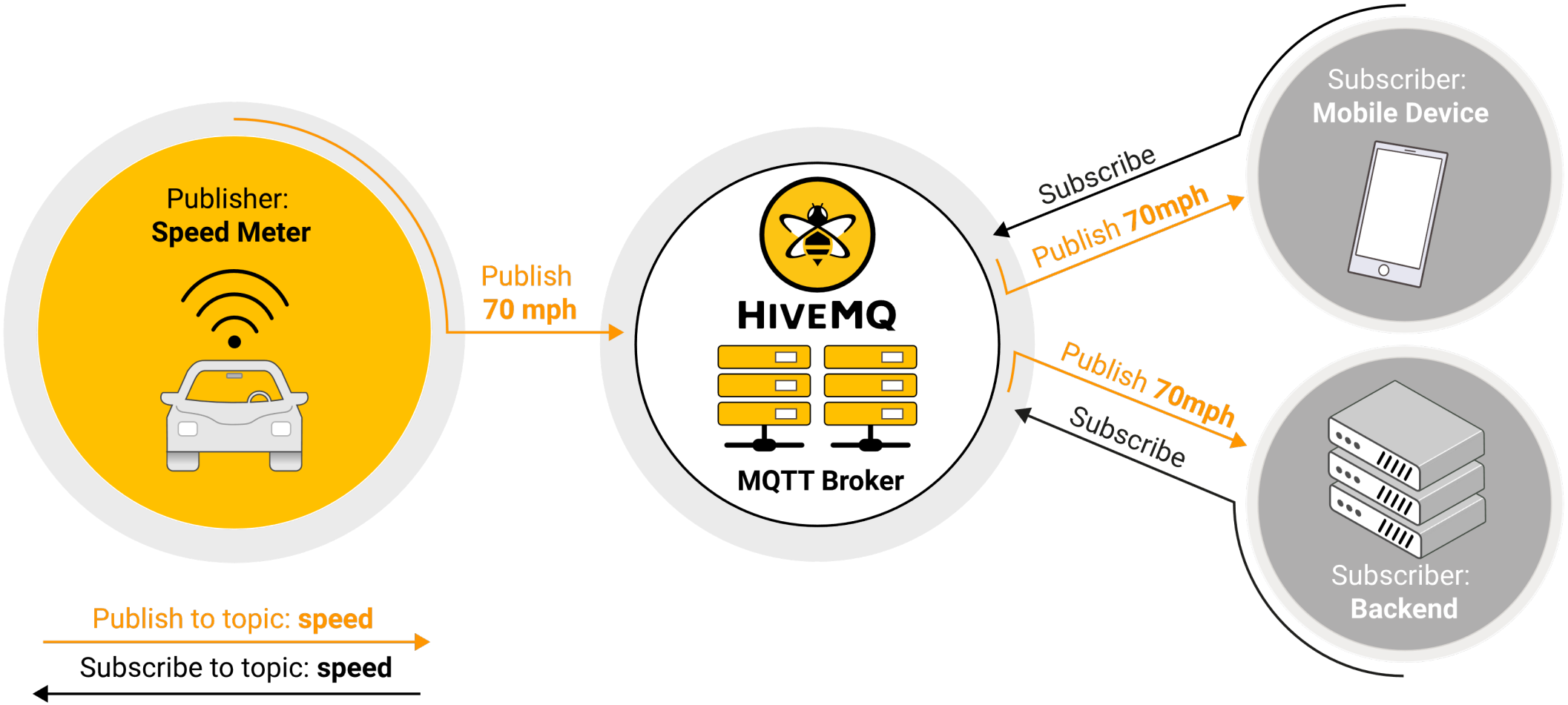
Topic-based (also referred to as subject-based):

Here, each notification is expressed in terms some *fields*, with one field denoting the topic.

Subscriptions are then defined in terms of the topic of interest.

This is equivalent to channel-based approaches, with the difference that topics are implicitly defined in the case of channels but explicitly declared in topic-based approaches

Example: topic based publish subscribe in IoT with MQTT



https://www.hivemq.com/blog/how-to-get-started-with-mqtt/

Variants of publish subscribe schemes

Content-based: Content-based approaches are a generalization of topic-based approaches allowing the expression of subscriptions over a range of fields in an event notification.

A content-based filter is a query defined in terms of compositions of constraints over the values of event attributes.

For example, a subscriber could express interest in events that relate to the topic of sport and regional news, where the sport topic is '*soccer teams*' and where the country is '*Italy*'

The sophistication of the associated query languages varies from system to system, but in general this approach is significantly more expressive than channel- or topic-based approaches

Example: content based P/S system

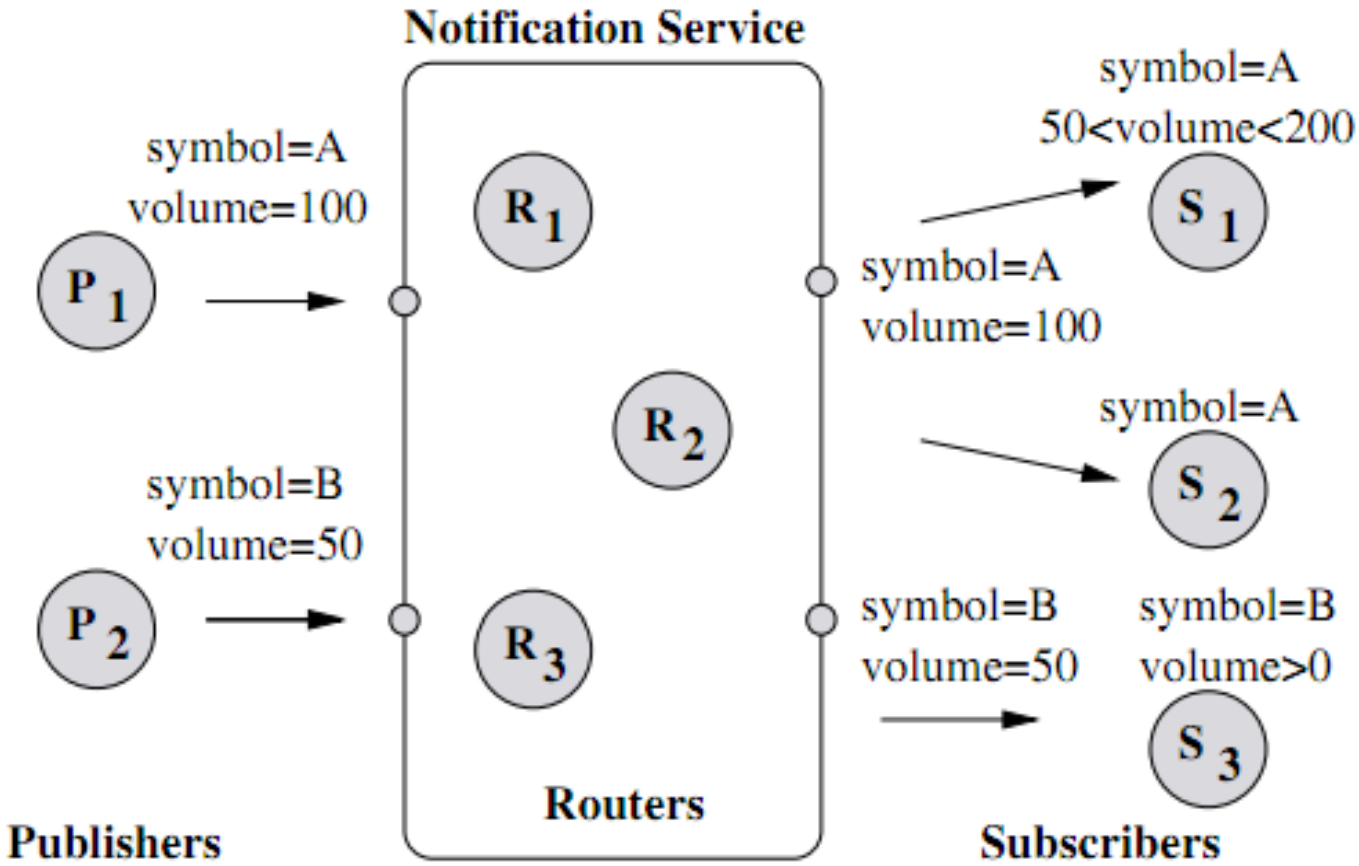


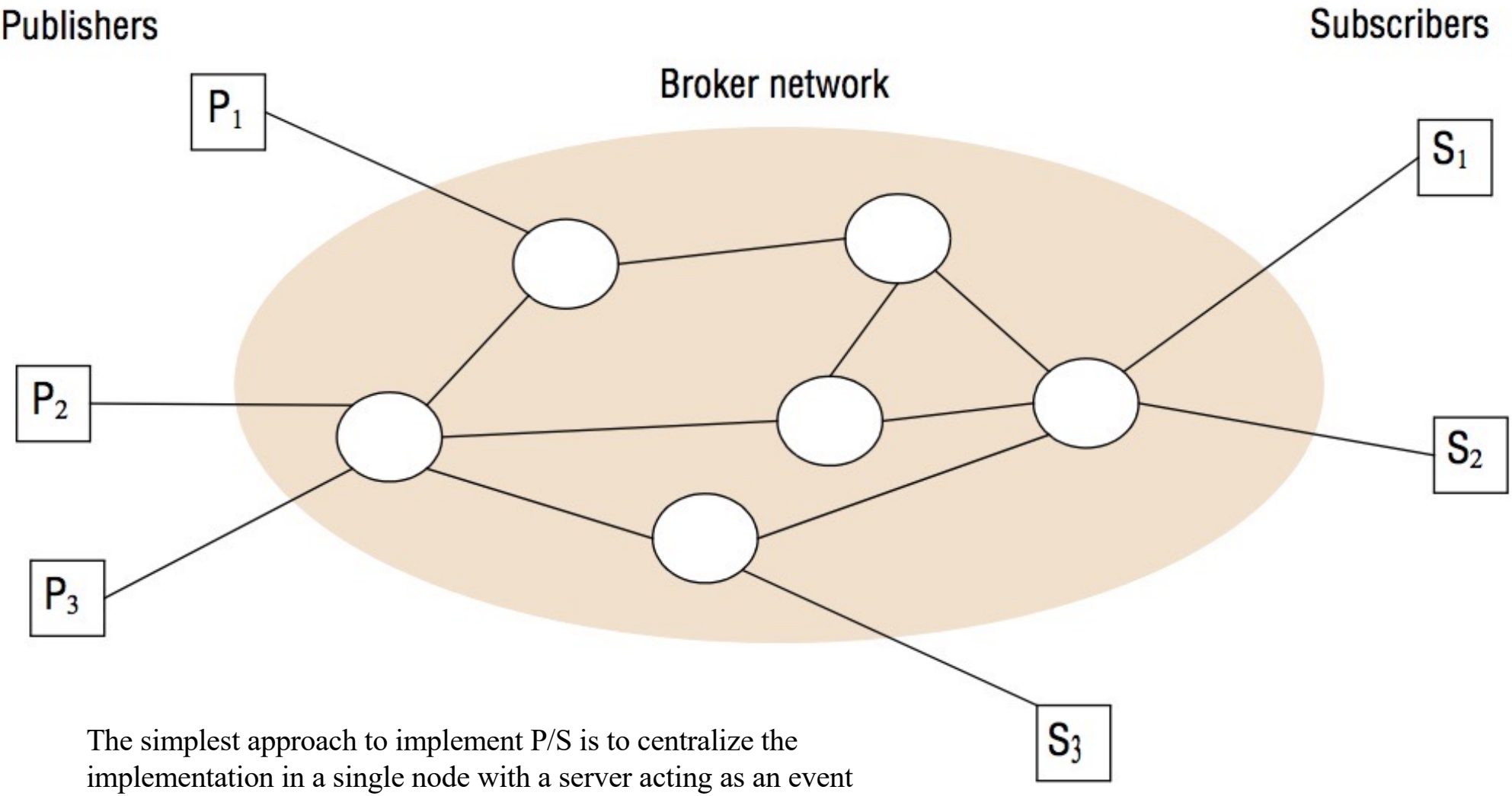
Figure 4.6: Example of content-based publish/subscribe.

Variants of publish subscribe schemes

Type-based: These approaches are linked with object-based approaches where objects have a specified type.

