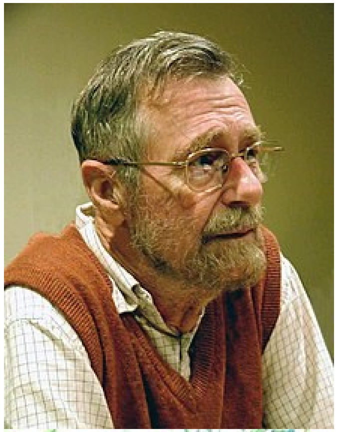
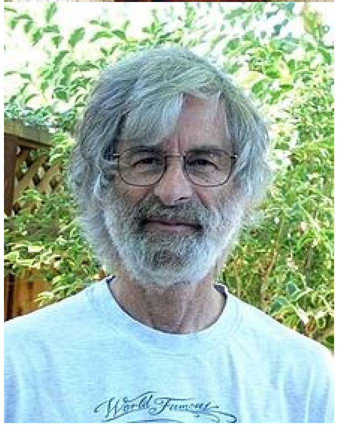
Distributed Systems

Dijkstra and Lamport

- Edsger W. Dijkstra: was a Dutch systems scientist, programmer, software engineer, science essayist,[9] and early pioneer in computing science.[10] He held the Schlumberger Centennial Chair in Computer Sciences at the University of Texas at Austin from 1984 until his retirement in 1999.
- **Leslie B. Lamport** is an American computer scientist. Lamport is best known for his seminal work in distributed systems and as the initial developer of the document preparation system LaTeX. Leslie Lamport was the winner of the 2013 Turing Lamport worked as SRI International from 1977 to 1985, and Digital Equipment Corporation and joined Microsoft Research





What is a Distributed System?

- A Collection of independent computer that appear to the users of the system as a single coherent computer.

Characteristics of DS

- The computers operate concurrently
- The computers fail independently
- The computers do not share a global clock

Examples

- What is the biggest example of DS
- WWW

Advantages

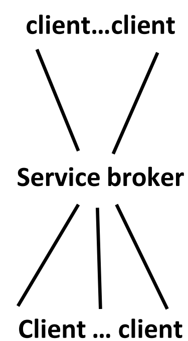
- A collection of microprocessors offer a better price/performance
- If one machine fails, the whole system can still survive
- Performance improves through load distribution
- Incremental growth

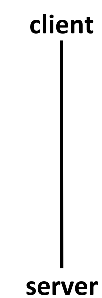
Technology Stack

- DB: Relational / Mongo DB, Cassandra, Riak, HDFS, etc.
- Computation: Hadoop, Spark, Storm
- Synchronization: NTP, Vector clocks
- Consensus: Paxos, Zookeeper
- Messaging: Kafka, RabbitMQ, etc.

Messaging

- Loosely Coupling systems
- Messages are consumed by subscribers and created by producers
- Messages are persisted for a short period of time
- Organized into topics
- Processed by brokers
- Preserve message ordering





Message Queue

Part of contents are from John O'Hara

Message Queue

- Message: an immutable array of bytes
- Topic: a feed of messages
- Producer: a process that publishes messages to a topic
- Consumer: a single-threaded process that subscribes to a topic
- Broker: one of servers that comprise a cluster

Message Passing

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
- **Non-blocking** is considered **asynchronous**

Solutions

- Synchronous Interaction
  - Service replication and load balancing
  - Enable complex interactions with some execution guarantee
- Asynchronous Interaction
  - the caller sends a message that gets stored somewhere until the receiver reads it and sends a response.
- Asynchronous interaction can take place in two forms:
  - non-blocking invocation (similar to batch jobs)
  - persistent queues

Failure Semantics

- At-most-once delivery
- At-least-once delivery
- Exactly-once delivery

Failure Semantics

- A great deal of the functionality built around RPC tries to address the problem of failure semantics,
- Exactly-once semantics solves this problem but it has hidden costs:
- it implies atomicity in all operations
- the server must support some form of 2PC
- it usually requires a coordinator to oversee the interaction (the coordinator may also fail)

Publish/Subscribe

- Standard client/server architectures and queuing systems assume the client and the server know each other
- a service publishes messages/events of given type
- clients subscribe to different types of messages/events
- when a service publishes an event, the system looks at a table of subscriptions and forwards the event to the interested clients;

Apache Kafka

A high-throughput distributed messaging system

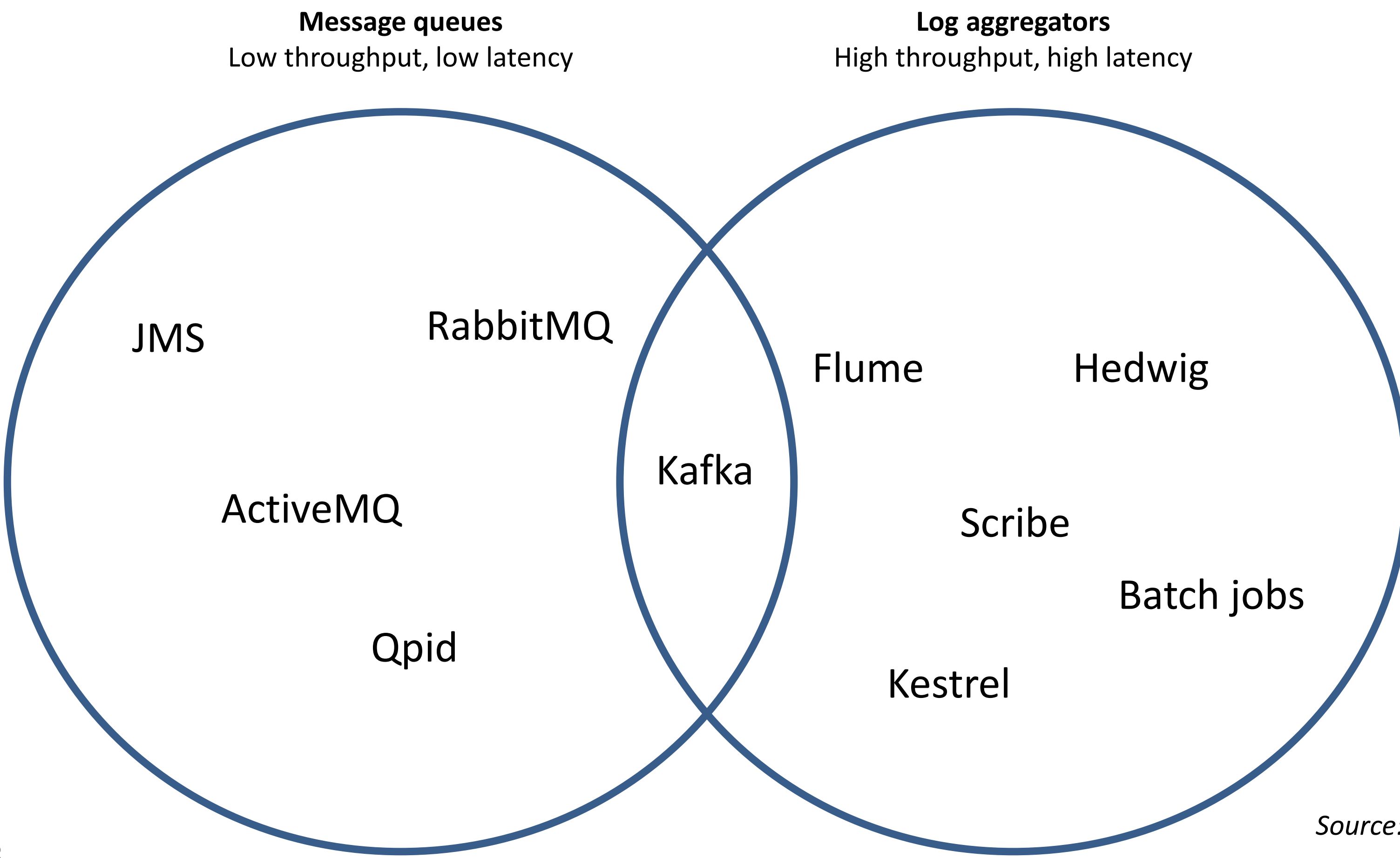
What is Apache Kafka?

- Distributed, high-throughput, pub-sub messaging system
  - Fast, Scalable, Durable
- Main use cases:
  - log aggregation, real-time processing, monitoring, queueing
- Originally developed by LinkedIn
- Implemented in Scala/Java
- Top level Apache project since 2012: http://kafka.apache.org/

