The Part-Time Parliament (Paxos)

Ahmed Ali-Eldin

Original paper

- Leslie Lamport wrote the paper in 1989
- Submitted it to ACM Transactions on Computer Systems
  - “All three referees said that the paper was mildly interesting, though not very important”
- Cast the algorithm in terms of a parliament on an ancient Greek island
  - An extremely funny paper to read
  - But people missed the whole point because of the Greek island/parliament story
- Paper was published 10 years later roughly and is a classical paper in DS
  - A cornerstone for all cloud systems
  - Lamport is the father of modern Distributed Systems (and Latex)

  http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#lamport-paxos

  http://amturing.acm.org/award_winners/lamport_1205376.cfm

Original paper

- Created a lost civilization and described the decision-making in the parliament of the Greek island of Paxos
- Used names of computer scientists for the Greek legislators, bogus Greek dialect
- Gave some lectures dressed as an Indiana Jones-like character

Original Paper abstract

"Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state-machine approach to the design of distributed systems"

http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf

Paxos made Simple

“At the PODC 2001 conference, I got tired of everyone saying how difficult it was to understand the Paxos algorithm, published in [122]. Although people got so hung up in the pseudo-Greek names that they found the paper hard to understand, the algorithm itself is very simple. So, I cornered a couple of people at the conference and explained the algorithm to them orally, with no paper. When I got home, I wrote down the explanation as a short note, which I later revised based on comments from Fred Schneider and Butler Lampson. The current version is 13 pages long, and contains no formula more complicated than $n1 > n2$. ”

http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#paxos-simple

Paxos Family

- Basic Paxos
- Multi-Paxos
- Cheap Paxos
- Fast Paxos
- Generalized Paxos
- Byzantine Paxos

So what is Paxos?

- It is a (simple) consensus algorithm
  - Assume a collection of processes that can propose values.
  - A consensus algorithm ensures that a single one among the proposed values is chosen.
    - If no value is proposed, then no value should be chosen.
    - If a value has been chosen, then processes should be able to learn the chosen value

Safety requirements of consensus

- Only a value that has been proposed may be chosen
- Only a single value is chosen,
- And, a process never learns that a value has been chosen unless it actually has been.

Paxos Properties

- Paxos is an asynchronous consensus algorithm
  - Asynchronous networks
    - No common clocks or shared notion of time (local ideas of time are fine, but different processes may have very different “clocks”)
    - No way to know how long a message will take to get from A to B

Paxos Properties

- Paxos is guaranteed safe.
  - Consensus is a stable property: once reached it is never violated; the agreed value is not changed.

Paxos Properties

- Paxos is not guaranteed live.
  - Consensus is reached if “a large enough subnetwork...is non-faulty for a long enough time.”
  - Otherwise Paxos might never terminate.

Liveness

- Fischer-Lynch-Patterson (1985)
  - No consensus can be guaranteed in an asynchronous communication system in the presence of any failures.
  - Intuition: a “failed” process may just be slow, and can rise from the dead at exactly the wrong time.

Liveness

- FLP tells us that it is impossible for an asynchronous system to agree on anything with accuracy and liveness!
  - Liveness requires that agents are free to accept different values in subsequent rounds.
  - But: safety requires that once some round succeeds, no subsequent round can change it.

Paxos (consensus) Agents

- Proposers
  - Sends a proposed value to a set of acceptors
- Acceptors
  - May accept the proposed value
  - Value chosen when a large enough number of acceptors accepted it.
- Learners
  - Must find out that a proposal has been accepted by a majority of acceptors
  - Learns the value

Assumptions

- Agents
  - Operate at arbitrary speeds
  - May fail by stopping
  - May restart
  - A solution is impossible unless some information can be remembered by an agent that has failed and restarted
- Messages
  - Can take arbitrarily long to be delivered (six months in the original paper ☺)
  - Can be duplicated
  - Can be lost
  - But they are not corrupted.

Choosing a Value

- To ensure that only a single value is chosen it must be chosen by a large enough set of acceptors.
- let a large enough set consist of any majority of the agents.
- Any two majorities have at least one acceptor in common
- This works if an acceptor can accept at most one value.
- In the absence of failure or message loss, we want a value to be chosen even if only one value is proposed by a single proposer.

Requirement 1

- P1. An acceptor must accept the first proposal that it receives.

Does this sound like a recipe for failure?

Choosing a Value

- Raises issues
  - Several values could be proposed by different proposers at about the same time
  - Situation in which every acceptor has accepted a value, but no single value is accepted by a majority of them.
  - A situation with just two proposed values, if each is accepted by about half the acceptors, failure of a single acceptor could make it impossible to learn which of the values was chosen

Choosing a Value

- Instead allow each acceptor to accept multiple proposals
- We keep track of the different proposals that an acceptor may accept by assigning a (natural) number to each proposal
  - A proposal consists of a proposal number and a value.
- We can allow multiple proposals to be chosen, but we must guarantee that all chosen proposals have the same value (This is consensus)

Requirement 2

- P2. If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v.
- P2a . If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v.
- P2b . If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v.
- P2c . For any v and n, if a proposal with value v and number n is issued, then there is a set S consisting of a majority of acceptors such that either (a) no acceptor in S has accepted any proposal numbered less than n, or (b) v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S.