- In type-based approaches, subscriptions are defined in terms of *types* of events, and matching is defined in terms of types or subtypes of the given filter.
- This approach can express a range of filters, from coarse-grained filtering based on overall type names to more fine-grained queries defining attributes and methods of a given object.
- Such fine-grained filters are similar to content-based approaches.
- The advantages of type-based approaches are that they can be integrated elegantly into programming languages and they can check the type correctness of subscriptions, eliminating some kinds of subscription errors

A network of brokers

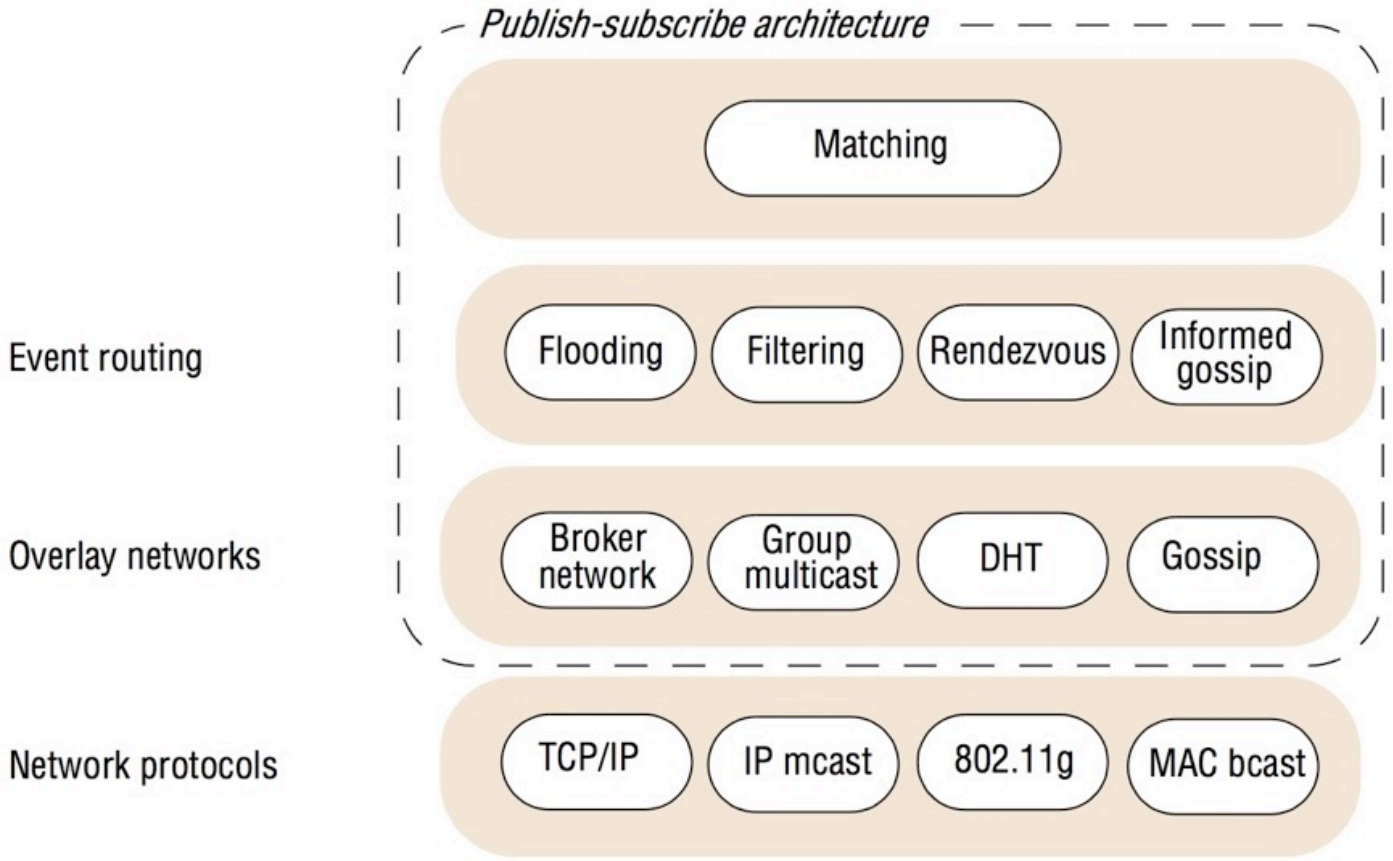


The simplest approach to implement P/S is to centralize the implementation in a single node with a server acting as an event broker

The architecture of publish-subscribe systems

The top layer implements matching – that is, ensuring that events match a given subscription. While this can be implemented as a discrete layer, often matching is pushed down into the event routing mechanisms.

Event routing performs the task of ensuring that event notifications are routed as efficiently as possible to appropriate subscribers, whereas the overlay infrastructure supports this by setting up appropriate networks of brokers or peer-to-peer structures.



Implementations of publish-subscribe

- Flooding: sending an event notification to **all** nodes
- Filtering: brokers forward notifications only to **valid** subscribers
- Advertisements: Let the publishers *advertise* that they will publish specific events
- Rendezvous: responsibility for the event space is partitioned among the set of brokers

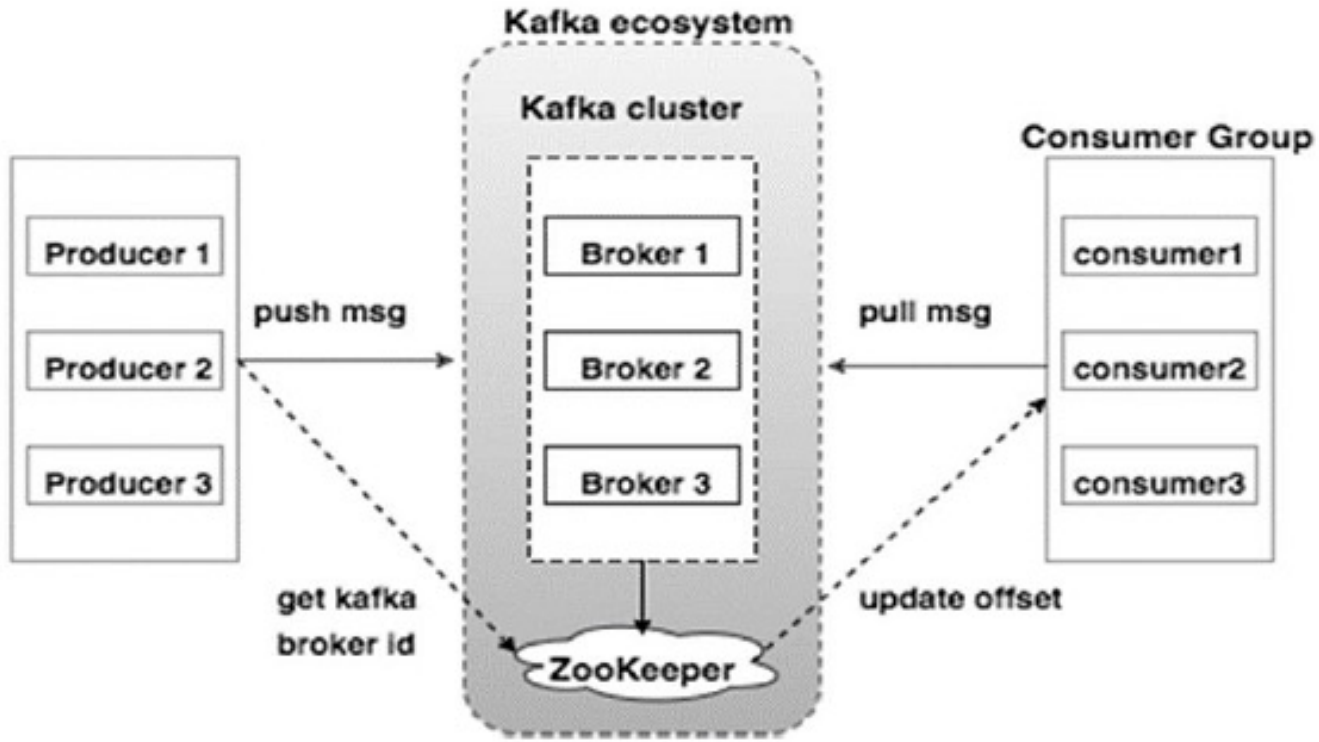
Some academic publish-subscribe systems

System (and further reading)	*Subscription model*	*Distribution model*	*Event routing*
CORBA Event Service (Chapter 8)	Channel-based	Centralized	-
TIB Rendezvouz [Oki *et al.* 1993]	Topic-based	Distributed	Ffiltering
Scribe [Castro *et al.* 2002b]	Topic-based	Peer-to-peer (DHT)	Rendezvous
TERA [Baldoni *et al.* 2007]	Topic-based	Peer-to-peer	Informed gossip
Siena [Carzaniga *et al.* 2001]	Content-based	Distributed	Filtering
Gryphon [www.research.ibm.com]	Content-based	Distributed	Filtering
Hermes [Pietzuch and Bacon 2002]	Topic- and content-based	Distributed	Rendezvous and filtering
MEDYM [Cao and Singh 2005]	Content-based	Distributed	Flooding
Meghdoot [Gupta *et al.* 2004]	Content-based	Peer-to-peer	Rendezvous
Structure-less CBR [Baldoni *et al.* 2005]	Content-based	Peer-to-peer	Informed gossip

Some modern P/S systems

- Apache Kafka
- Apache Pulsar
- Amazon Kinesis
- Google Pub/Sub
- Microsoft Event Hubs

Apache Kafka



- Platform for handling real-time data feeds, written in Java and Scala
- ZooKeeper supports high availability through redundant services
- Multiple brokers to maintain load balance
- Brokers are stateless: they use ZooKeeper for maintaining their cluster state; ZK is also used to notify producer and consumer about new or crashed broker

Kafka use cases: Linkedin and Netflix

Apache Kafka at Large Scale → No need to do a POC

LinkedIn (2018)

Operation Challenges

- The scale of Kafka deployment @LinkedIn
  - 2,100+ brokers
  - ~ 60,000 topics
  - > 4.5 trillion messages / day
- Huge operation overhead
  - Hardware failures are norm
  - Workload skews

Strata DATA CONFERENCE

NETFLIX (2018)