Source: Johan Lundahl

Comparison to other messaging systems

- Traditional: JMS, xxxMQ/AMQP
- New gen: Kestrel, Scribe, Flume, Kafka

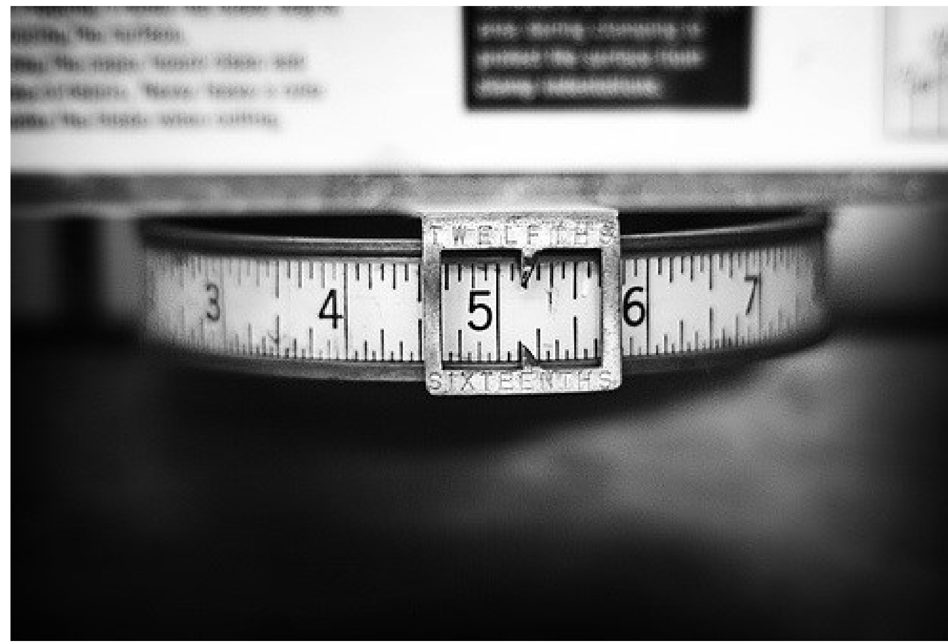


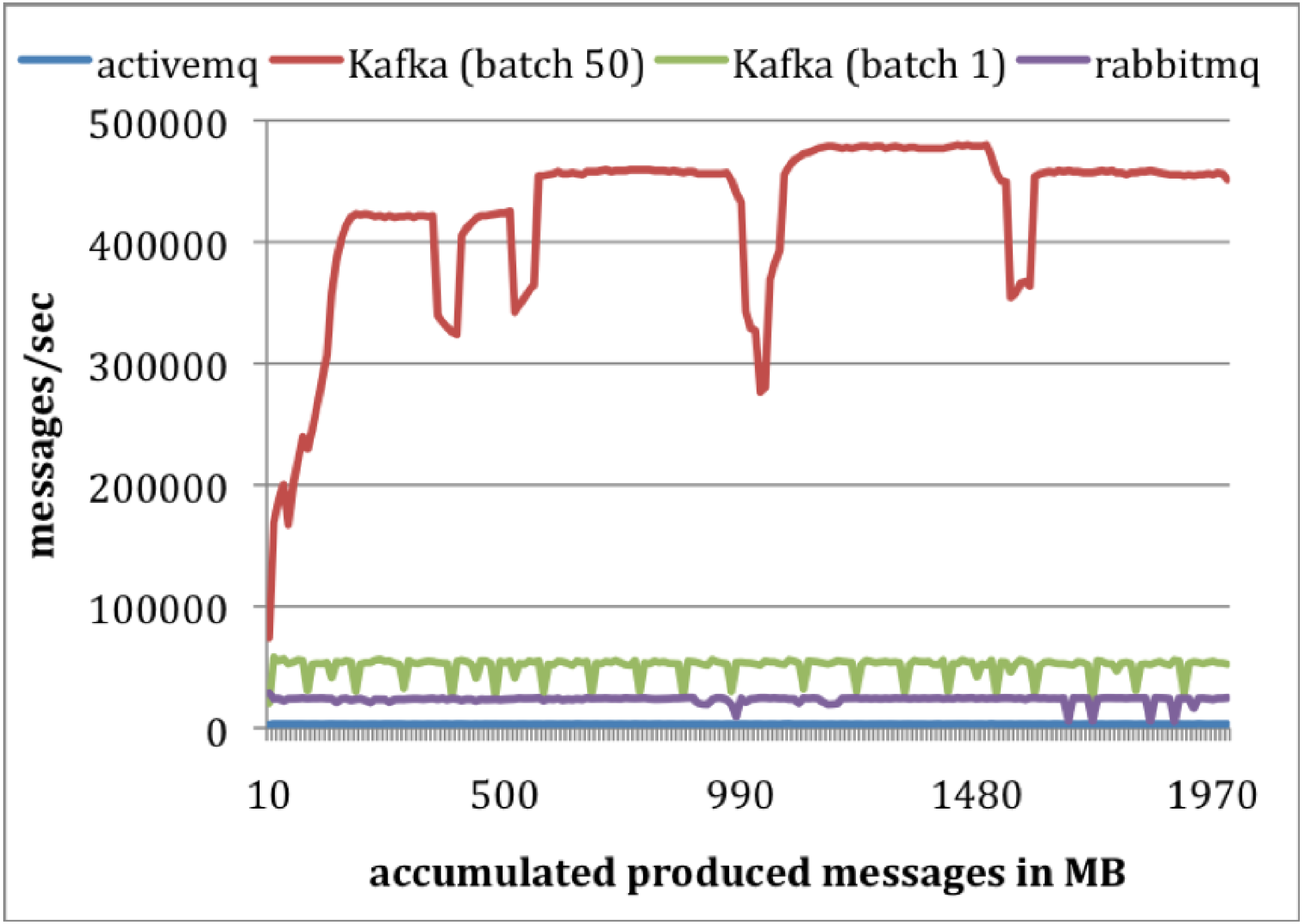
Source: Johan Lundahl

Who is using Kafka

Volume

- 20B events/day
- 3 terabytes/day
- 150K events/sec





activemq
Kafka (batch 50)
Kafka (batch 1)
rabbitmq
messages/sec
500000
400000
300000
200000
100000
0
10
500
990
1480
1970
accumulated produced messages in MB