So what does a proposal do (1)

- A proposer chooses a new proposal number n and sends a request to each member of some set of acceptors, asking it to respond with:
  - (a) A promise never again to accept a proposal numbered less than n,
  - and (b) The proposal with the highest number less than n that it has accepted, if any.
- This request is called a *prepare request* with number n.

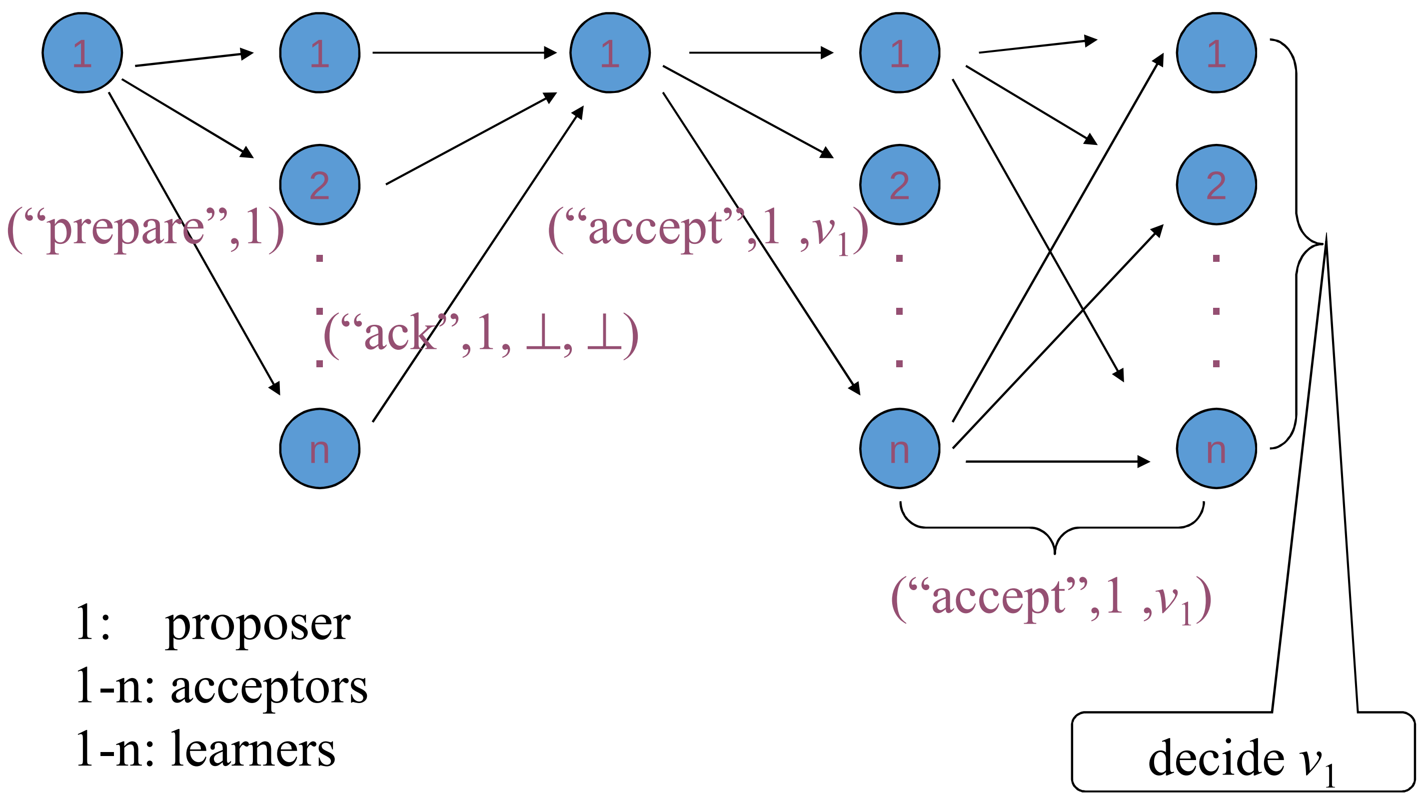
So what does a proposal do (2)

- If the proposer receives the requested responses from a majority of the acceptors
  - it can issue a proposal with number n and value v, where v is the value of the highest-numbered proposal among the responses, or is any value selected by the proposer if the responders reported no proposals
- A proposer issues a proposal by sending, to some set of acceptors, a request that the proposal be accepted. (This need not be the same set of acceptors that responded to the initial requests.) Let’s call this an accept request.