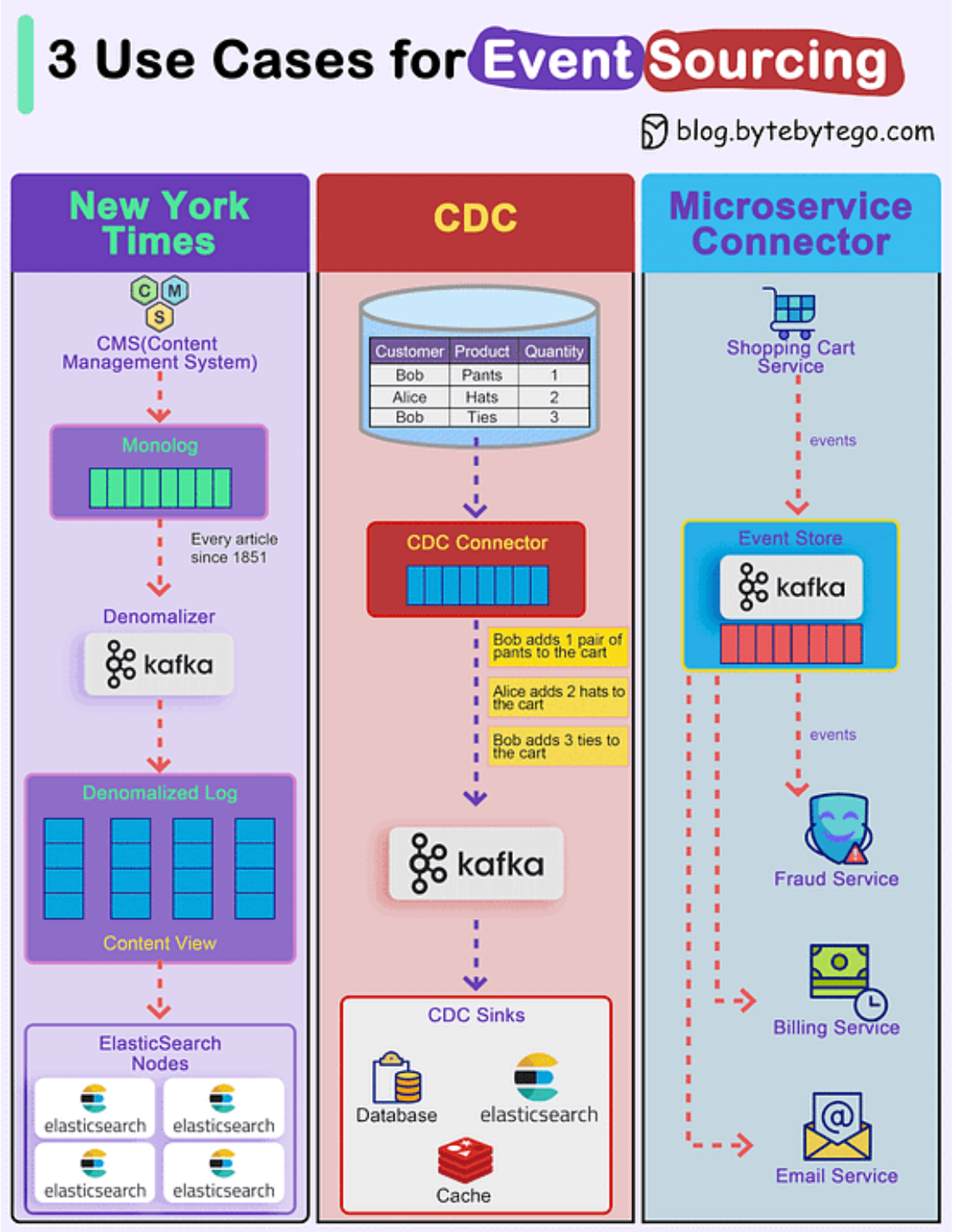
Kafka @ Netflix Scale

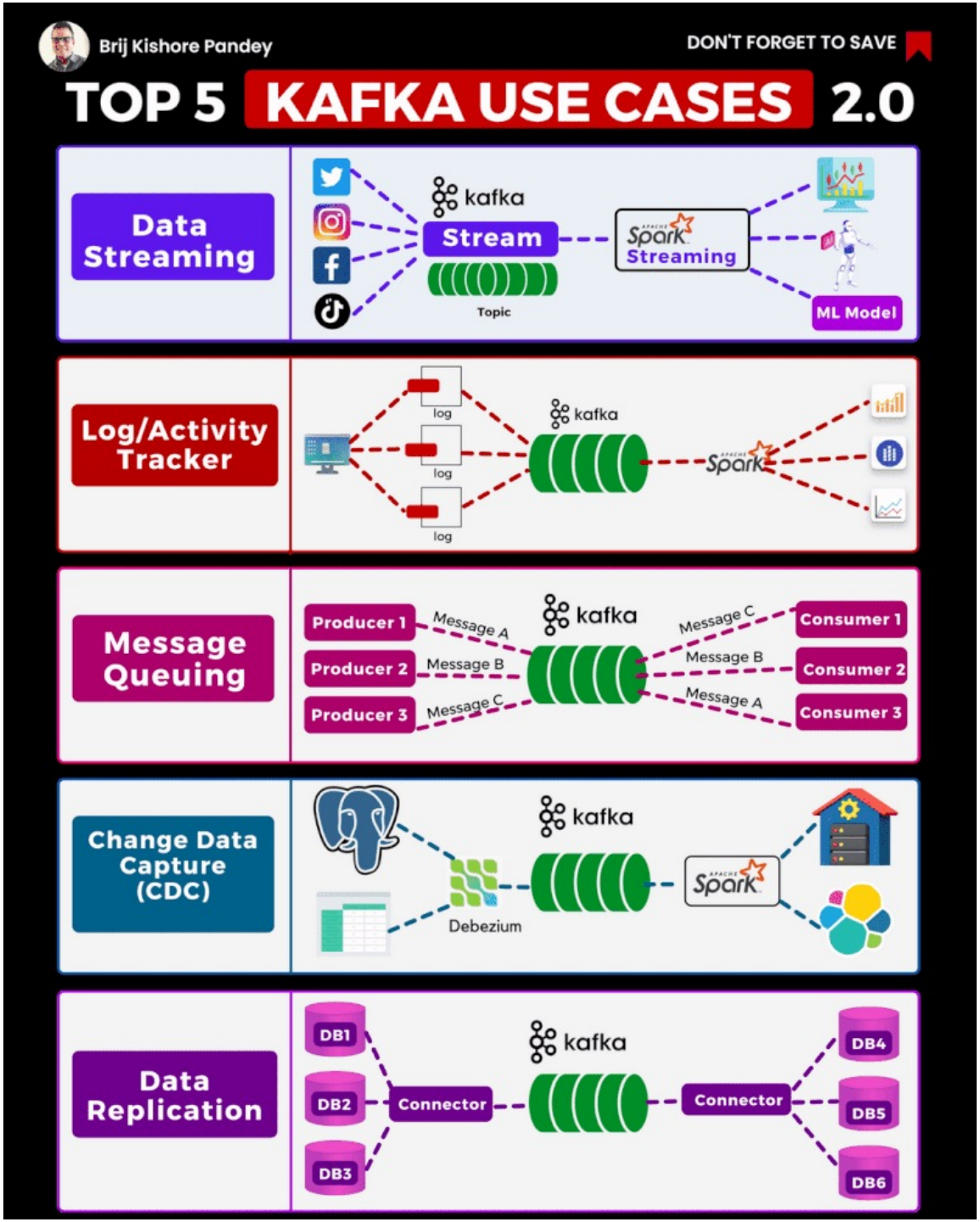
- 4,000+ brokers and ~50 clusters in 3 AWS regions
- > 1 Trillion messages per day
- At peak (New Years Day 2018)
  - 2.2 trillion messages (1.3 trillion unique)
  - 6 Petabytes

QCon

https://conferences.oreilly.com/strata/strata-ca/public/schedule/detail/63921
https://qconlondon.com/london2018/presentation/cloud-native-and-scalable-kafka-architecture

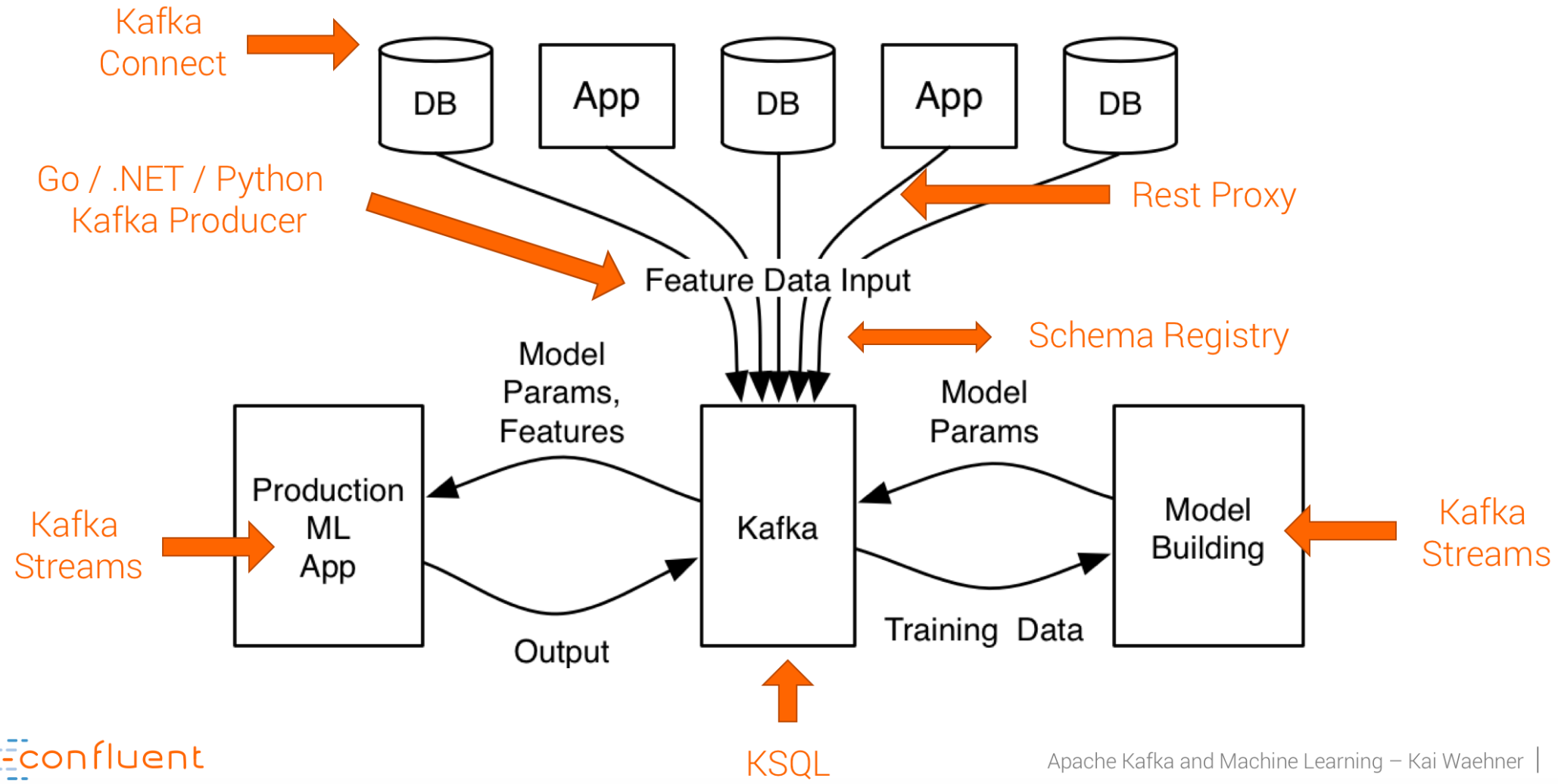
1. New York Times
   The newspaper website stores every article, image, and byline since 1851 in an event store. The raw data is then denormalized into different views and fed into different ElasticSearch nodes for website searches.
2. CDC (Change Data Capture)
   A CDC connector pulls data from the tables and transforms it into events. These events are pushed to Kafka and other sinks consume events from Kafka.
3. Microservice Connector
   We can use event event-sourcing paradigm for transmitting events among microservices. For example, the shopping cart service generates events for adding or removing items from the cart. Kafka broker acts as the event store, and other services including the fraud service, billing service, and email service consume events from the event store. Since events are shared each service can determine the domain model on its own




Brij Kishore Pandey
DON'T FORGET TO SAVE
TOP 5 KAFKA USE CASES 2.0
Data Streaming
kafka
Stream
Topic
Spark Streaming
ML Model
Log/Activity Tracker
log
log
log
kafka
Spark
Message Queuing
Producer 1
Producer 2
Producer 3
Message A
Message B
Message C
kafka
Message C
Message B
Message A
Consumer 1
Consumer 2
Consumer 3
Change Data Capture (CDC)
kafka
Debezium
Spark
Data Replication
DB1
DB2
DB3
kafka
Connector
Connector
DB4
DB5
DB6

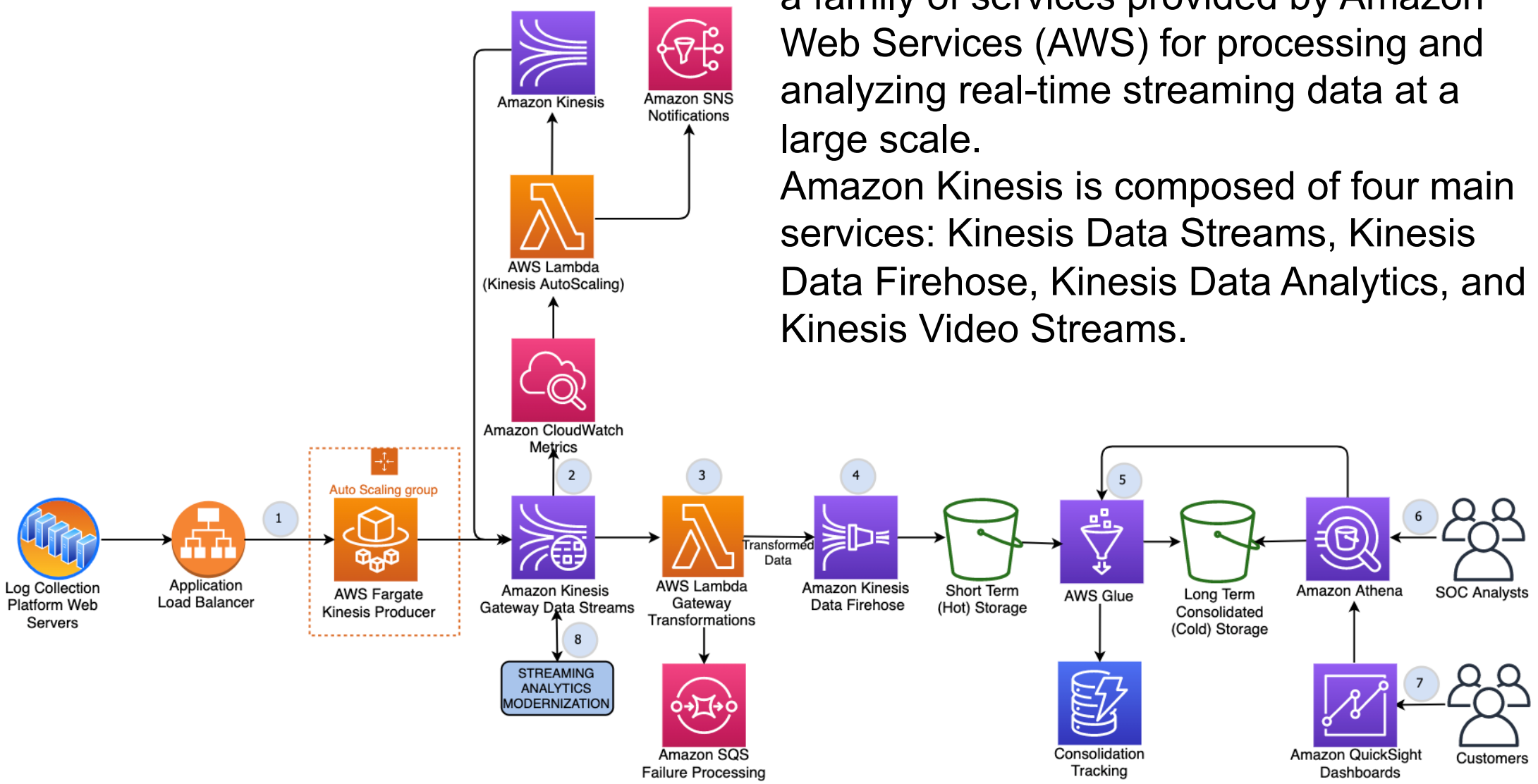
Kafka use cases: an architecture including Machine Learning

Apache Kafka's Open Source Ecosystem as Infrastructure for Machine Learning



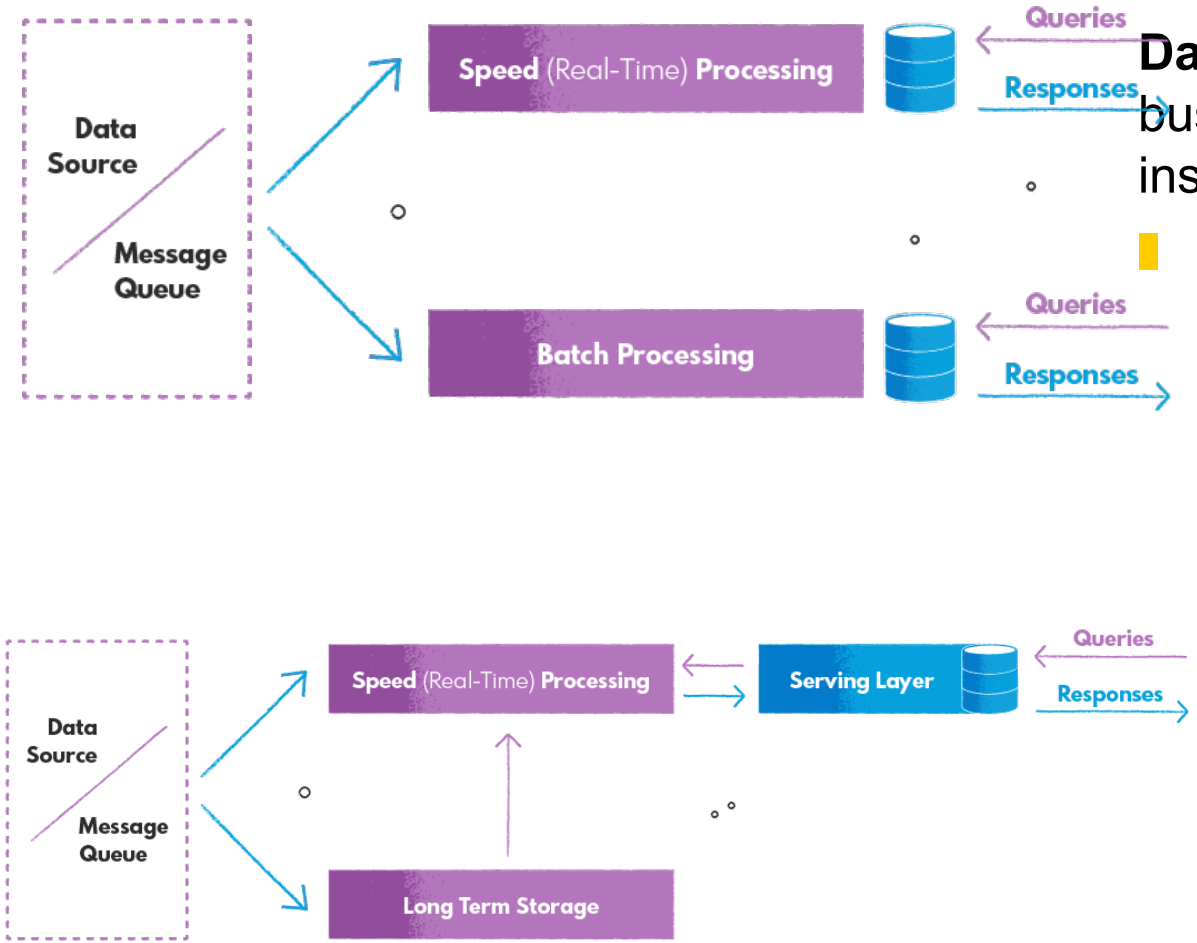
Amazon Kinesis

a family of services provided by Amazon Web Services (AWS) for processing and analyzing real-time streaming data at a large scale.
Amazon Kinesis is composed of four main services: Kinesis Data Streams, Kinesis Data Firehose, Kinesis Data Analytics, and Kinesis Video Streams.