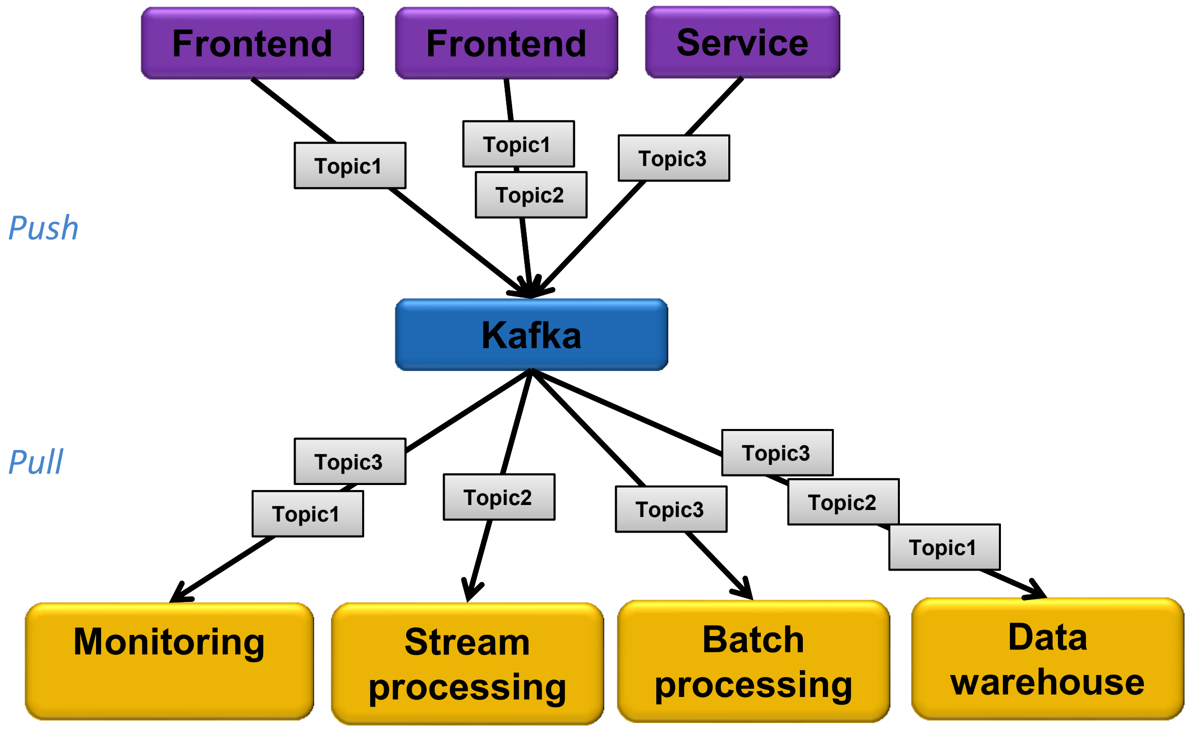
Kafka concepts

Pub and Sub

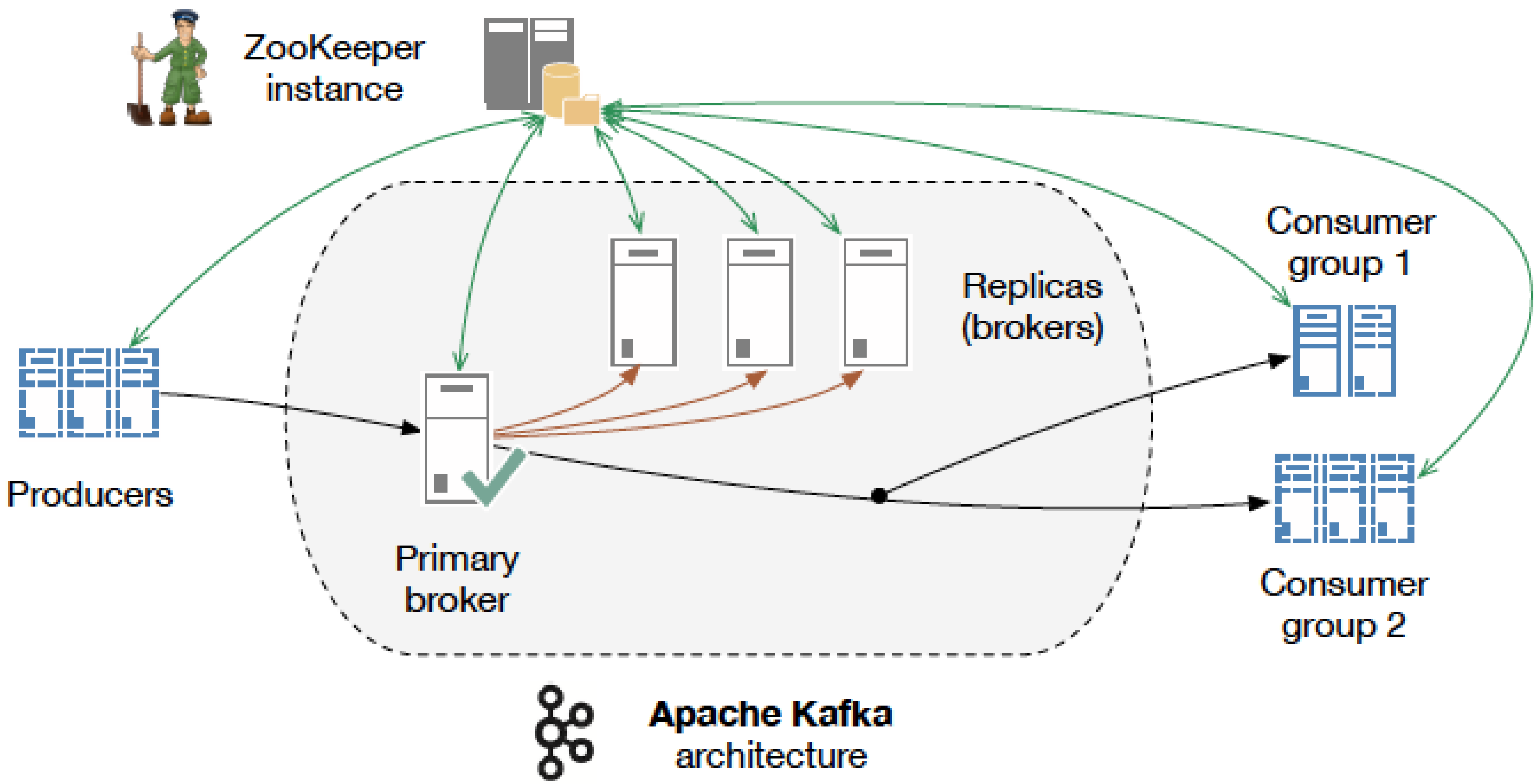
Producers

Broker

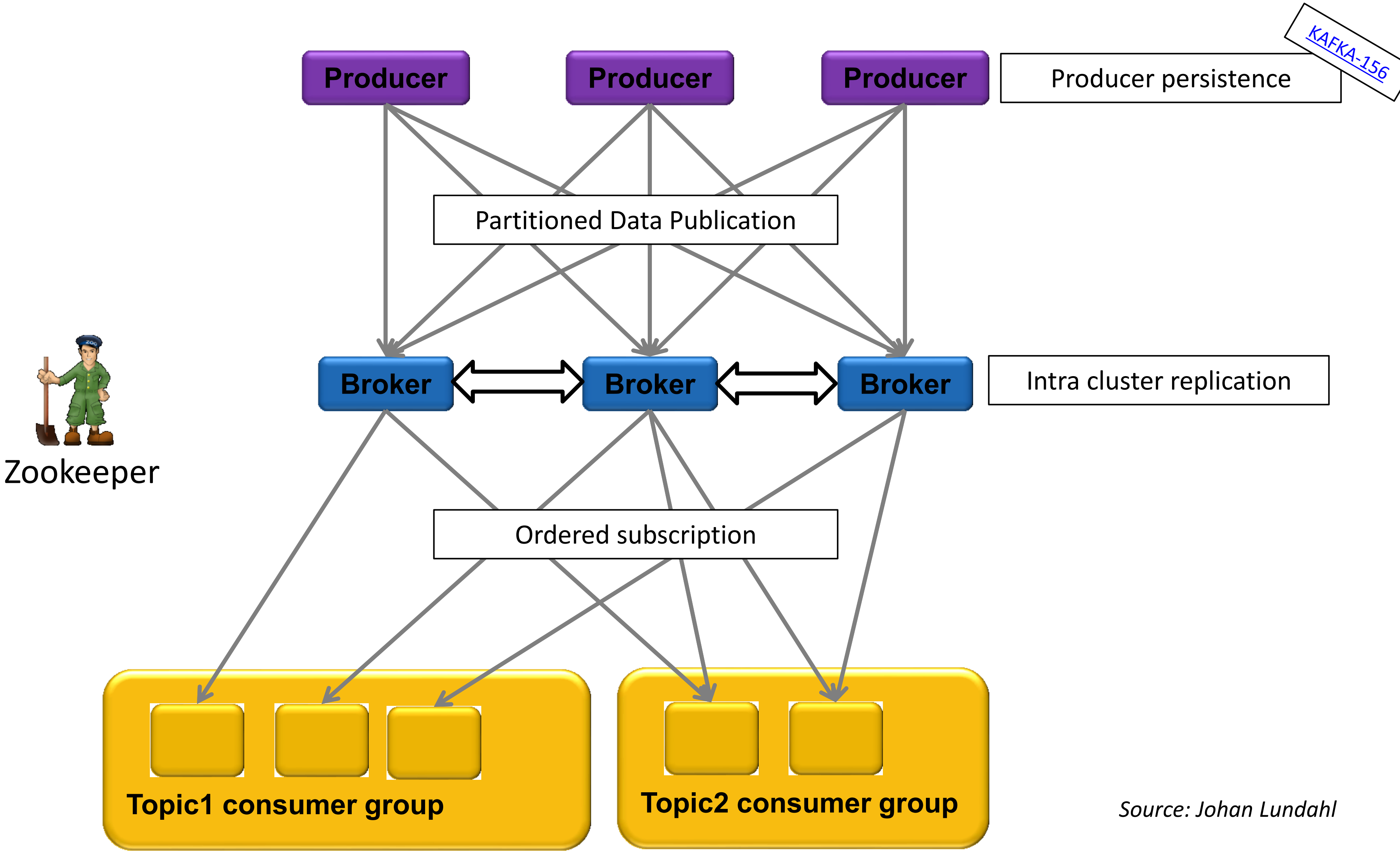
Consumers



Kafka

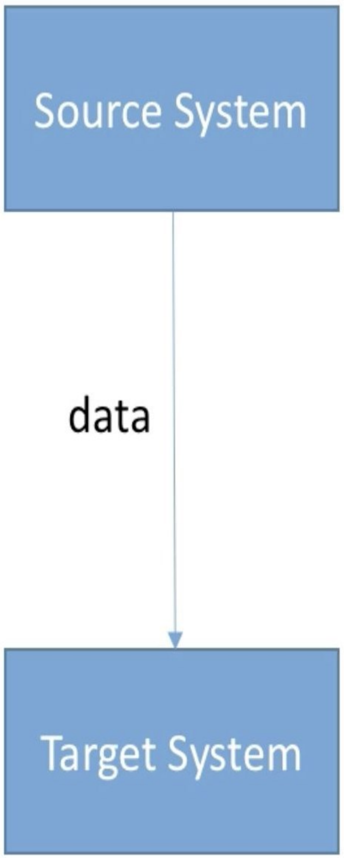

ZooKeeper
instance
Consumer
group 1
Replicas
(brokers)
Producers
Primary
broker
Consumer
group 2
Apache Kafka
architecture