What does the acceptor do? (1)

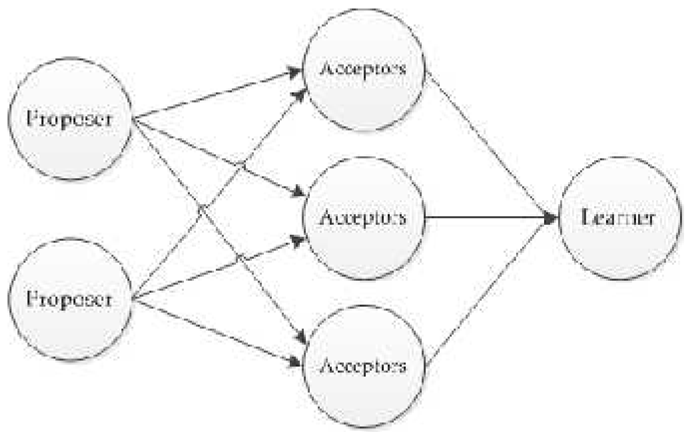
- It can receive two kinds of requests from proposers: prepare requests and accept requests.
- can ignore any request without compromising safety
- P1a . An acceptor can accept a proposal numbered n iff it has not responded to a prepare request having a number greater than n.
- an acceptor needs to remember only the highest numbered proposal that it has ever accepted and the number of the highest numbered prepare request to which it has responded.

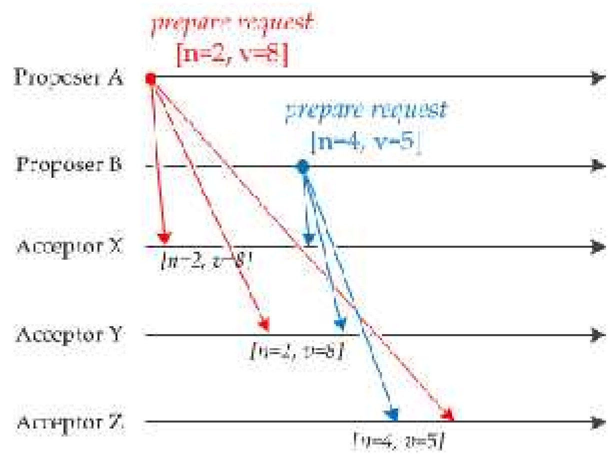
In Well-Behaved Runs



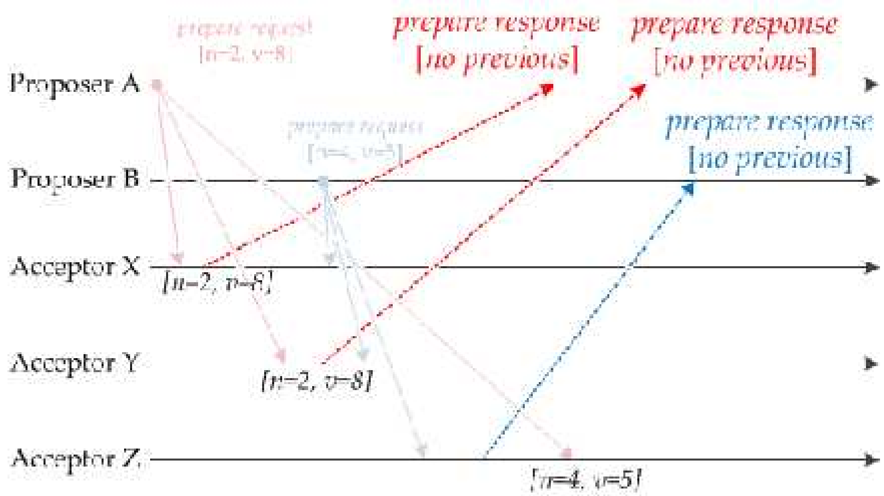
1: proposer
1-n: acceptors
1-n: learners