https://aws.amazon.com/it/blogs/architecture/designing-a-high-volume-streaming-data-ingestion-platform-natively-on-aws/

Stream data processing: Lambda and Kappa architectures

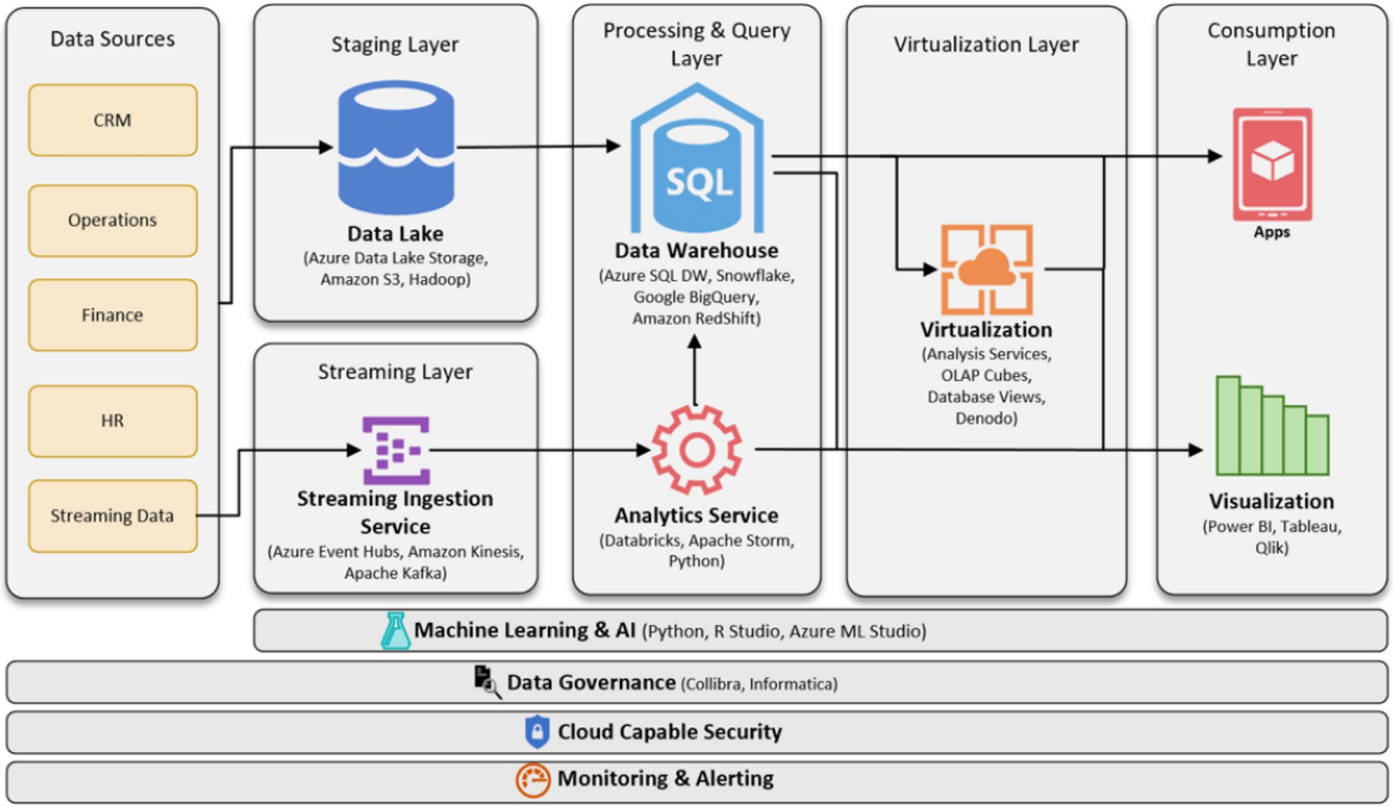


Data processing architectures help businesses analyze and extract valuable insights from their data in real-time.

- **Lambda architecture** uses separate batch and stream processing systems, making it scalable and fault-tolerant but complex to set up and maintain (as it duplicates processing logic).
- **Kappa architecture** simplifies the pipeline with a single stream processing system as it treats all data as streams, providing flexibility and ease of maintenance, but requires anyway experience in stream processing and distributed systems

https://nexocode.com/blog/posts/lambda-vs-kappa-architecture/

Lambda



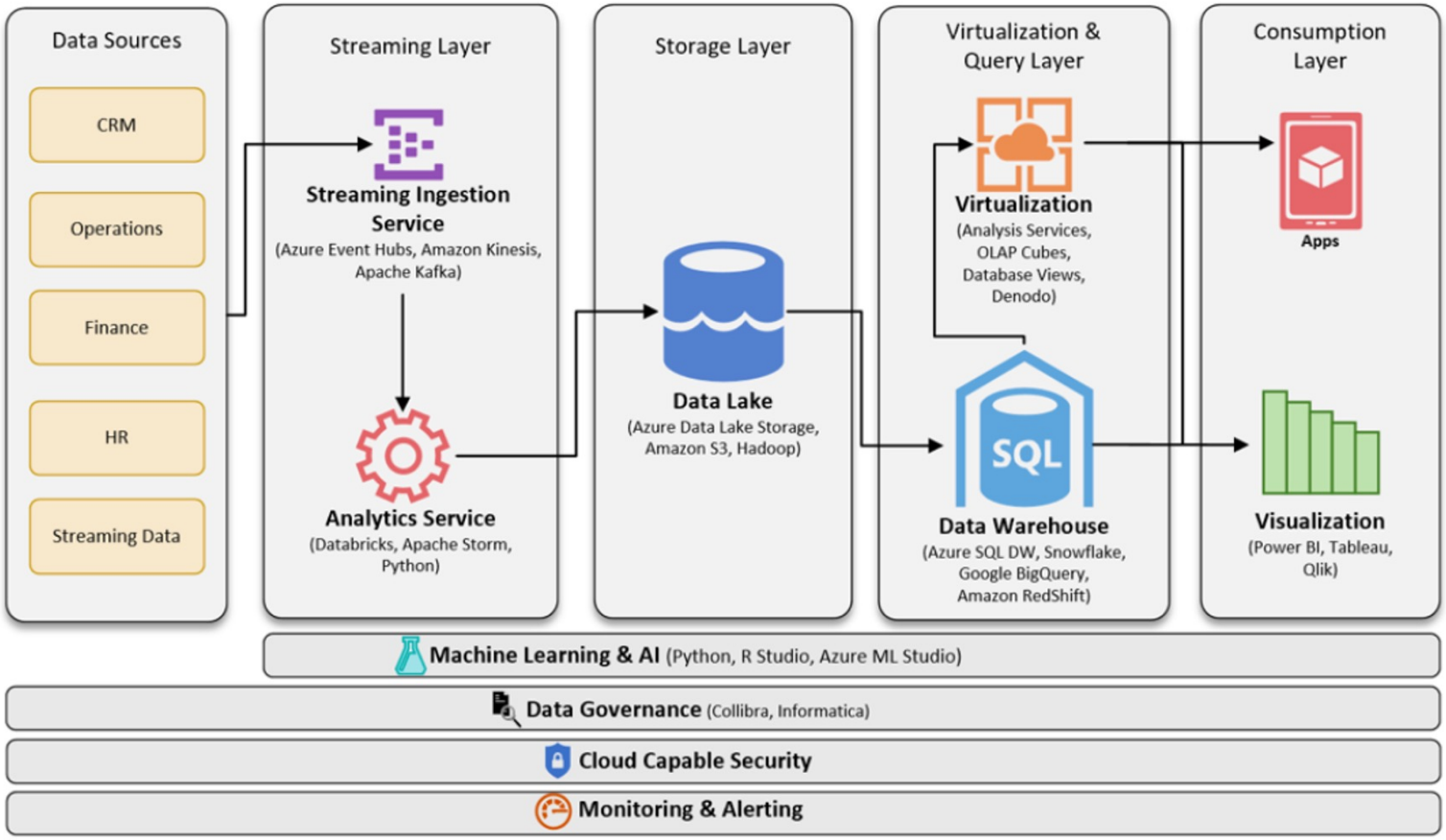
Lambda architecture example

Source: Credera

The Lambda architecture attempts to balance latency, throughput, and fault-tolerance by using batch processing to provide comprehensive and accurate views of batch data, while simultaneously using real-time stream processing to provide views of online data.

https://www.qlik.com/blog/lambda-or-kappa-the-need-for-a-new-data-processing-architecture