Distributed model



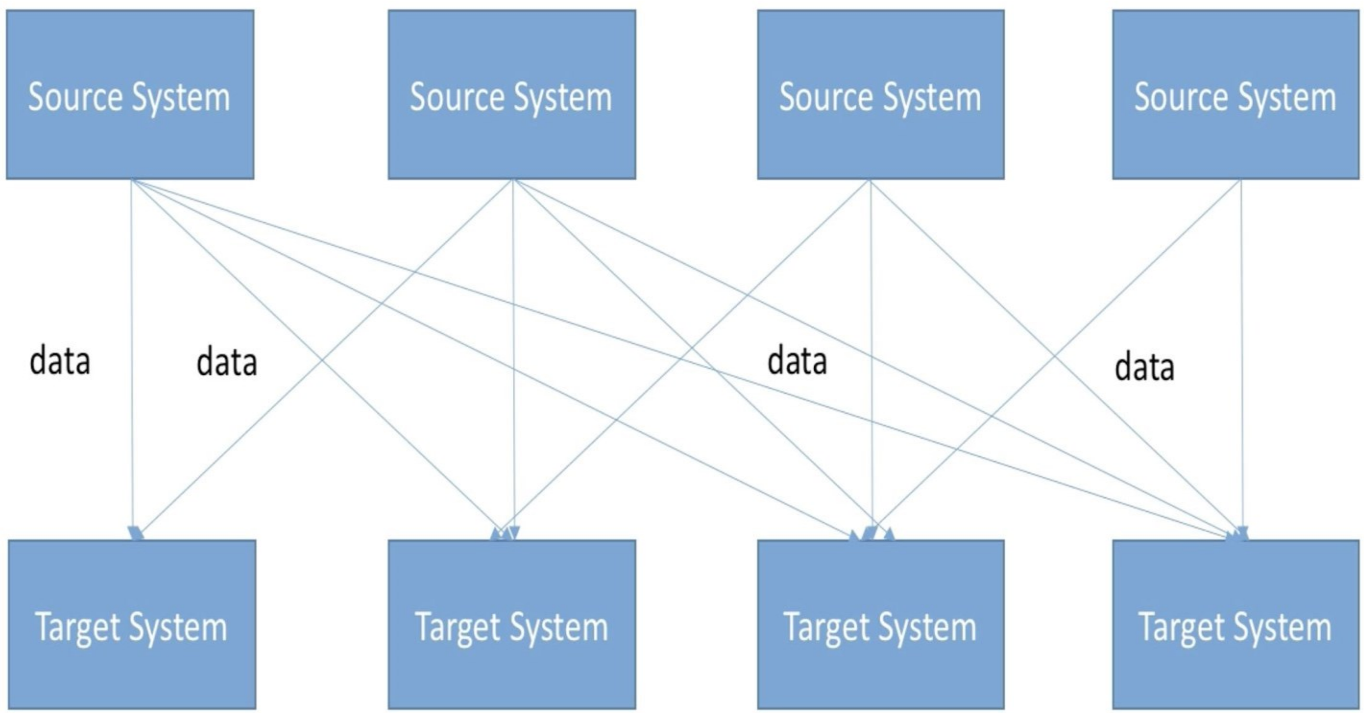
Source: Johan Lundahl

Introduction to Kafka



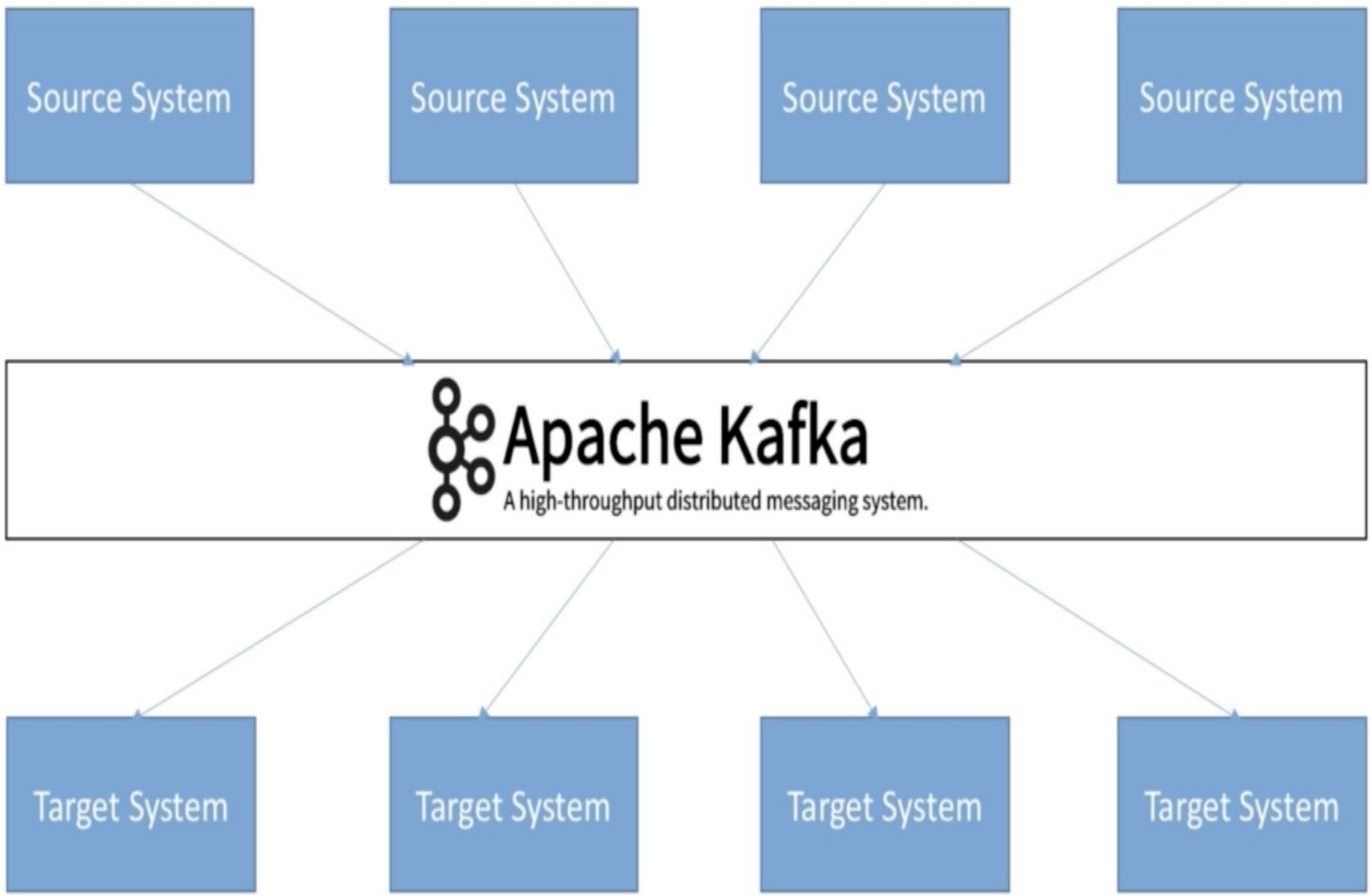
Simple at first!

Introduction to Kafka



Very complicated!

Introduction to Kafka

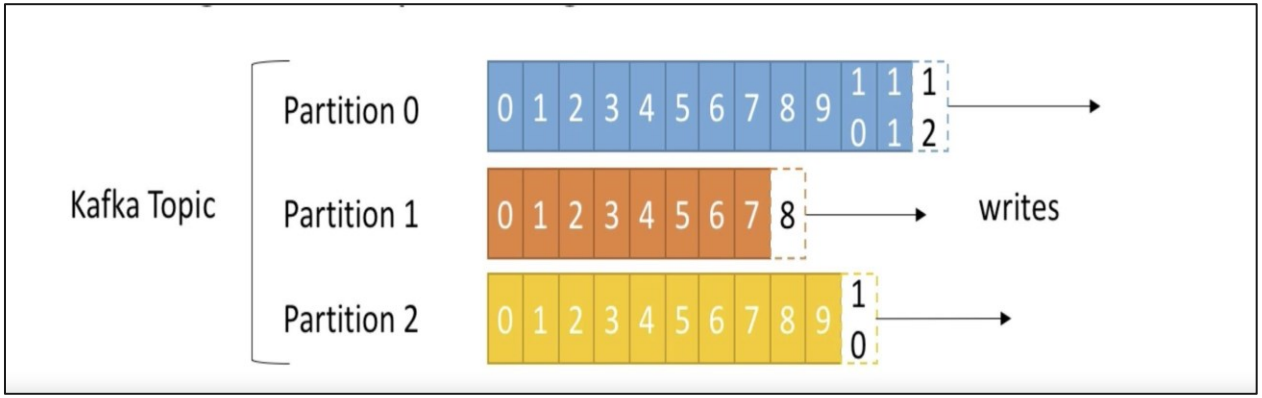


Apache Kafka : Use Cases

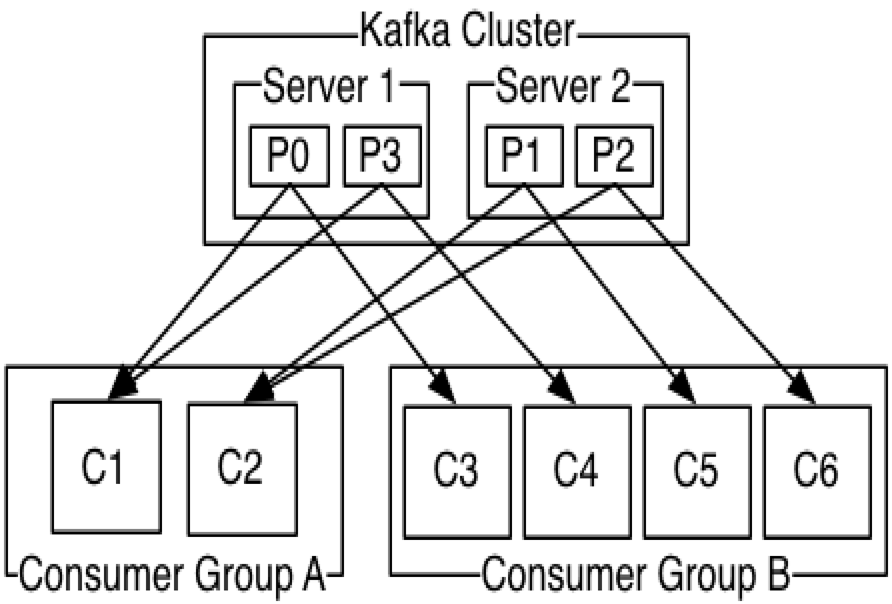
- Messaging System
- Activity Tracking
- Metrics Gathering
- Application Logs Gathering
- Stream Processing
- Integration with Spark, Hadoop and other big data technologies.

Topics, Partitions and Offsets

- **Topics:**
  - Stream of Data. Similar to a table in Database
  - Identified by its name.
- **Partitions:**
  - Topics are split into partitions.
  - Specify number of partitions you want during topic creation.
- **Offset:**
  - Each message within a partition gets an incremental id, called offset.
  - Order is guaranteed within partitions (not across)



Kafka Consumers and Consumer Groups



Brokers

- A Kafka Cluster consists of multiple brokers. Each brokers is basically a server.

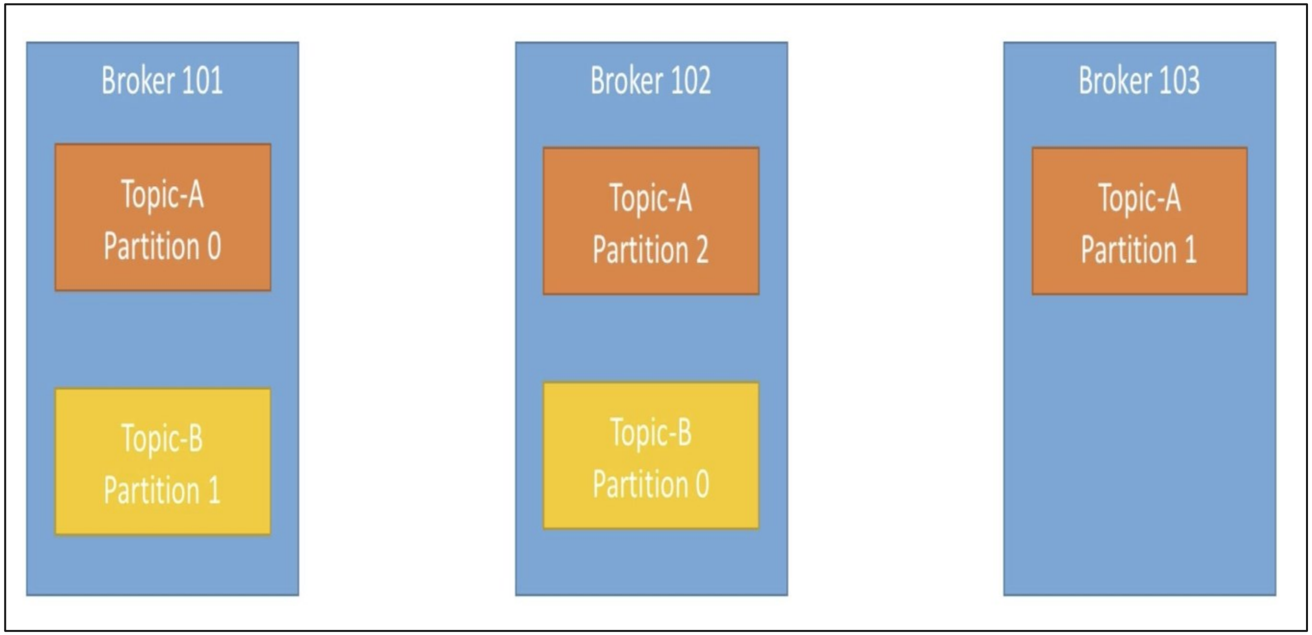
- Each broker is identified by a numeric ID.
- Each broker will contain only certain topic partitions. Idea is to make a distributed system where each broker store some data and not all data.