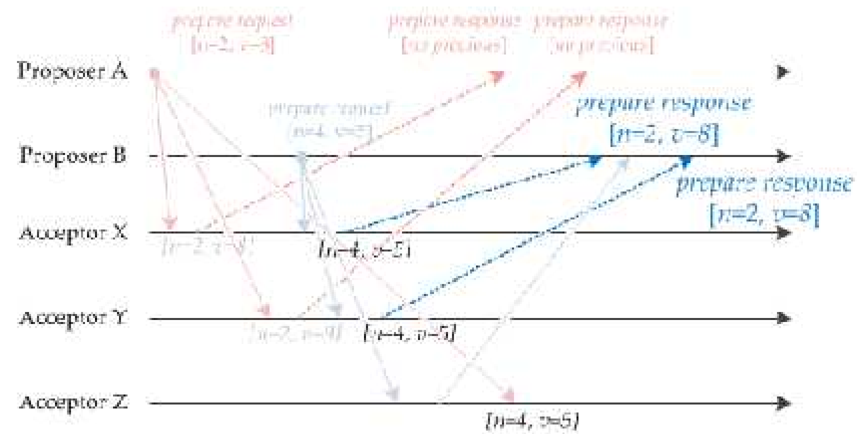
Visualizing Paxos



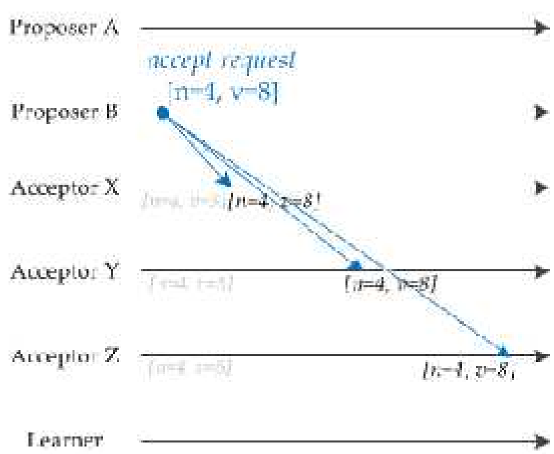


http://angus.nyc/writing/paxos-by-example/

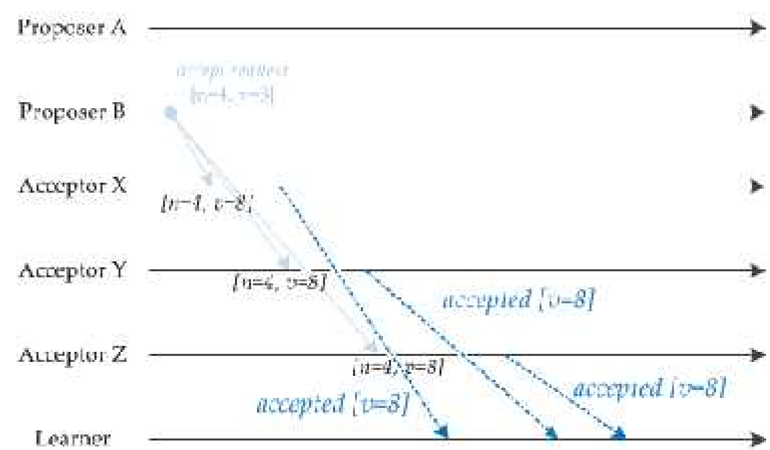




http://angus.nyc/writing/paxos-by-example/



Proposer A
accept request
[n=4, v=8]
Proposer B
Acceptor X
[n=4, v=8]
Acceptor Y
[n=4, v=8]
Acceptor Z
[n=4, v=8]
Learner



Proposer A
Proposer B
Acceptor X
[n=4, v=8]
Acceptor Y
[n=4, v=8]
accepted [v=8]
Acceptor Z
[n=4, v=8]
accepted [v=8]
accepted [v=8]
Learner

- Read Section 3 in the paper for a practical example ☺
    - The example also talks about some further complications to make Paxos work

Chubby: a lock service for loosely-coupled distributed systems

Locks and semaphores

- A lock allows only one thread to enter the part that's locked and the lock is not shared with any other processes.
  - Lock owner
- A semaphore allows x number of threads to enter.
  - You have three threads contending for two resources
  - When two threads acquire the resources, the third is blocked

Locks and Semaphores

- Pretty easy in a centralized system
  - You have control over your computer
- In a DS
  - You do not know how many computers are there
  - You have no clue in general on who is alive, unless you build a full mesh
- Question, How do you build a distributed lock?
  - One answer, Chubby

Chubby

- a Chubby instance (also known as a Chubby cell) might serve ten thousand 4-processor machines connected by 1Gbit/s Ethernet.
- Most Chubby cells are confined to a single datacentre or machine room
    - At least one Chubby cell whose replicas are separated by thousands of kilometres.
- The purpose of the lock service is
    - to allow its clients to synchronize their activities
    - and to agree on basic information about their environment.

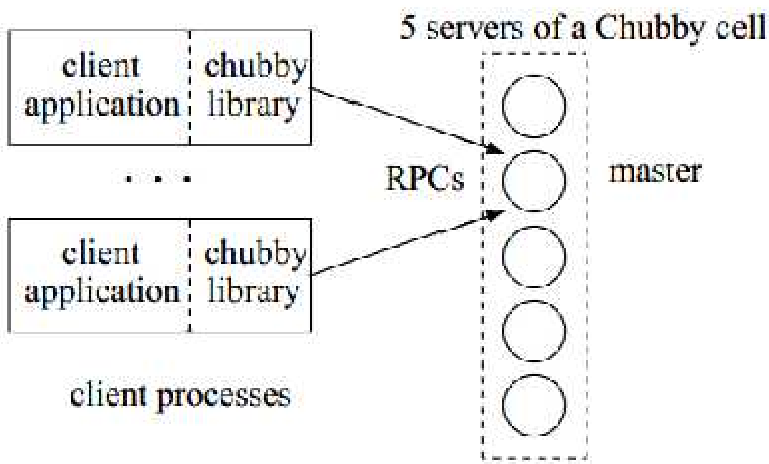
Chubby quotes

- “Readers familiar with distributed computing will recognize the election of a primary among peers as an instance of the distributed consensus problem”
- “Indeed, all working protocols for asynchronous consensus we have so far encountered have Paxos at their core. “
- “Building Chubby was an engineering effort required to fill the needs mentioned above; it was not research.”