Kappa



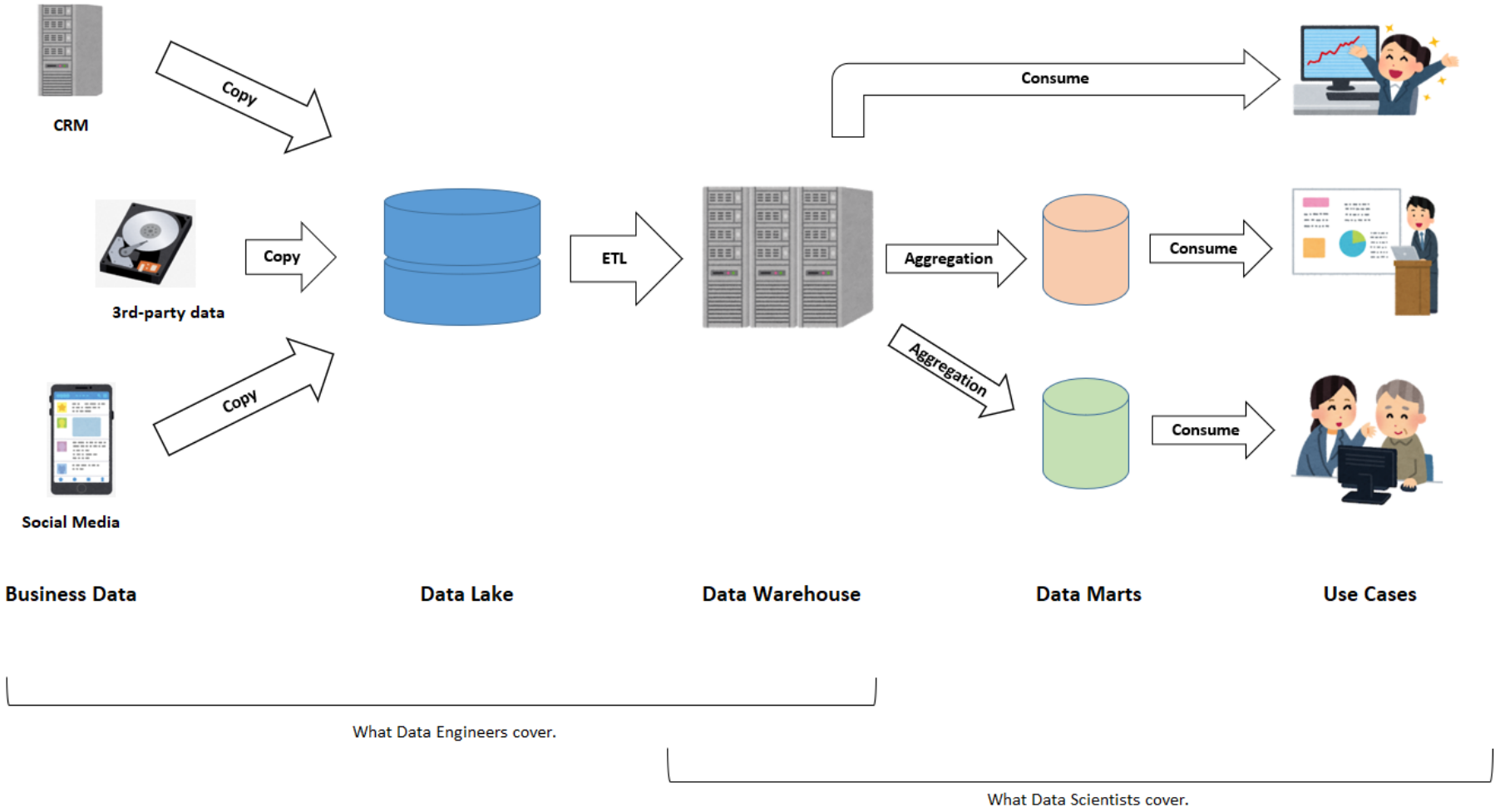
Kappa architecture example

Source: Credera

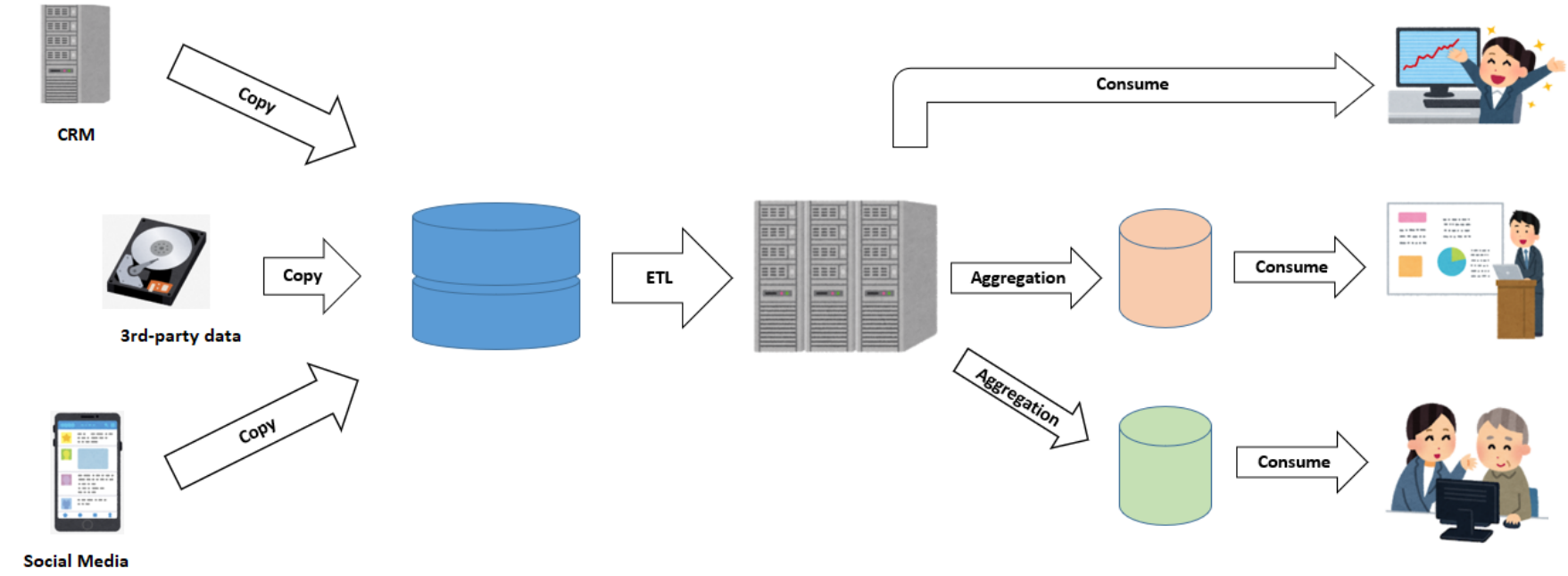
The Kappa architecture solves the redundant part of the Lambda architecture. It is designed with the idea of replaying data. This architecture avoids maintaining two different code bases for the batch and speed layers.

https://www.qlik.com/blog/lambda-or-kappa-the-need-for-a-new-data-processing-architecture

Intermezzo: architectures for data analytics



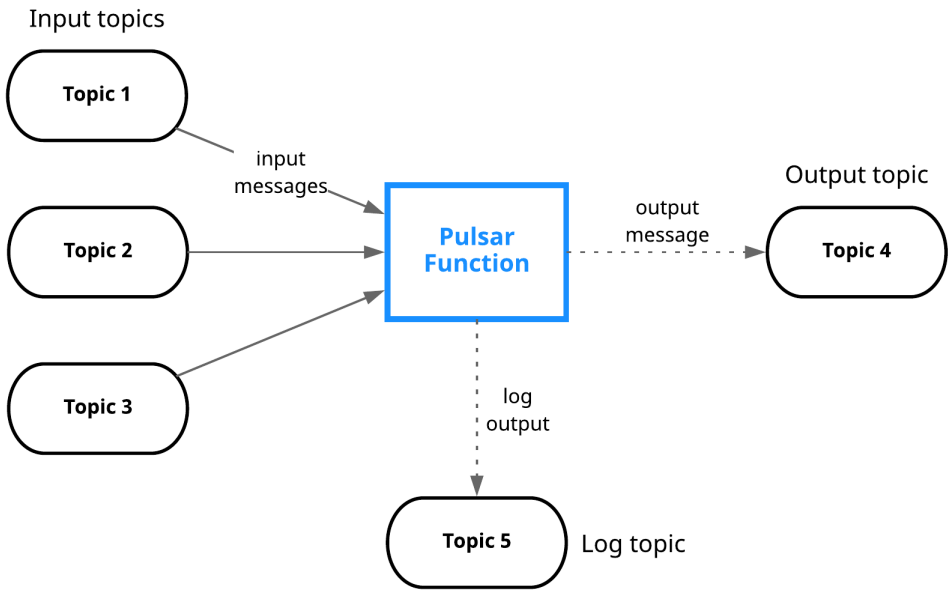
Intermezzo: architectures for data analytics



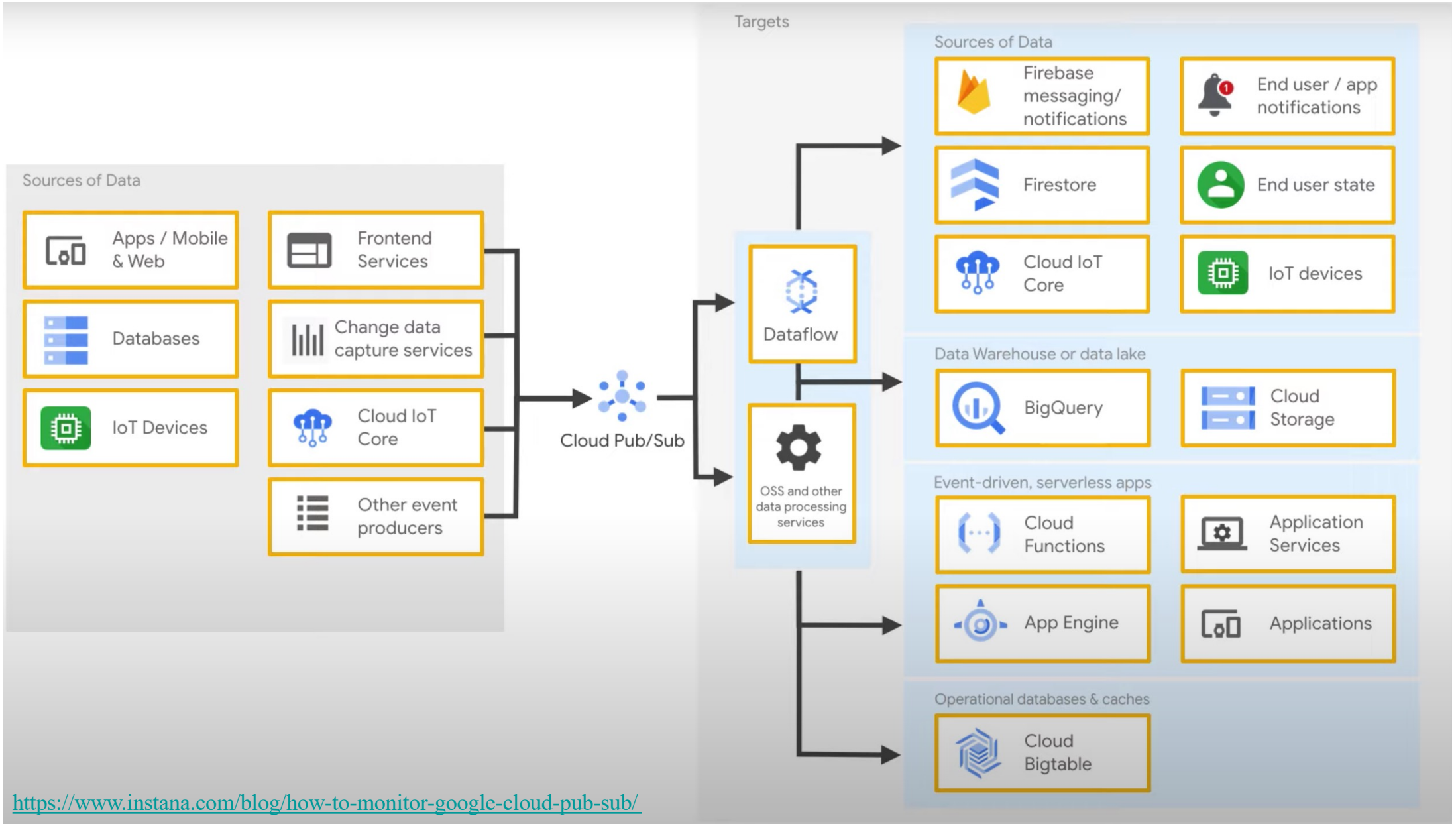
	Most Important Use	Who Builds This?	Who Uses This?	Data Velocity	Maintenance Cost
Data Lake	Keeping data stored. Transaction-oriented.	Data Architect, Data Engineer	Data Engineer, Data Scientist	High real-time	$$$$$
Data Warehouse	Keeping data available in structured format and managed. Analytic-oriented.	Data Architect, Data Engineer, Data Scientist	Data Scientist, Business Expert	Medium real-time/regular	$$$
Data Mart	Cherry pick data for the use of specific line of business. Reporting-oriented.	Data Scientist, Business Expert	Business Expert	Low regular	$

Apache Pulsar

- Pulsar is a cloud-native, distributed messaging and streaming platform created at Yahoo! and now a top-level Apache sw Foundation project
- Pulsar Functions are inspired by stream processing engines :
  - Apache Storm, Heron, and Flink
  - "Serverless" and "Function as a Service" (FaaS) cloud platforms like Amazon WS Lambda, Google Cloud Functions, and Azure Cloud Functions



Google pub/sub



https://www.instana.com/blog/how-to-monitor-google-cloud-pub-sub/

Message queues

- The programming model offered by message queues is simple: it offers an approach to communication through queues.
- In particular, producer processes can send messages to a specific queue; other (consumer) processes can then receive messages from this queue.

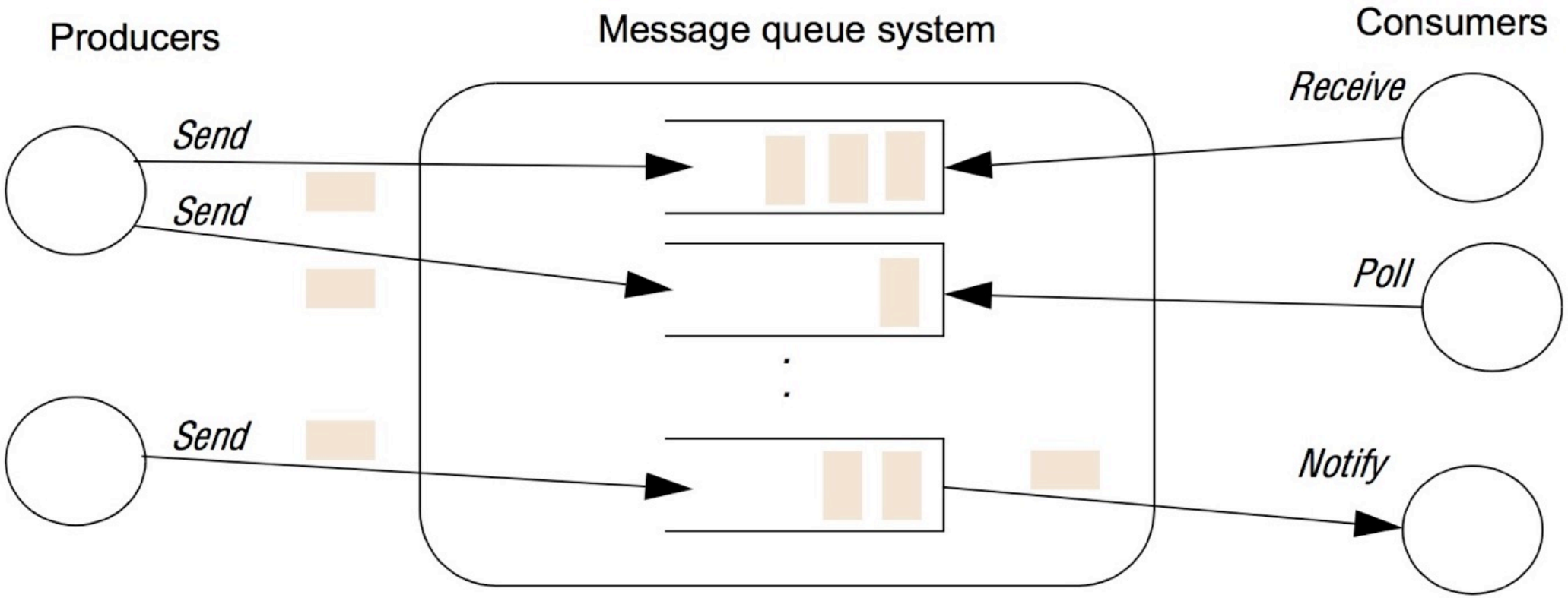
Three styles of `receive` are generally supported:

I. a blocking *receive*, which will block until an appropriate message is available;

II. a non-blocking *receive* (a polling operation), which will check the status of the queue and return a message if available, or a not available indication otherwise;