- If you are connected to one broker of the cluster, you will be connected to the entire cluster.

- Ideally 3 number of brokers is a good number to have, some have 100 brokers.

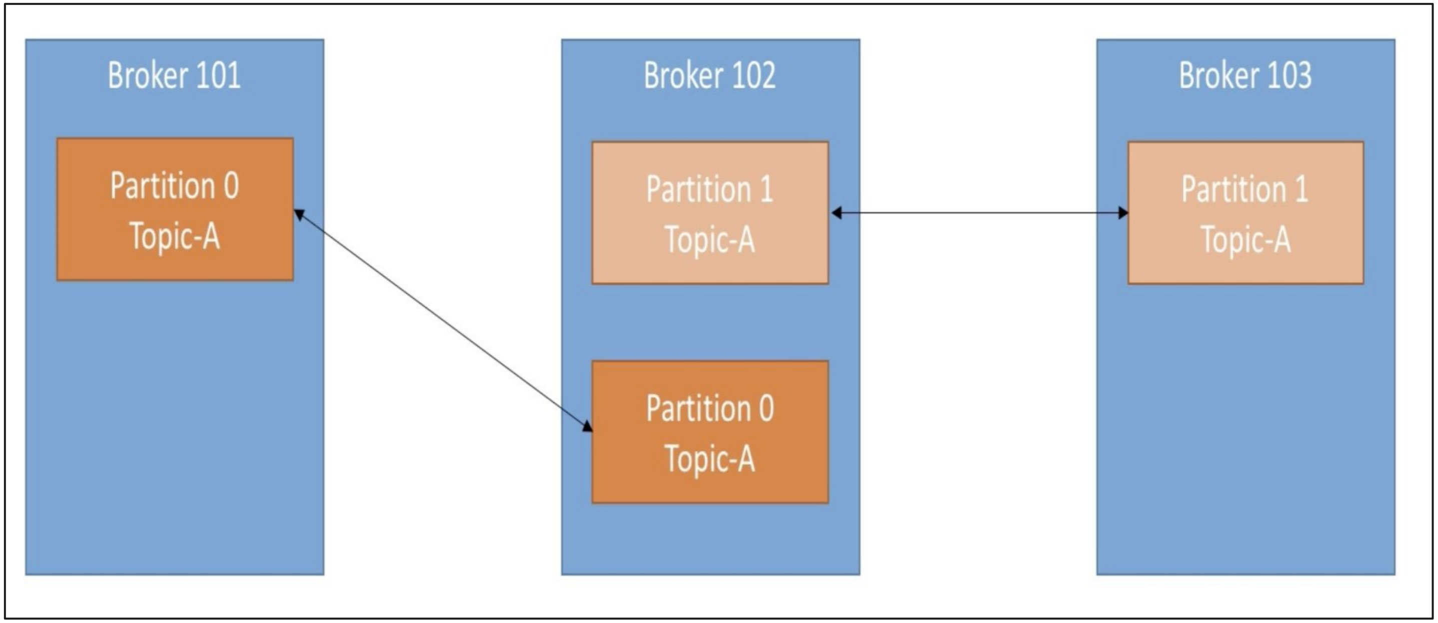
Brokers and Topics

- Consider a Topic named "**Topic - A**" which has **3 partitions.**
- Consider a Topic named "**Topic - B**" which has **2 partitions.**



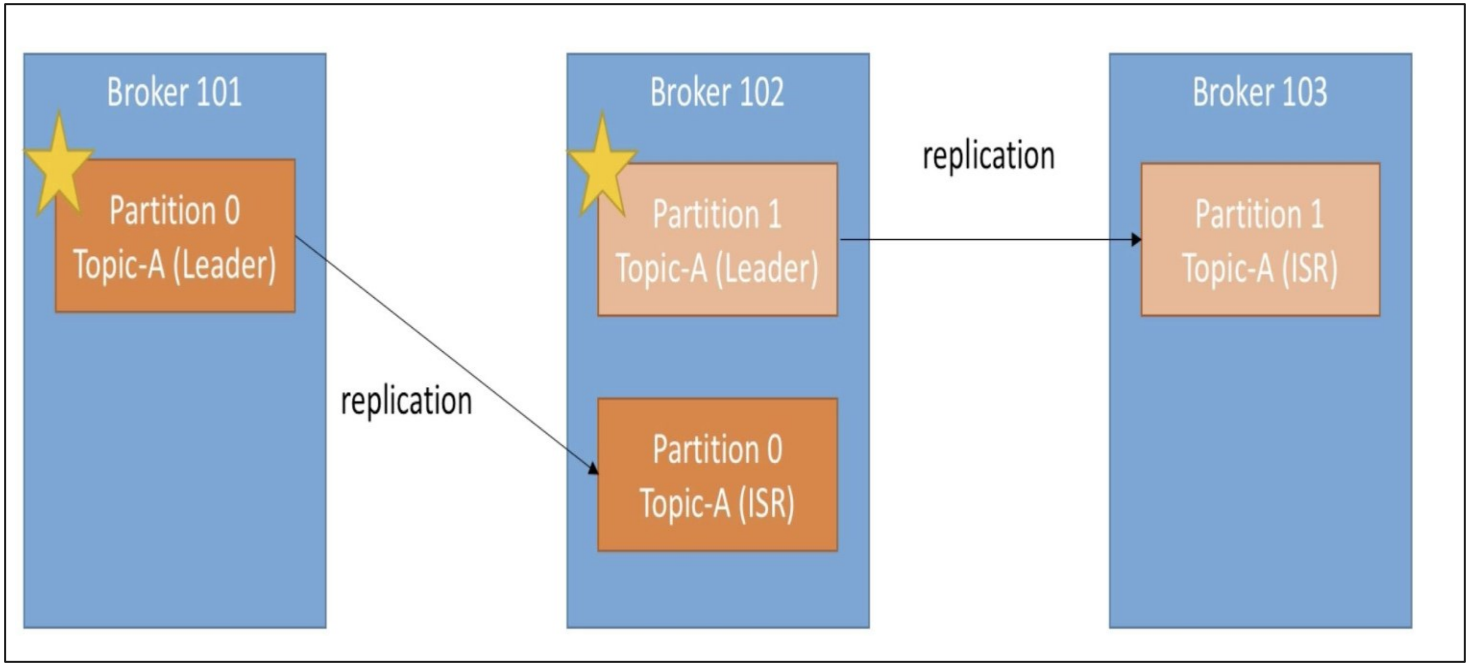
Replication Factor for a Topic

- When creating a topic you need to specify the replication factor. (Usually between 2 and 3)
- Reason : If a broker is down, another broker can serve the data.
- Example : Topic A with 2 partitions and a replication factor of 2.



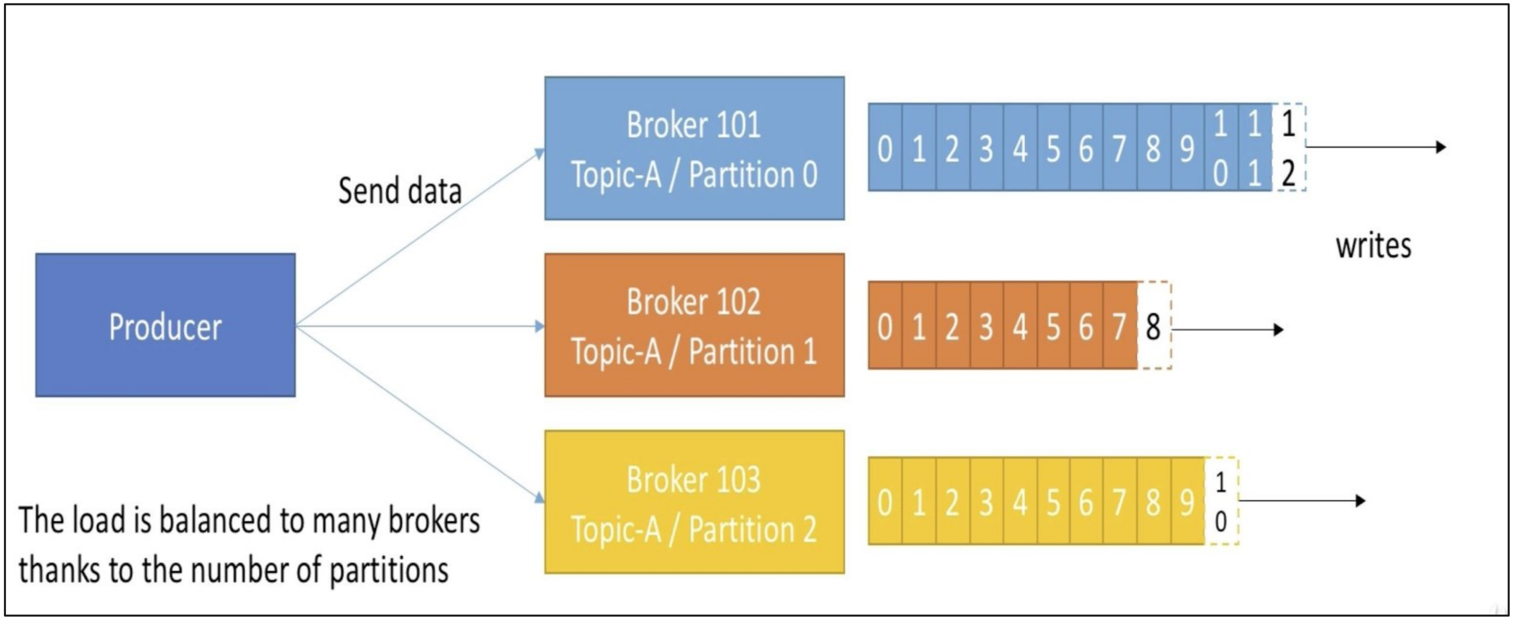
Leader for a Partition

- Only One Broker can be the Leader for a Partition.
- Only Leader can receive and serve data for a partition.
- Other replicas will just synchronize the data.