Some design decisions (II)

- Developers are confused by non-intuitive caching semantics, so they prefer consistent caching
- To avoid both financial loss and jail time, they provide security mechanisms, including access control.
- do not expect lock use to be fine-grained, in which they might be held only for a short duration (seconds or less)
- expect coarse-grained use. For example, an application might use a lock to elect a primary, which would then handle all access to that data for a considerable time, perhaps hours or days.

System structure



Chubby Cell

- A small set of servers
- Placed to reduce the likelihood of correlated-failures
- Elect a master
  - Master has a majority vote
  - Oath of faith lasting a few seconds (master lease)
    - No other master can be elected before the lease expires
    - Master lease periodically renewed
      - If master still wins elections
- Replicas maintain copies of a simple DB
  - But only master can initiate reads and writes
  - Replicas copy what the master does

Clients

- Clients find the master by sending master location requests to the replicas listed in the DNS.
- Replicas return the master's id
- Clients then communicate only with the master
  - Until master does not respond
  - Master declares he is no longer the master
- Write requests are propagated using Paxos
  - Acknowledged once a majority of the replicas get the requests
- Read requests fulfilled by the master only

Master failure

- Run election algorithm again once previous master lease expires
  - Takes a few seconds up to half a minute
- If the failed node does not come back online for a few hours
  - A fresh machine from a free pool is started and joins the cell
  - The DNS is updated replacing the old machine with the new one

Chubby Interface

- Resembles a UNIX file system but simpler
- /ls/…. For lock service

Locks

- Locks are advisory
  - they conflict only with other attempts to acquire the same lock
  - holding a lock called F neither is necessary to access the file F, nor prevents other clients from doing so
- Why?
  - protect resources implemented by other services, rather than just the file associated with the lock
  - needed to access locked files for debugging or administrative purposes
  - developers perform error checking in the conventional way

Locking

- Complex in DS
  - Process fails while holding lock and issuing requests
  - Other Processes acquire lock after faults
  - A mechanism for ordering events

- Costly to introduce sequence numbers into all the interactions in an existing complex system.
  - Chubby provides a means by which sequence numbers can be introduced into only those interactions that make use of locks.

Rest of paper

- They go more in to the implementation details
- If you want some insights in to SW development at Google give it a read.
- The paper is accessible in most parts

Distributed Logging

What is the problem?

- Facebook has a lot of servers distributed worldwide.
- The servers produce more than 1 Million logging messages/second.
- Need to analyze them together
  - Maybe using MapReduce
- How do you send them to the MapReduce cluster?
  - Distributed logging

Examples

- **Apache Kafka (LinkedIn)**
- **Apache Flume (Cloudera and others)**
- Chukwa (UCB)
- Scribe (Facebook, now less used)
- (There are more systems)

Kafka

“A distributed messaging system that we developed for collecting and delivering high volumes of log data with low latency.” from the paper

Logged data

- User activity events corresponding to logins, page views, clicks, “likes”, sharing, comments, and search queries;
- Operational metrics such as service call stack, call latency, errors, and system metrics such as CPU, memory, network, or disk utilization on each machine.