III. a *notify* operation, which will issue an event notification when a message is available in the associated queue.

The message queue paradigm

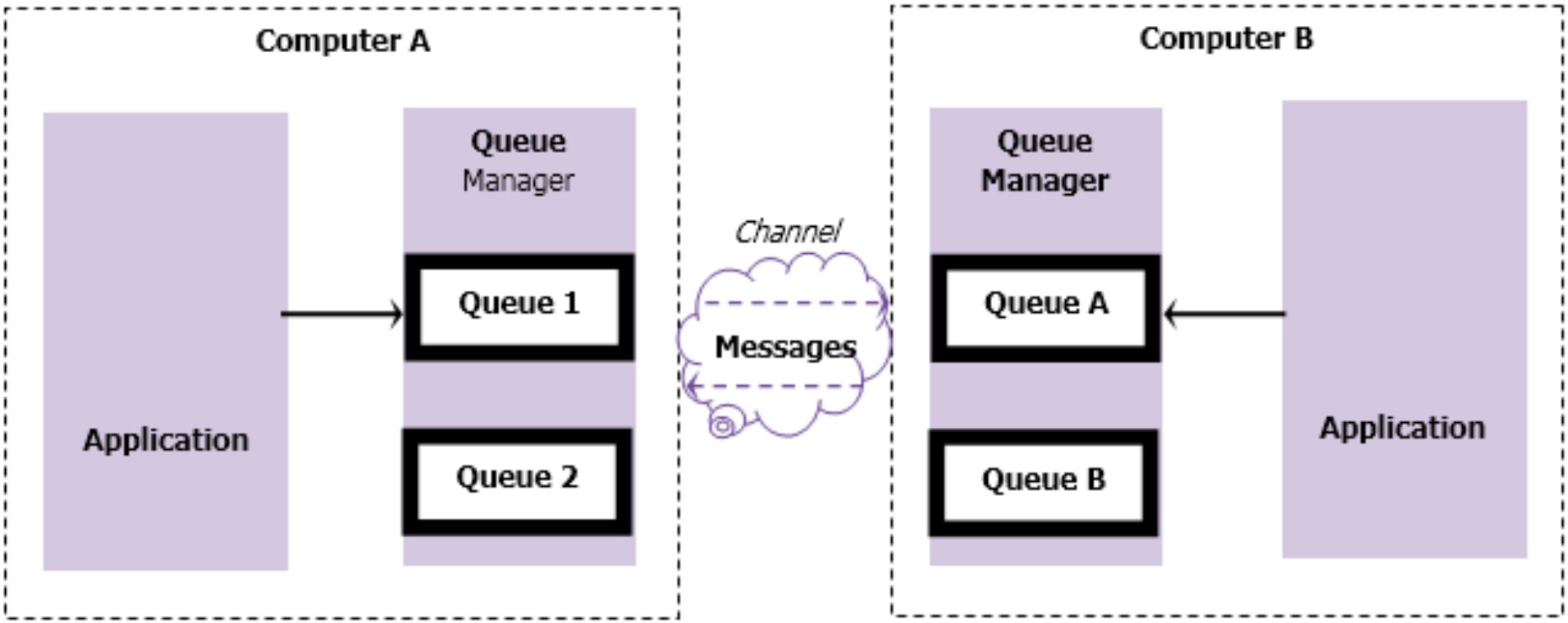


IBM WebSphere MQ

- WebSphere MQ is a middleware developed by IBM based on the concept of message queues, offering an indirection between senders and receivers of messages
- Client applications accessing a queue manager may reside on the same physical server.
- More generally, though, they will be on different machines and must then communicate with the queue manager through what is known as a client channel
- Client channels adopt the familiar concept of a *proxy*

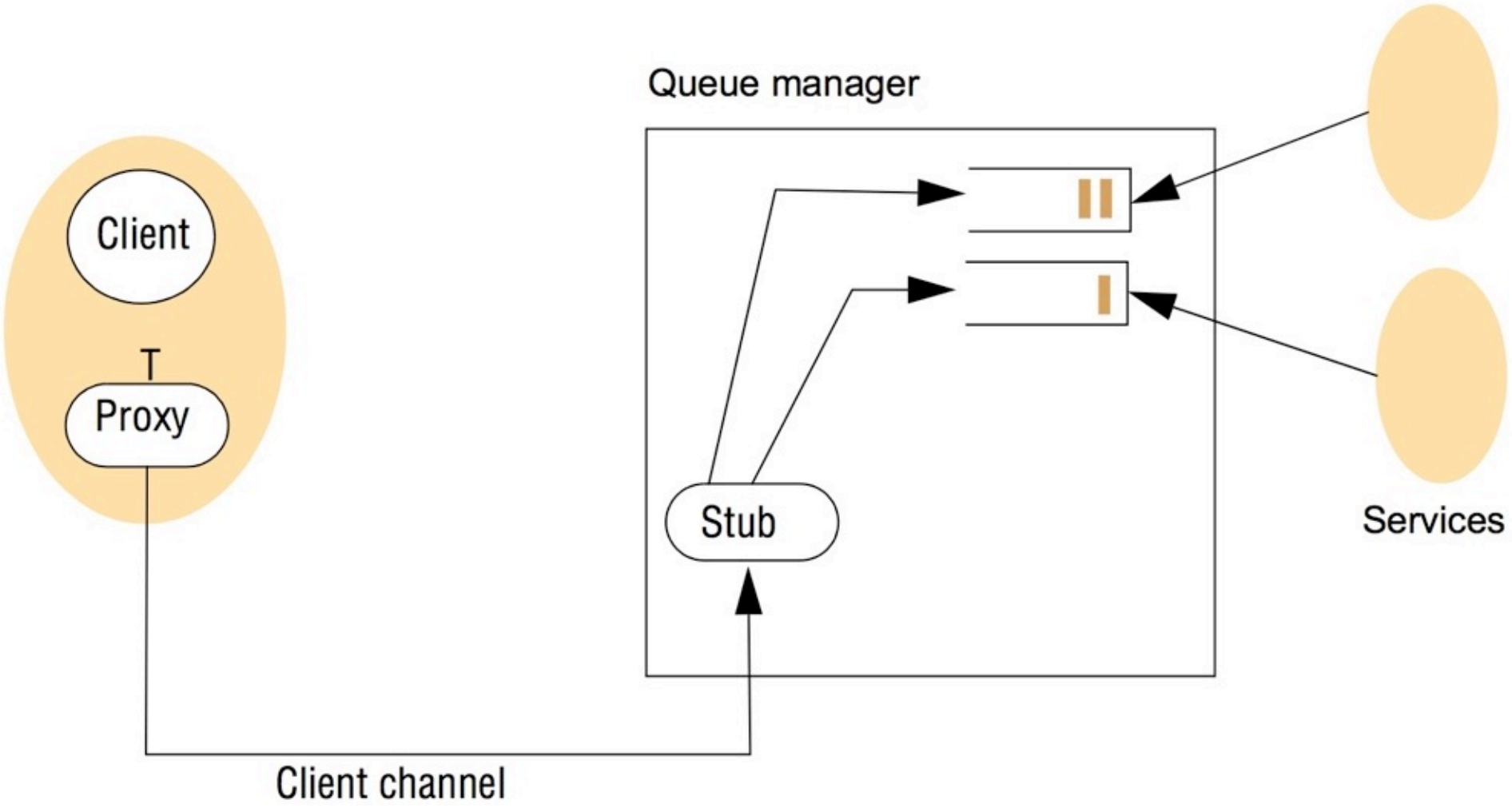
https://www.ibm.com/docs/en/ibm-mq/7.5

WebSphere MQ architecture

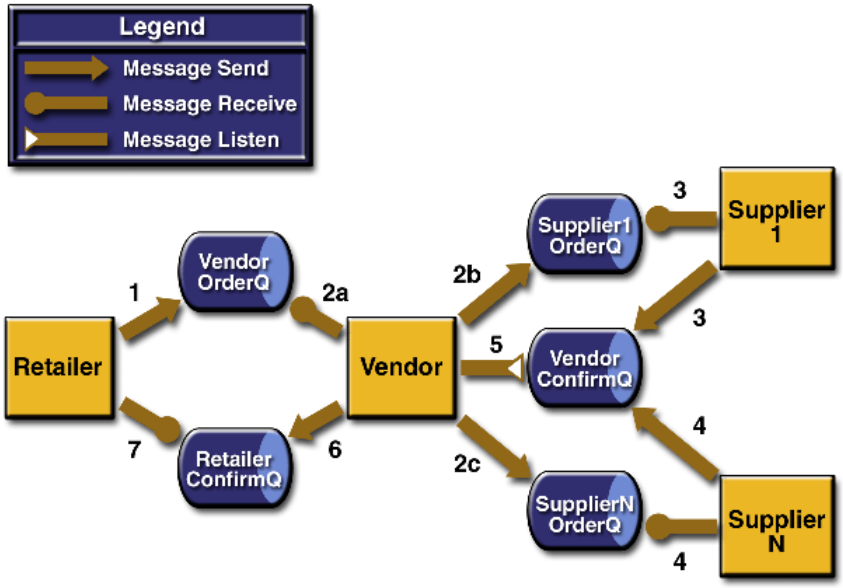


A queue manager is a program that provides messaging services to applications. The queue manager ensures that messages are sent to the correct queue or are routed to another queue manager.
A queue is a container for messages. Channels are of two types, unidirectional or bidirectional.

A simple networked topology in WebSphere MQ



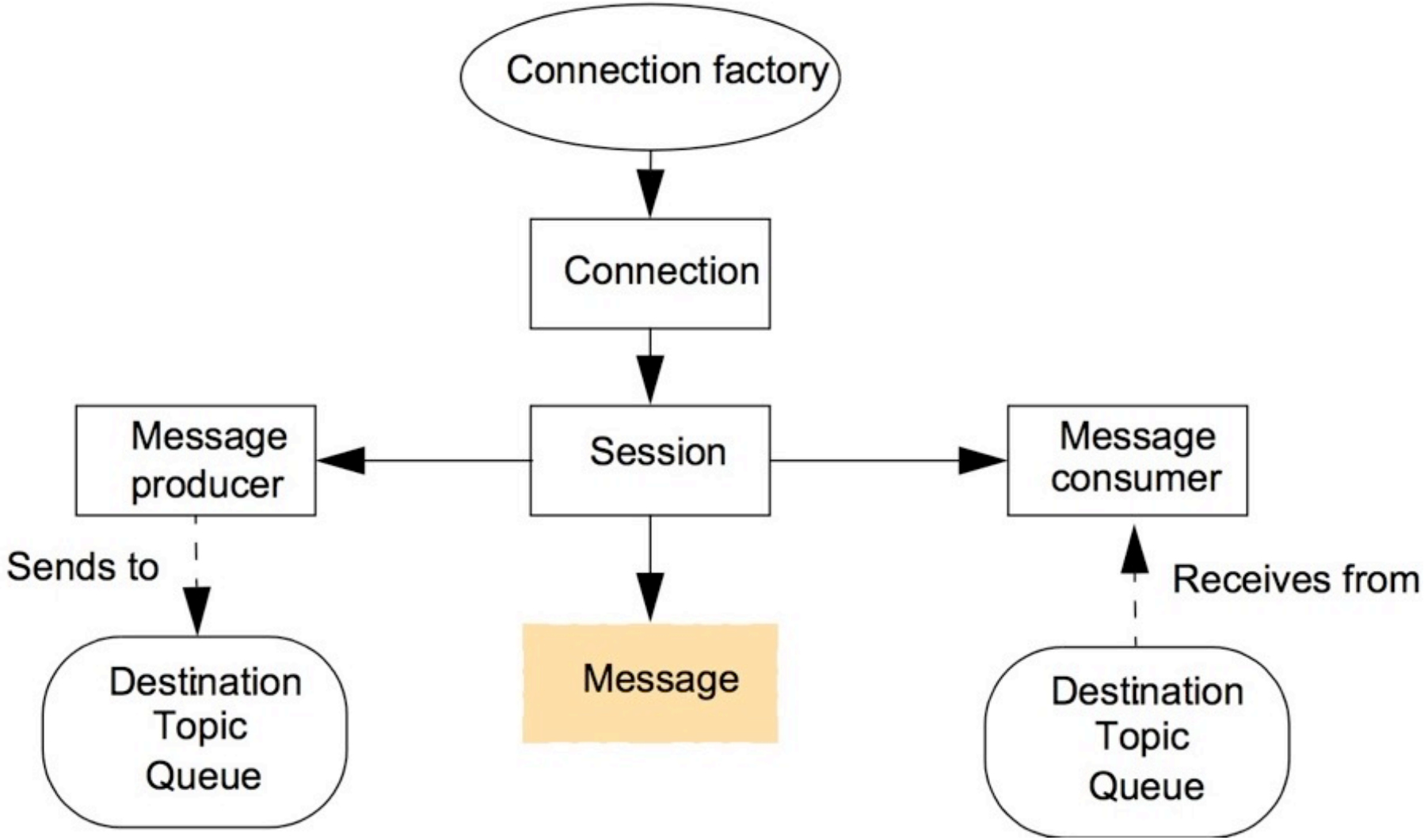
Jakarta Message Service JMS (was Java Message Service)