Producers

- Role of the Producers is to write data to Kafka Topics.
- Producers know to which broker and partition to write to.
- In case of Broker failure, Producers will automatically recover.

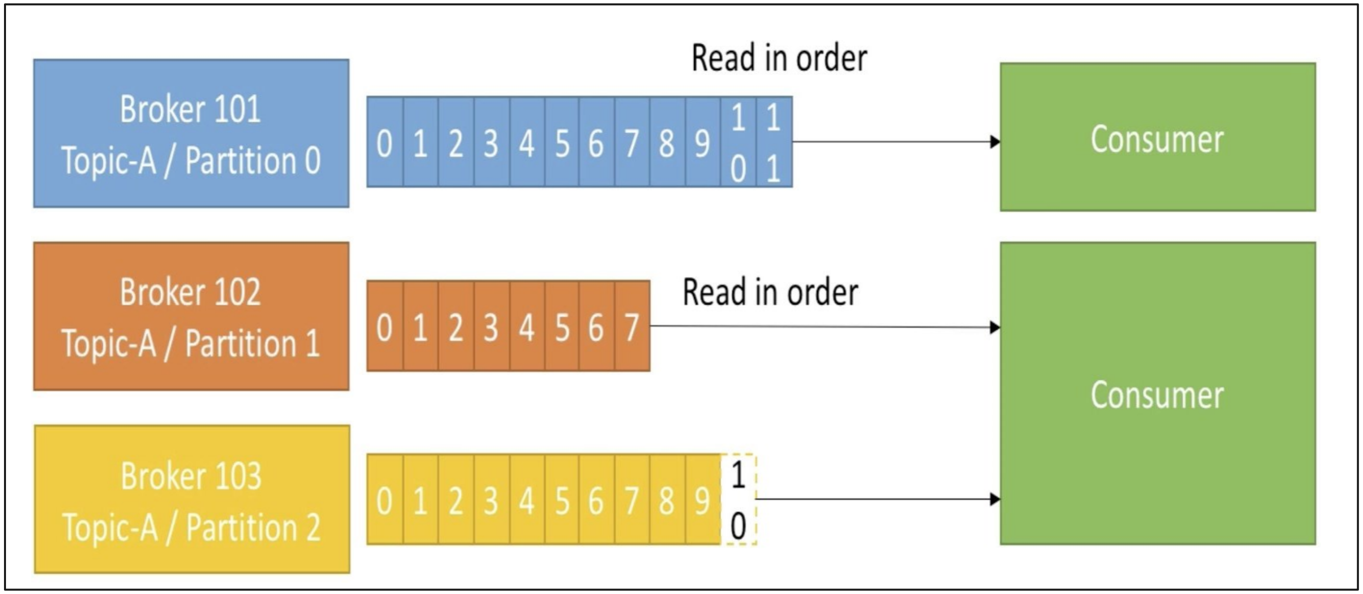


Producers - Contd

- Producer can choose to receive acknowledgement for data writes:
  - Acks = 0: Do not wait for ACK.(Can cause data loss)
  - Acks = 1 : Wait till leader acknowledges.
  - Acks = all : Wait for ACKS from all.
- Producers can choose to send a key with the message.
  - If key = null : Data is sent in round robin to all brokers.
  - If key is sent that all data for that key will go to the same partition.
  - A key is send if you need message ordering

Consumers

- Consumers read data from a Kafka topic.
- Consumers know which broker to read data from.
- In case of broker failure, Consumers will recover automatically.
- Data is read in order within each partitions.



Consumer Offsets

- Kafka stores the offset a which a consumer has been reading. (similar to checkpointing).
- These offsets are stored live in Kafka topic named consumer_offsets.
- Reason: If a consumer dies it can start reading where it left off by looking at the offset.

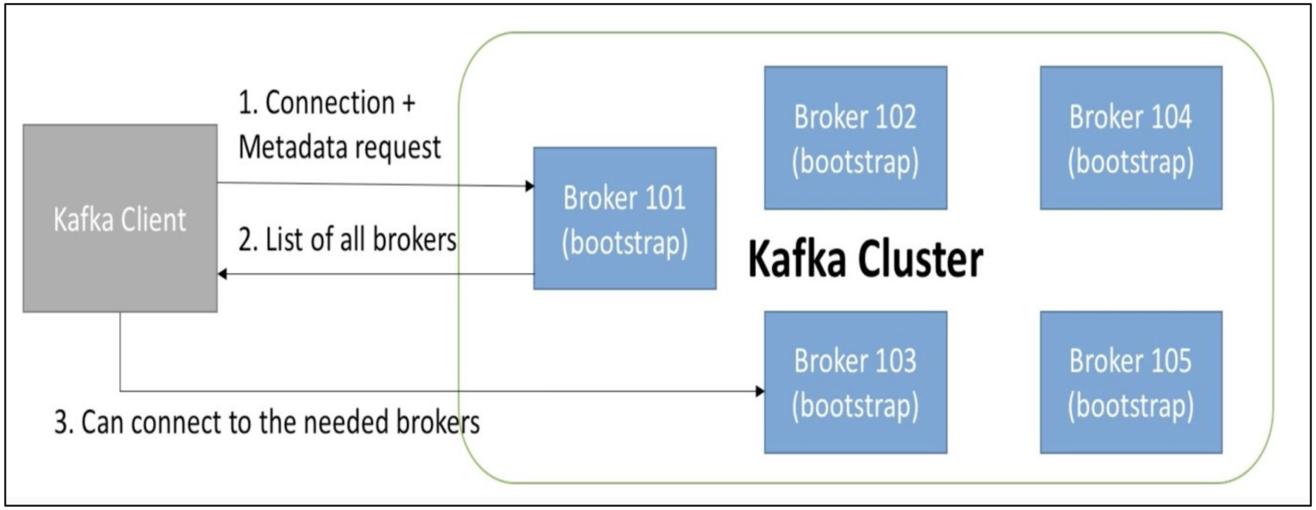
- Delivery Semantics: When to commit offsets

  - At Most Once: Offsets are committed as soon as messages are received.

  - At Least Once : Commit Offsets only after the message is processed.
  - Exactly Once : Ideal condition and is difficult to achieve. Can be achieved between Kafka - Kafka systems.

Kafka Broker Discovery

- Every Kafka Broker is also called as bootstrap broker.
- If you connect to any broker, you will be automatically connected to the entire cluster.
- Each broker knows about all other brokers, topics and partitions.