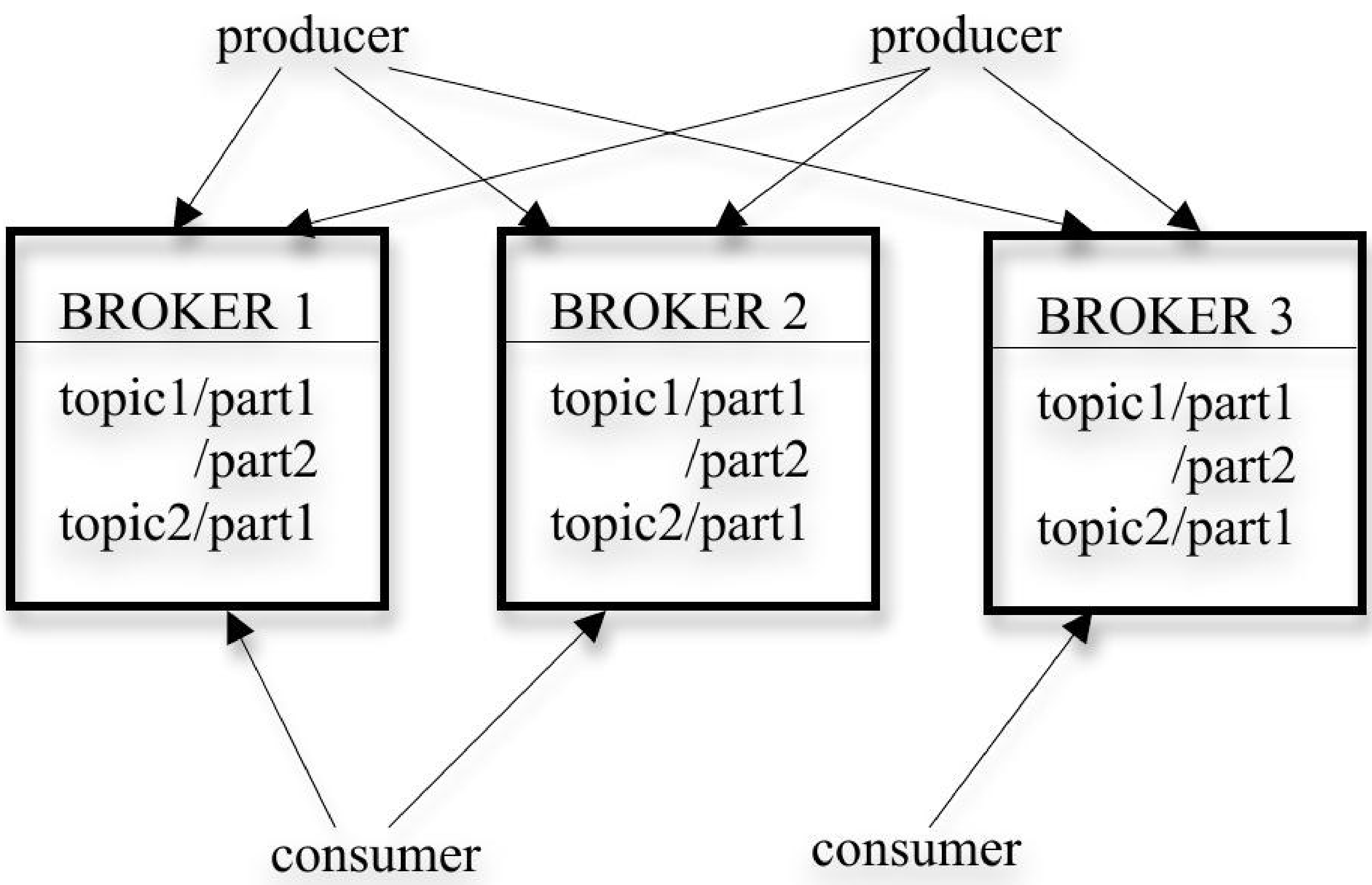
Usage of logged data (LinkedIn)

- Search relevance
- Recommendations
  - driven by item popularity
  - co-occurrence in the activity stream
- Ads
- Security applications
  - Abusive behavior, e.g., Spam
- Newsfeed features that aggregate user status updates or actions for their “friends” or “connections” to read

Challenge

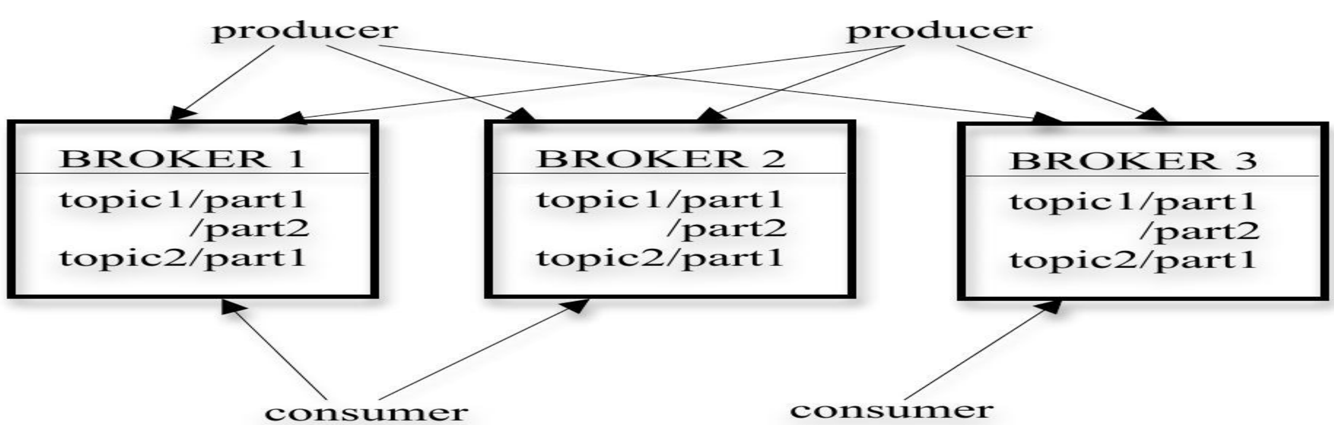
- Log data is larger than “real data”
- It is not just what you click
  - It is also what you did not click
- In 2009, Facebook collected (on average) 6 TB of log data/day

Kafka architecture



Kafka architecture

- Written in Scala
- A stream of messages of a particular type is defined by a topic.
- Producers produce messages
- Published messages stored in broker
- Consumer consumes message