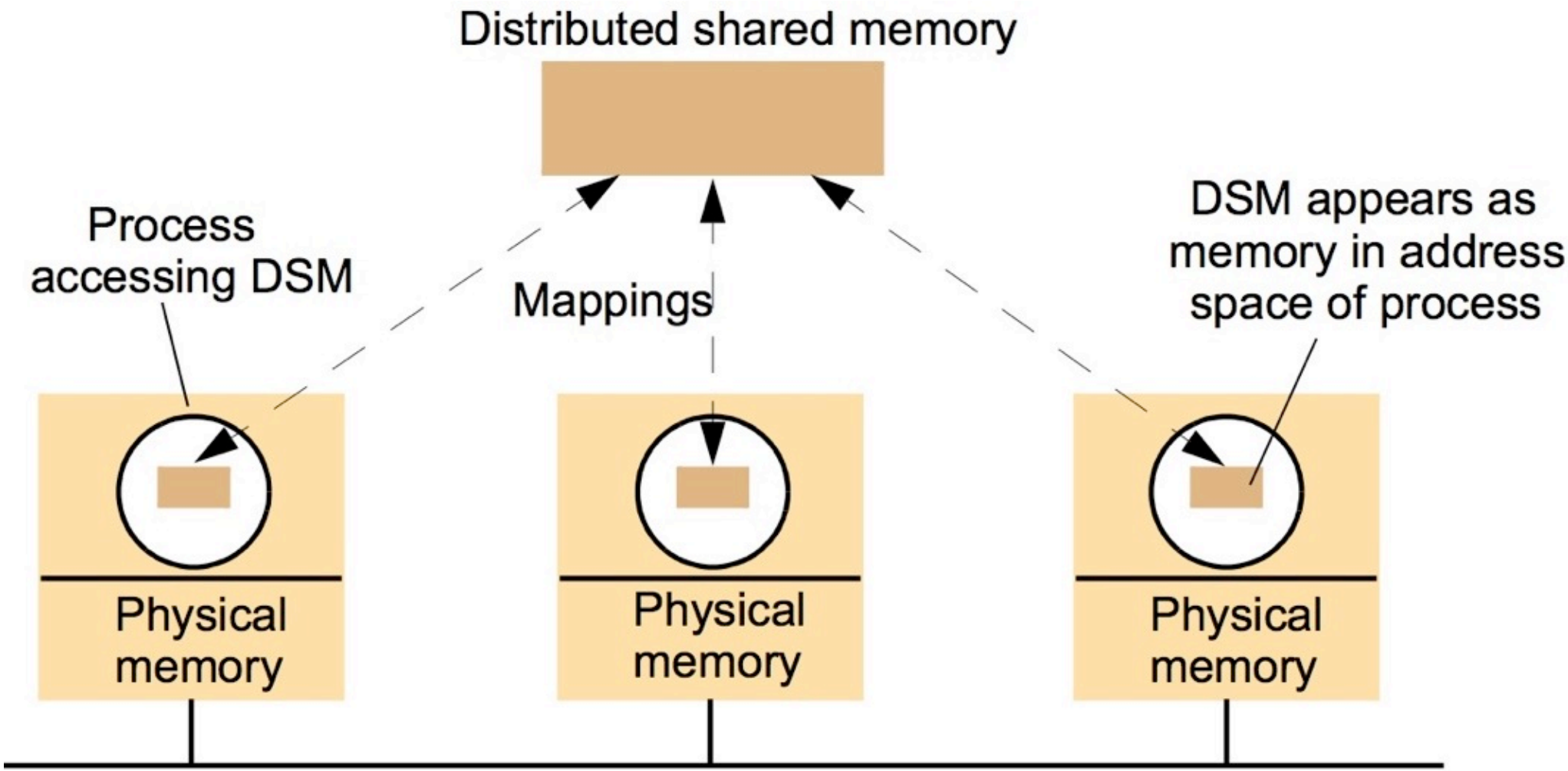
https://www.huihoo.org/jfox/jfoxmq/jms_tutorial/client_samples.html

1. A retailer sends a message to the vendor order queue, ordering a quantity of computers, and waits for the vendor's reply.
2. The vendor receives the retailer's order message and places an order message into each of its suppliers' order queues, all in one transaction. This JMS transaction combines one synchronous receive with multiple sends.
3. One supplier receives the order from its order queue, checks its inventory, and sends the items ordered to the destination named in the order message's JMSReplyTo field. If it does not have enough in stock, the supplier sends what it has. The synchronous receive and the send take place in one JMS transaction.
4. The other supplier receives the order from its order queue, checks its inventory, and sends the items ordered to the destination named in the order message's JMSReplyTo field. If it does not have enough in stock, the supplier sends what it has. The synchronous receive and the send take place in one JMS transaction.
5. The vendor receives the replies from the suppliers from its confirmation queue and updates the state of the order. Messages are processed by an asynchronous message listener; this step illustrates using JMS transactions with a message listener.
6. When all outstanding replies are processed for a given order, the vendor sends a message notifying the retailer whether it can fulfill the order.
7. The retailer receives the message from the vendor.

The programming model offered by JMS



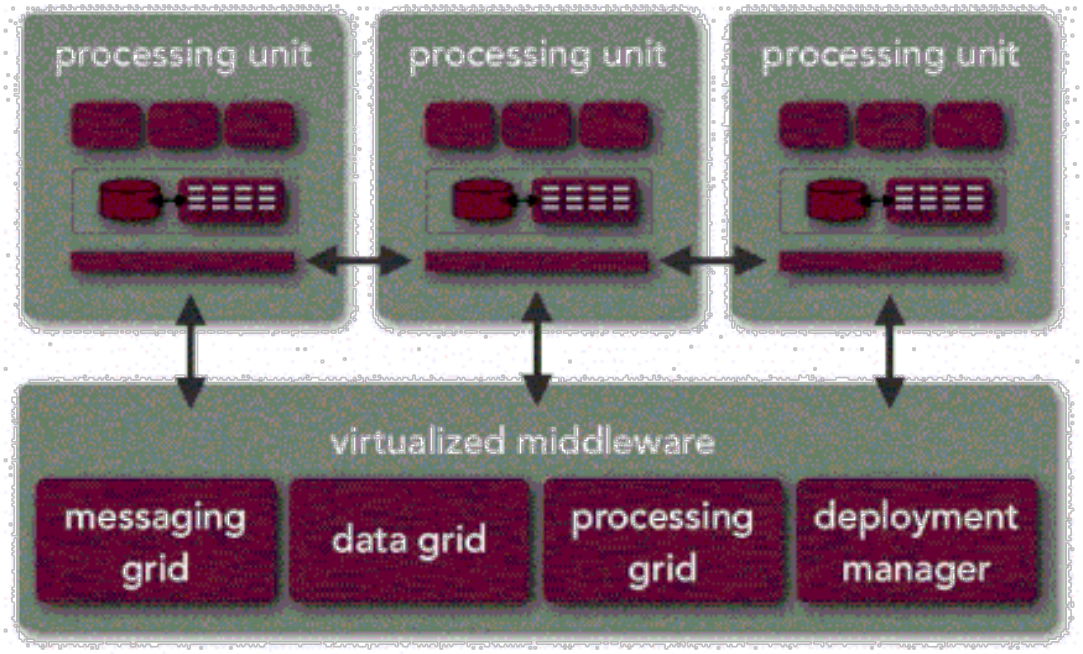
The Distributed Shared Memory (DSM) abstraction



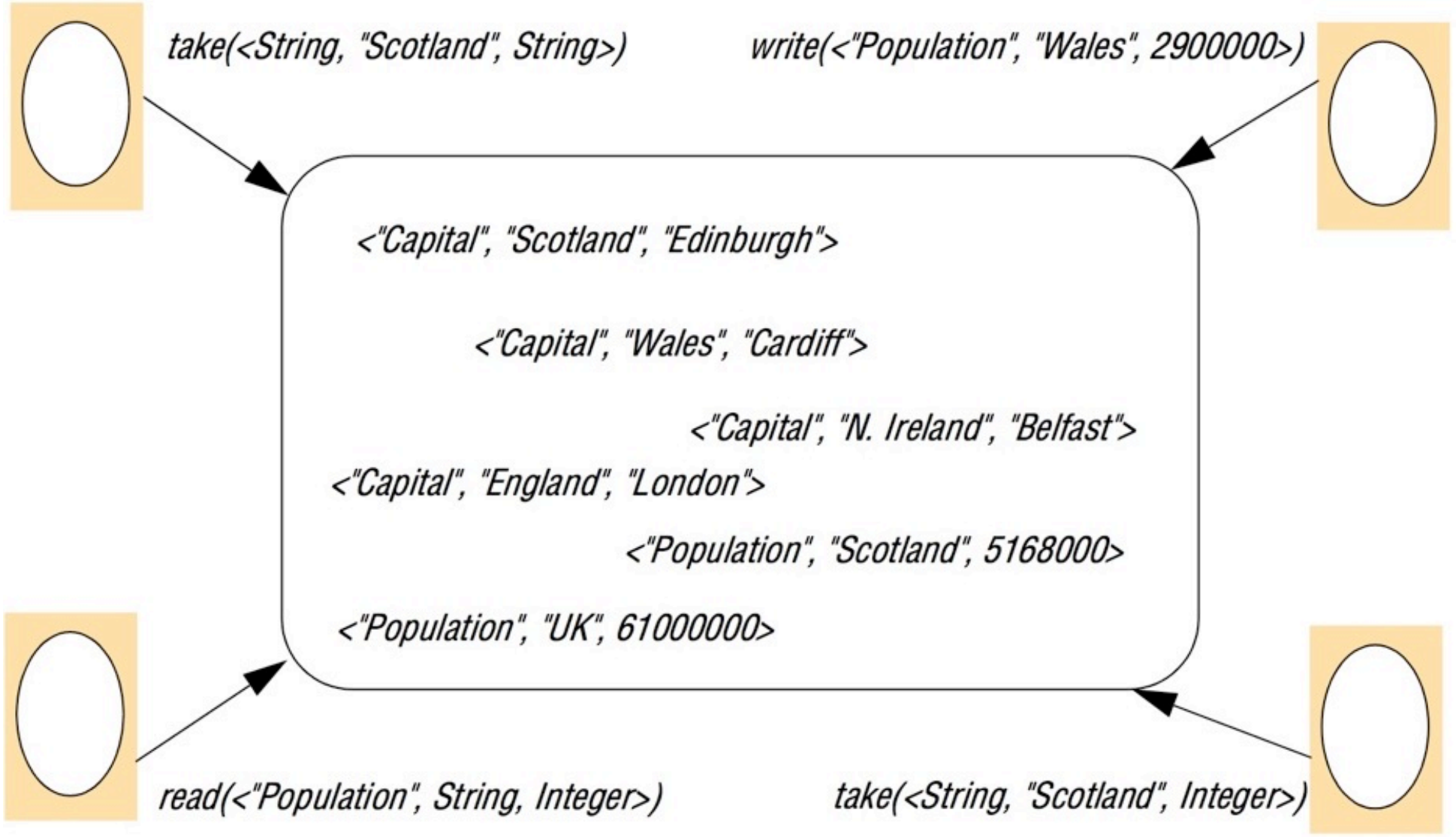
Processes access DSM by reads and updates to what appears to be ordinary memory within their address space; an underlying runtime system ensures transparently that processes executing at different computers observe the updates made by one another

(Shared) Space based architectures

- A Space-based architecture (SBA) is an approach to distributed computing where the active components interact with each other by exchanging tuples or entries via one or more shared spaces.
- This is contrasted with the more common Message queuing service approaches, where the components interact with each other by exchanging messages via a message broker.
- In a sense, both approaches exchange messages with some central agent, but how they exchange messages is different.



The tuple space abstraction



Replication and the tuple space operations [Xu and Liskov 1989]

write

1. The requesting site multicasts the *write* request to all members of the view;
2. On receiving this request, members insert the tuple into their replica and acknowledge this action;
3. Step 1 is repeated until all acknowledgements are received.

read

1. The requesting site multicasts the *read* request to all members of the view;
2. On receiving this request, a member returns a matching tuple to the requestor;
3. The requestor returns the first matching tuple received as the result of the operation (ignoring others); 4. Step 1 is repeated until at least one response is received.

continued on next slide

(continued)
Replication and the tuple space operations [Xu and Liskov 1989]

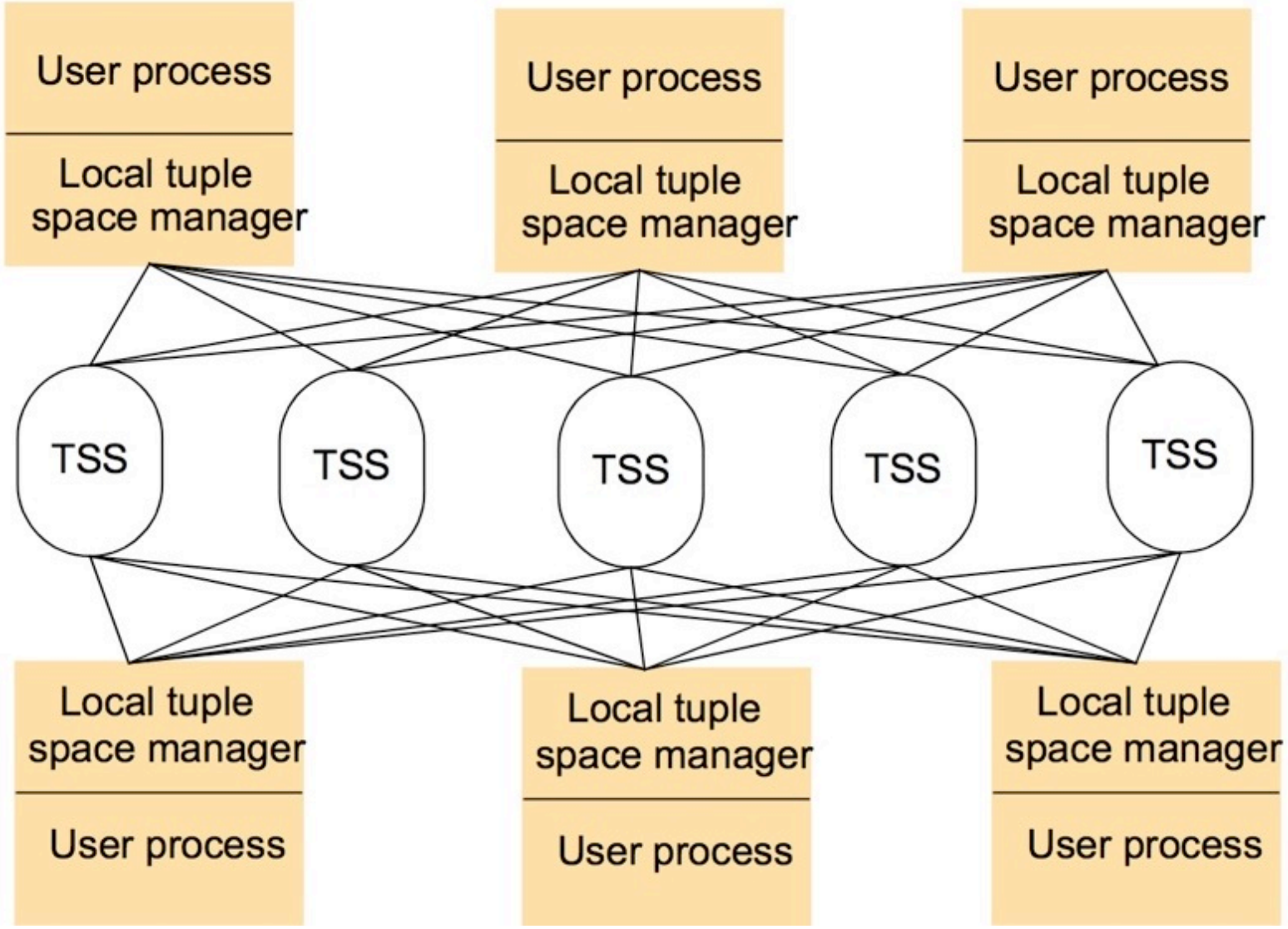
take *Phase 1: Selecting the tuple to be removed*

1. The requesting site multicasts the *take* request to all members of the view;
2. On receiving this request, each replica acquires a lock on the associated tuple set and, if the lock cannot be acquired, the *take* request is rejected;
3. All accepting members reply with the set of all matching tuples;
4. Step 1 is repeated until all sites have accepted the request and responded with their set of tuples and the intersection is non-null;
5. A particular tuple is selected as the result of the operation (selected randomly from the intersection of all the replies);
6. If only a minority accept the request, this minority are asked to release their locks and phase 1 repeats.