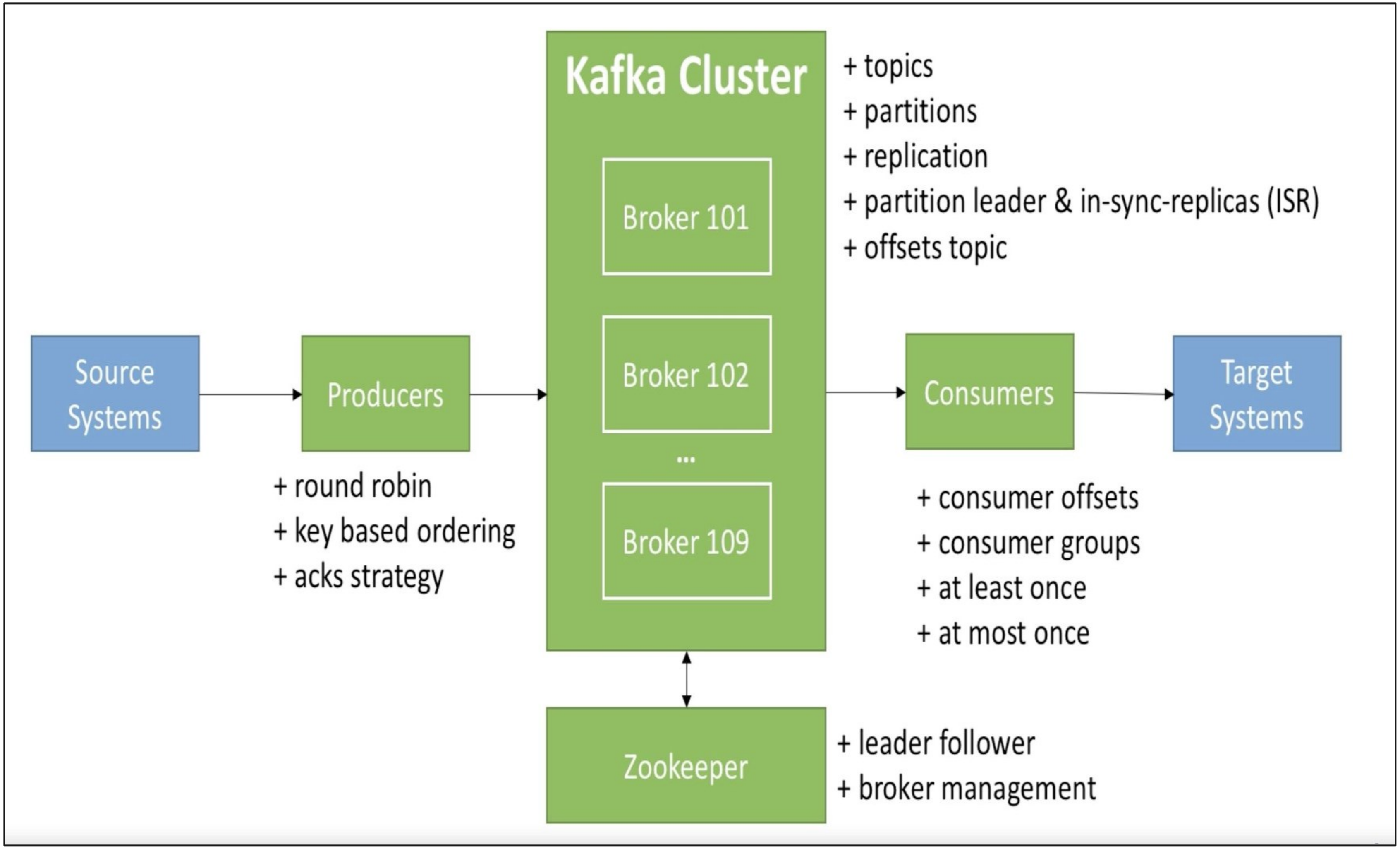
Zookeeper

- Holds all the brokers together.
- Helps in performing leader election for a partitions.
- Send Notification to Kafka Cluster in case of any changes in the system

  (e.g. new topic created, topic deleted, broker fails, broker comes up,

  etc.)
- **Kafka cannot work without a zookeeper.**
- Zookeeper operates with a odd numbers of servers(3,5, 7).
- Zookeeper also has the concept of leader(handle writes) and followers(handle reads).

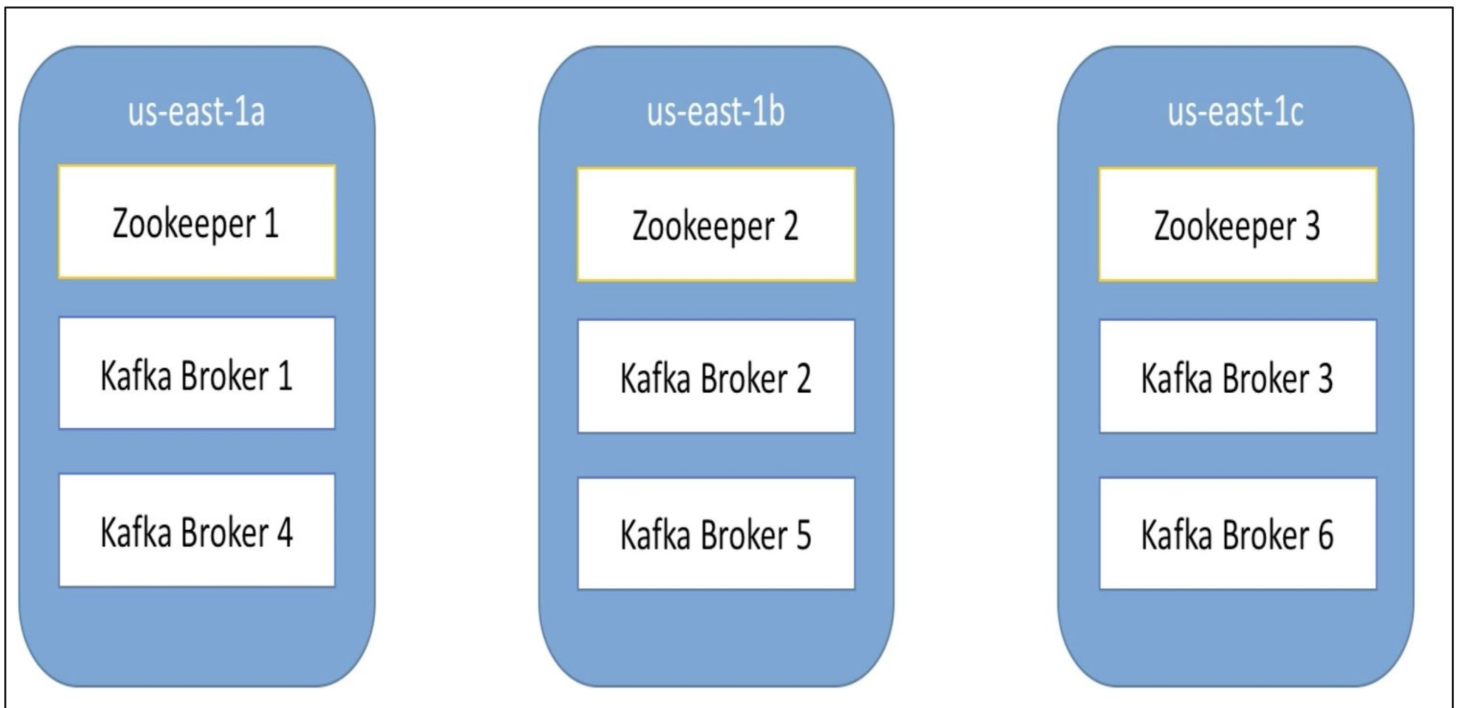
All at one place



Demo

Kafka Cluster Setup - High Level Architecture

- You want multiple brokers in different data centers to serve the data.
- You also want the number of Zookeeper to be at least 3.

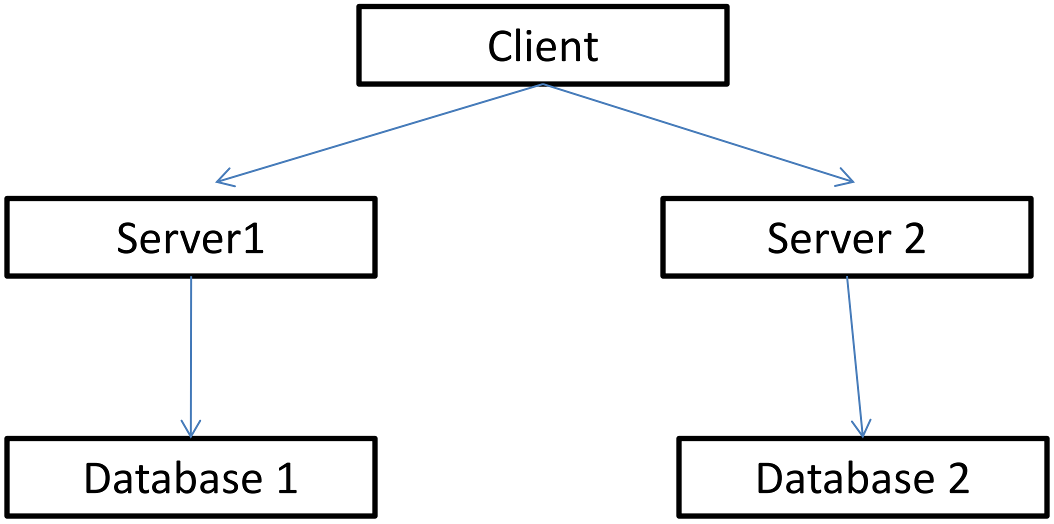


TP-Monitors

- The problems of synchronous interaction are not new. The first systems to provide alternatives were TP-Monitors :
- asynchronous RPC: client makes a call that returns immediately; the client is responsible for making a second call to get the results
- Reliable queuing systems (e.g., Tuxedo) where instead of through procedure calls, client and server interact by exchanging messages. Making the messages persistent by storing them in queues

Distributed DB

- Not all the data can reside in the same database
- The application is built on top of the database.



2 phase commit protocol

- „prepare to commit“ message to server
- server responds „ready to commit“
  - guarantees successful commit of procedure
- check whether all servers are ready
  - ready: commit results of requests
  - not ready: cancelation, rollback
  - log states of transactions

transactional RPC

- conventional RPC
  - not taking care of dependencies within various calls
  - difficult error detection
- TRPC
  - derived from database transactions
  - transaction criteria: atomicity, consistency, isolation and durability
  - if not all criteria are fullfilled: rollback
  - criteria also applicable to RPC calls