Sample producer code

- producer = new Producer(...);
- message = new Message(“test message str”.getBytes());
- set = new MessageSet(message);
- producer.send(“topic1”, set);

Sample consumer code

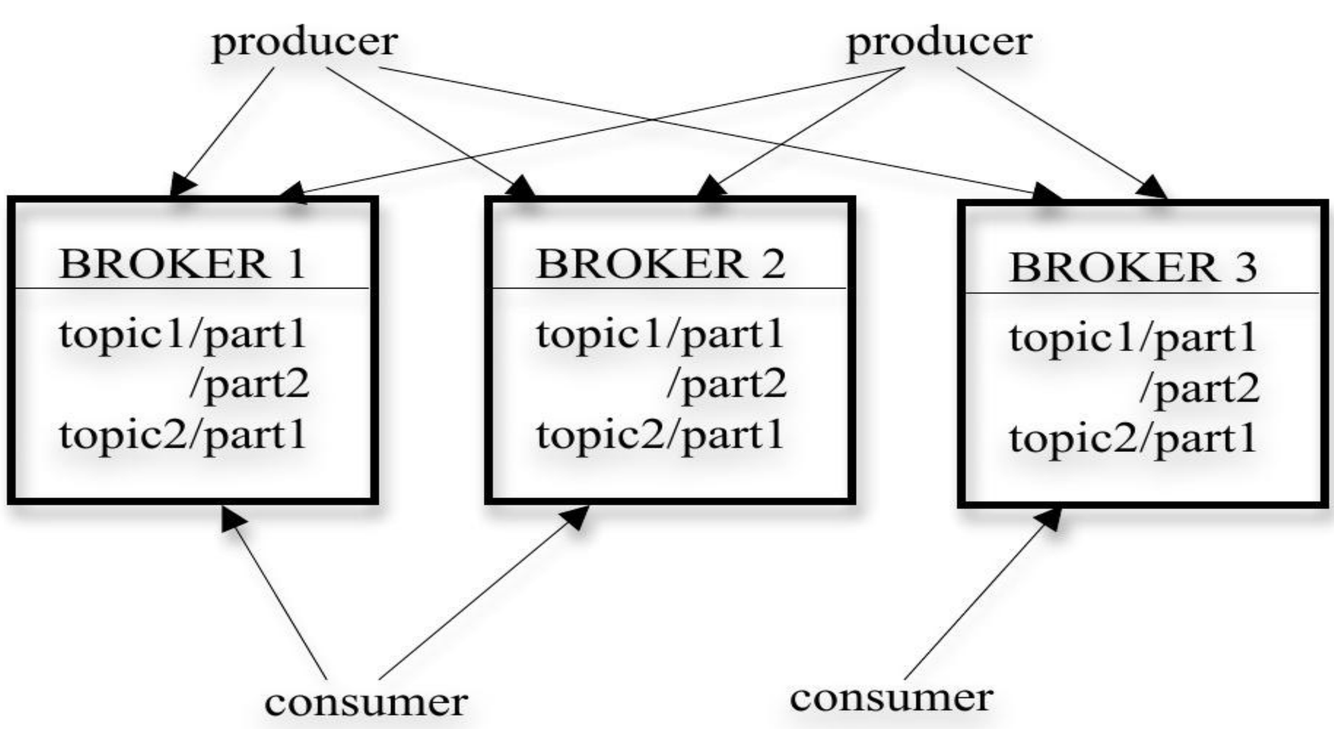
```
streams[] = Consumer.createMessageStreams(“topic1”, 1)
for (message : streams[0]) {
    bytes = message.payload();
    // do something with the bytes
    }
```

Message streams

- Unlike traditional iterators, the message stream iterator never terminates.
  - If there are currently no more messages
    - block until new messages are published to the topic.
- Both point-to-point delivery model
  - multiple consumers jointly consume a single copy of all messages in a topic
- and publish/subscribe model
  - multiple consumers each retrieve its own copy of a topic.