Phase 2: Removing the selected tuple

1. The requesting site multicasts a *remove* request to all members of the view citing the tuple to be removed;
2. On receiving this request, members remove the tuple from their replica, send an acknowledgement and release the lock;
3. Step 1 is repeated until all acknowledgements are received.

Partitioning in the York Linda Kernel

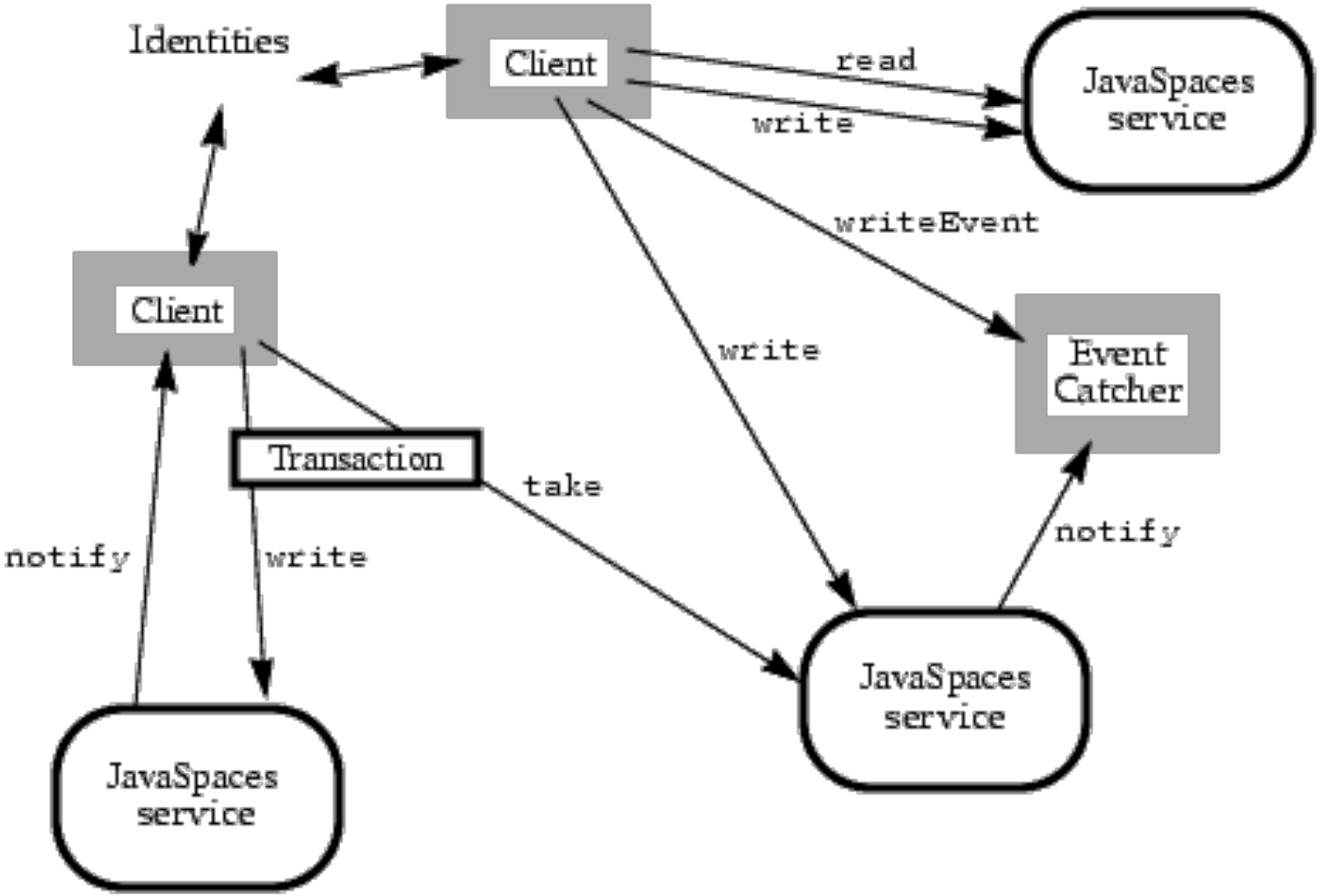


JavaSpaces

The JavaSpaces service specification lists the following design goals for the JavaSpaces technology:

- It is a platform that simplifies the design and implementation of distributed computing systems.
- The client side has few classes, both to keep the client simple and to speed the downloading of client classes.
- The client side has a small footprint because it will run on computers with limited local memory.
- A variety of implementations is possible.
- It is possible to create a replicated JavaSpaces service.

JavaSpaces application (example)



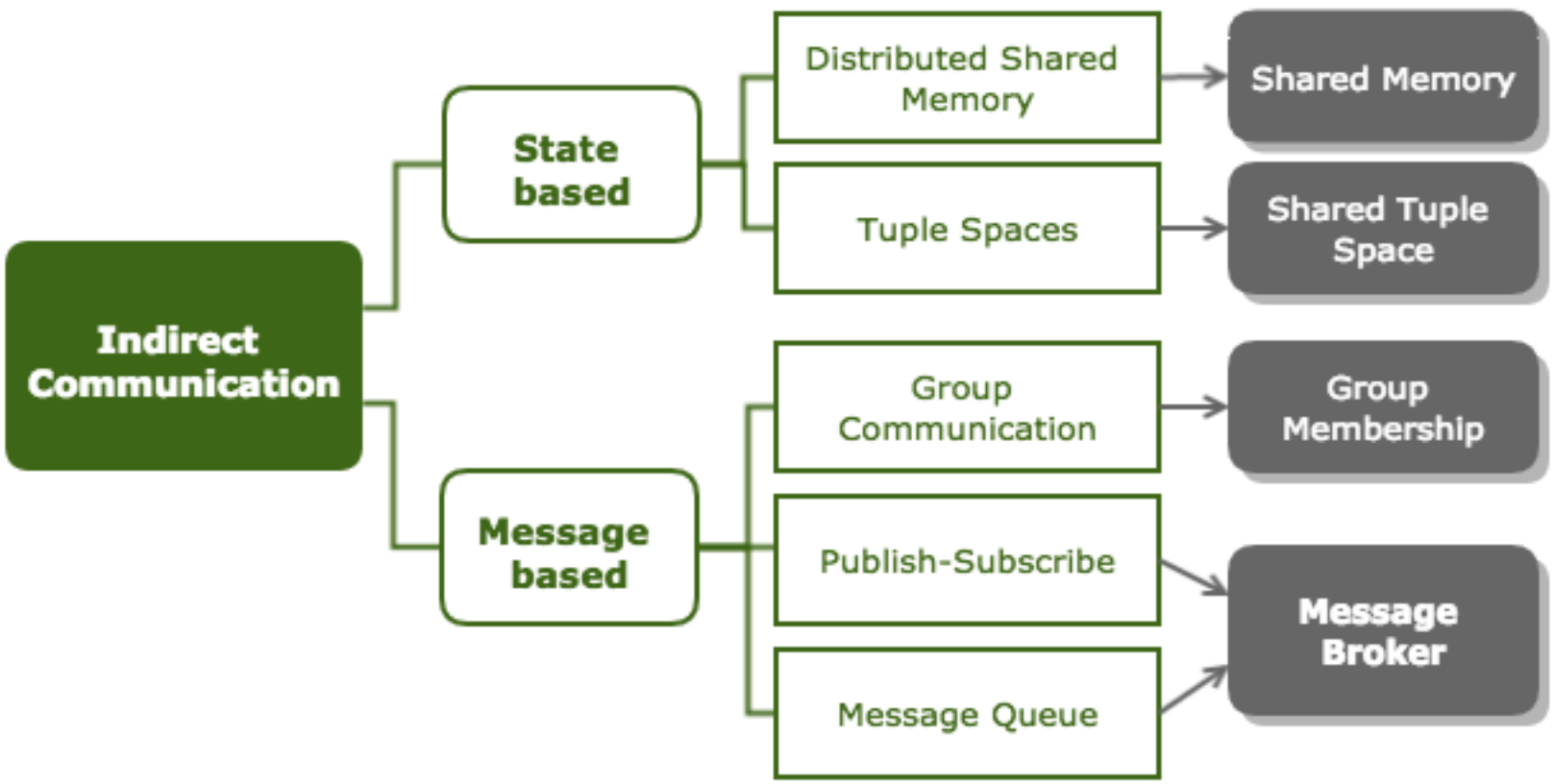
Jini (Apache River)

- Jini is a software architecture supporting distributed systems as modular cooperating services
- River is the implementation of Jini service oriented architecture.
- It defines a programming model which both exploits and extends Java technology to enable the construction of distributed systems.
- Apache River has been terminated in 2022

Summary of indirect communication styles

	Groups	*Publish-subscribe systems*	*Message queues*	*DSM*	*Tuple spaces*
Space-uncoupled	Yes	Yes	Yes	Yes	Yes
Time-uncoupled	Possible	Possible	Yes	Yes	Yes
Style of service	Communication-based	Communication-based	Communication-based	State-based	State-based
Communication pattern	1-to-many	1-to-many	1-to-1	1-to-many	1-1 or 1-to-many
Main intent	Reliable distributed computing	Information dissemination or EAI; mobile and ubiquitous systems	Information dissemination or EAI; commercial transaction processing	Parallel and distributed computation	Parallel and distributed computation; mobile and ubiquitous systems
Scalability	Limited	Possible	Possible	Limited	Limited
Associative	No	Content-based publish-subscribe only	No	No	Yes

Summary



https://dzone.com/articles/message-brokers-in-indirect-communication-paradigm

Conclusions

- The three techniques: *groups*, *publish-subscribe*, and *message queues* offer a programming model that emphasizes communication (through messages or events), whereas *distributed shared memory* and *tuple spaces* offer a state-based abstraction.
- This is a fundamental difference in terms of scalability; the communication-based abstractions can scale to very large systems with an appropriate routing infrastructure

Summary of some Apache open source middleware

Middleware	Main use cases	Originators
Apache Kafka	Message brokering, Pub/Sub	Linkedin,Netflix
Apache Pulsar	Clous message brokering, Pub/Sub	Yahoo
Apache RocketMQ	Message brokering, Pub/Sub	Alibaba
Apache Spark	Bigdata analytics, lakehouses	UC Berkeley
Apache Flink	Bigdata analytics, stream processing, rich API	Stratosphere
Apache Storm	Stream processing, mapreduce	Twitter
Apache Openwhisk	Serverless backend	Nimbella
Apache Wildfly	Java application server	RedHat
Apache NiFi	Visual data flow with routing and manipulation	US NSA
Apache Mesos	Cluster manager, load balancing, cloud	Ebay, twitter
Apache Karaf	Modulith runtime, microservice container	Apache
Apache Camel	Enterprise integration patterns, MOM	Apache
Apache River	Space based architectures	Apache

Exercise Getting Started with Apache Kafka

- **Requirements:**

1. Set up a Kafka Cluster:
    1. Install and configure Apache Kafka on your local machine or a cloud-based server.
    2. Start a Kafka broker (single-node cluster) with the default configuration.
2. Create a Topic:
    1. Using the Kafka command-line tools (kafka-topics.sh), create a new topic named "test-topic" with a single partition and replication factor of 1.
3. Implement a Kafka Producer:
    1. Write a Python or Java program to create a Kafka producer.
    2. The producer should send a series of messages to the "test-topic" at regular intervals (e.g., every second).
    3. The messages can be simple strings or JSON objects.
4. Implement a Kafka Consumer:
    1. Write a Python or Java program to create a Kafka consumer.
    2. The consumer should subscribe to the "test-topic" and continuously consume msgs.
    3. Print each received message to the console.

Hint – step by step guide

Setting Up Kafka:

Download and install Apache Kafka by following the official documentation.

1. Start a Kafka broker using the following command: kafka-server-start.sh config/server.properties

Creating a Topic:

1. Use the Kafka command-line tool to create the "test-topic":
2. kafka-topics.sh --create --bootstrap-server localhost:9092 --topic test-topic --partitions 1 --replication-factor 1

Implementing a Kafka Producer:

1. Write a Python or Java program that configures a Kafka producer.
2. Configure the producer to connect to the Kafka broker running on localhost.
3. Implement logic to send messages to the "test-topic" at regular intervals (e.g., using a loop and a timer).

Implementing a Kafka Consumer:

1. Write a Python or Java program that configures a Kafka consumer.
2. Configure the consumer to connect to the Kafka broker running on localhost.
3. Subscribe the consumer to the "test-topic."
4. Implement logic to continuously consume and print messages from the topic.

Testing:

1. Start your Kafka producer program and observe it sending messages to the "test-topic."
2. Start your Kafka consumer program and observe it receiving and printing messages from the "test-topic."
3. Verify that the producer and consumer are working correctly by checking the messages in the Kafka topic.