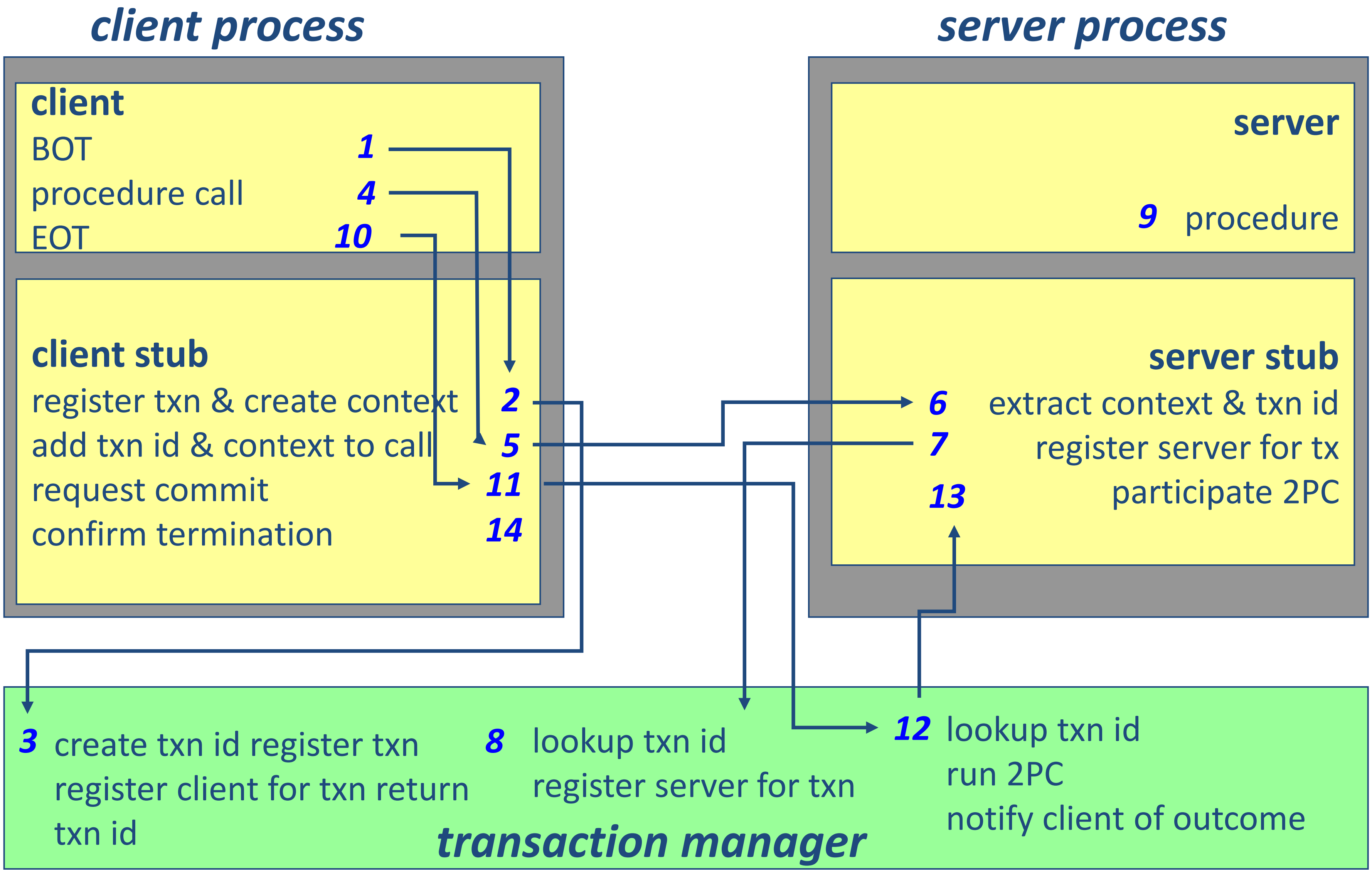
transactional RPC

- define procedures within transactional brackets
  - „beginning of transaction“ – BOT
  - „end of transaction“ – EOT
  - handled by transaction management tool
    - responsible for interaction between client and server

Transactional Queues

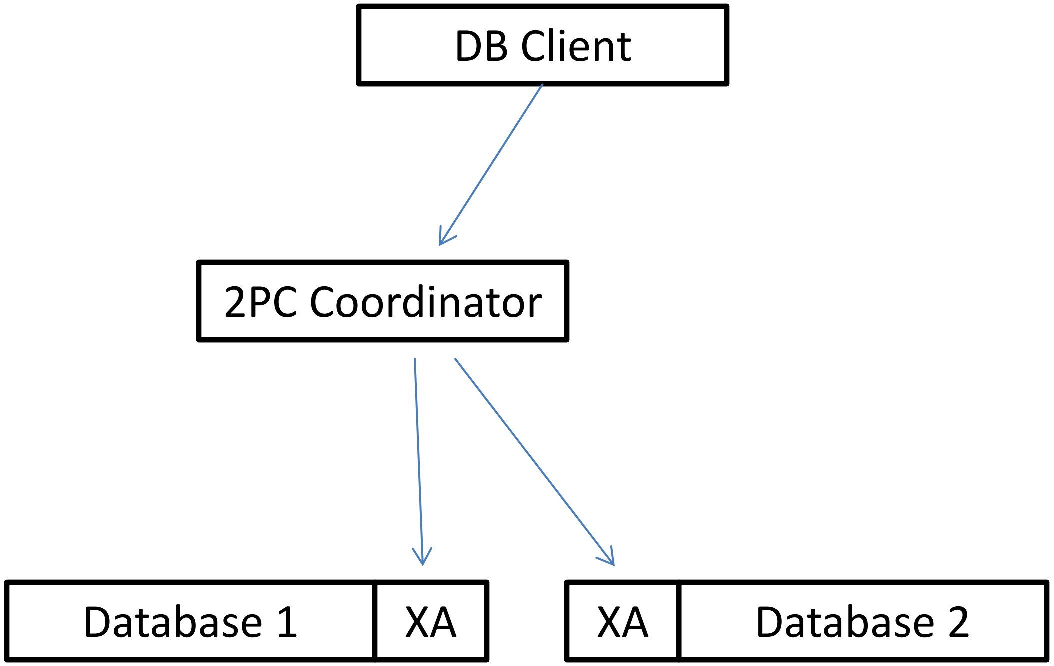
- Persistent queues are closely tied to transactional interaction:
  - to send a message, it is written in the queue using 2PC
  - messages between queues are exchanged using 2PC
  - reading a message from a queue, processing it and writing the reply to another queue is all done under 2PC
- This introduces a significant overhead but it also provides considerable advantages. The overhead is not that important with local transactions (writing or reading to a local queue).

TPM – transactional RPC



Coordinator

- Intermediate layer is needed to run the 2PC protocol

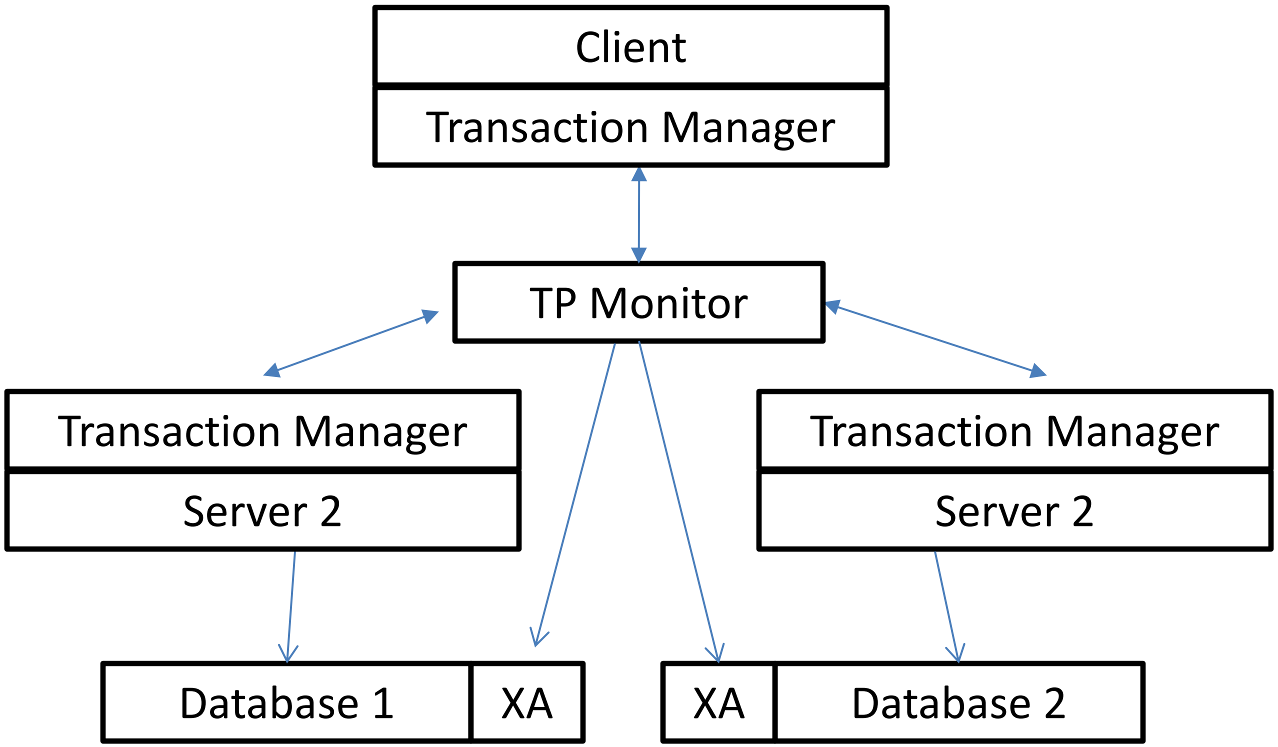


Why Coordinate?

- Control of Participants: A transaction may involve many resource managers, somebody has to keep track of which ones have participated in the execution
- Preserving Transactional Context
- Transactional Protocols: somebody acting as the coordinator in the 2PC protocol
- Make sure the participants understand the protocol

Coordinator

- the TM runs 2PC with resource managers instead of with the server



TP-Monitor

- Common interface to several applications while maintaining or adding transactional properties. Examples: CICS, Tuxedo, Encina.
- A TP-Monitor extends the transactional capabilities of a database beyond the database domain
- TP-Monitors are, perhaps, the best, oldest, and most complex example of middleware.

TP-Monitor Functionality

- Transactional RPC: Implements RPC and enforces transactional semantics, scheduling operations accordingly
- Transaction manager: runs 2PC and takes care of recovery operations
- Log manager: records all changes done by transactions so that a consistent version of the system can be reconstructed in case of failures
- Lock manager: a generic mechanism to regulate access to shared data outside the resource managers

References

- **Java Messaging** by Eric Bruno (Charles River Media, 2005)
- http://en.wikipedia.org/wiki/Two-phase_commit_protocol
- Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman (1987):*Concurrency Control and Recovery in Database Systems*, Chapter 7, Addison Wesley Publishing Company, ISBN 0-201-10715-5
- **The XA Specification** https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?catalogno=c
- Transaction Processing: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems
- Microsoft Message Queues: http://msdn.microsoft.com/en-us/library/ms644927(VS.85).aspx http://msdn.microsoft.com/en-us/library/ms711472.aspx
- O'Hara, J. (2007). "Toward a commodity enterprise middleware". *Acm Queue* **5**: 48–55.
- Vinoski, S. (2006). "Advanced Message Queuing Protocol". *Ieee Internet Computing* **10**: 87–89