Load balancing

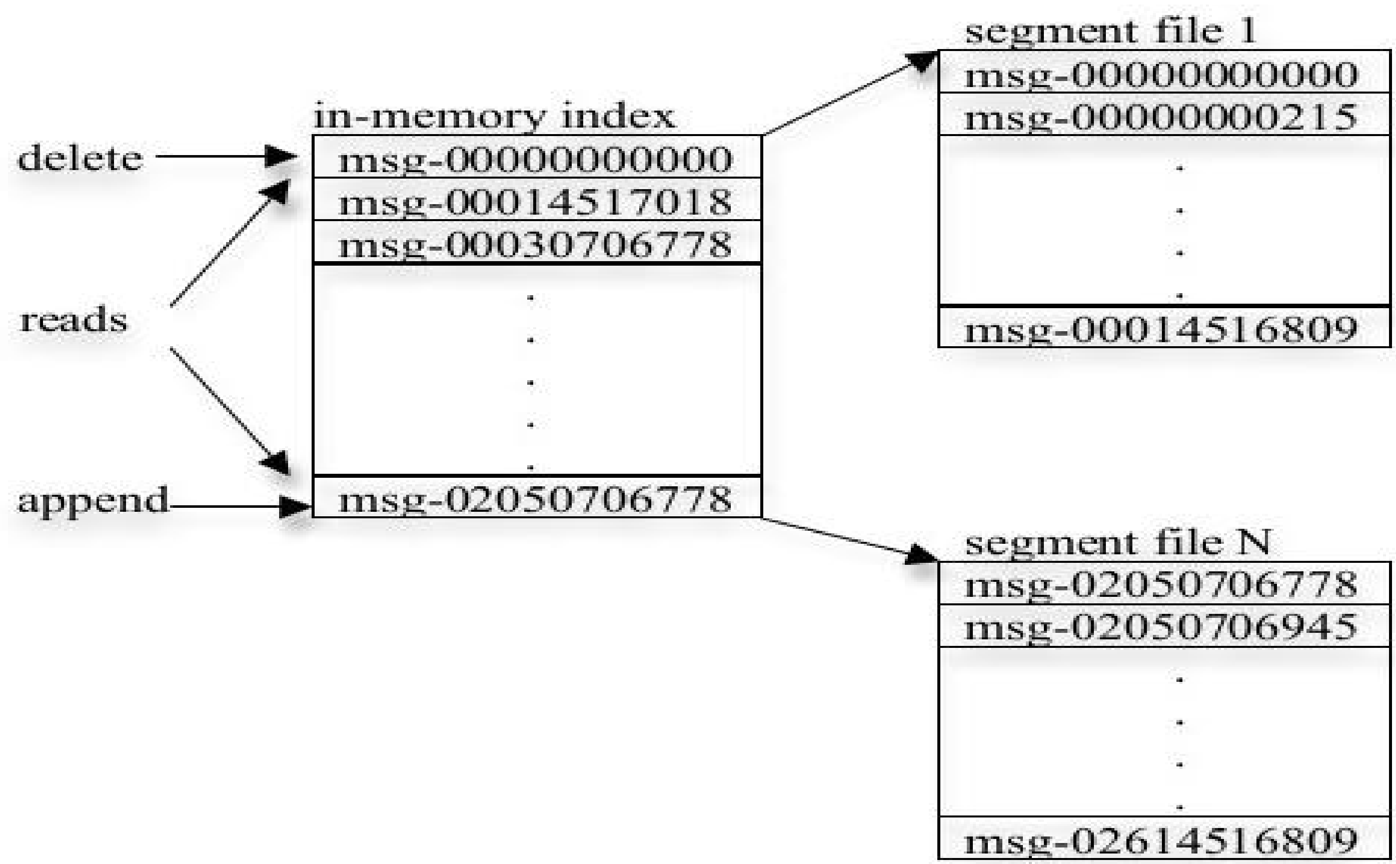
- Divide topic into partitions
  - Each broker stores one or more copies of the partition



Partition

- Simple storage
  - One partition==one (logical) log
  - One (logical) log==a set of segment files of approximately the same size
- One (segment) file open for writing/partition
  - Append new messages to that file
  - fush the segment files to disk only after
    - a configurable number of messages have been published
    - or a certain amount of time has elapsed.
  - A message is only exposed to the consumers after it is flushed.
- Messages addressed by their offset in the log
  - No special id
  - Message id+(message length)=next message id

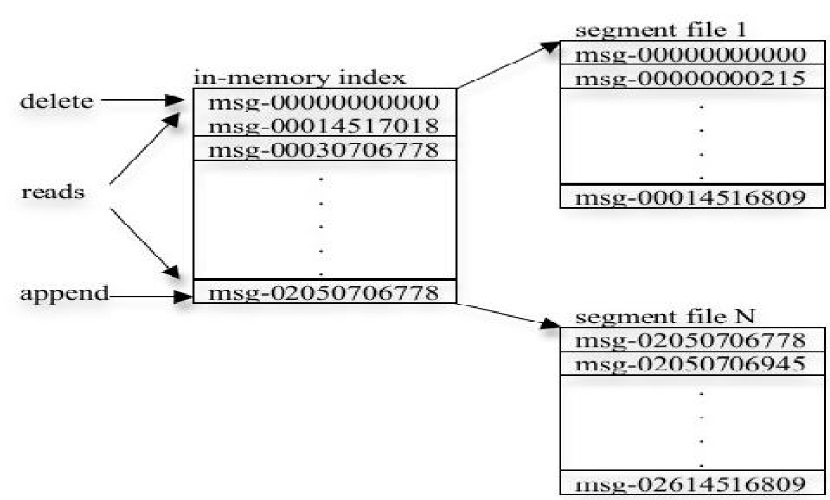
Kafka log



delete
reads
append
in-memory index
msg-00000000000
msg-00014517018
msg-00030706778
msg-02050706778
segment file 1
msg-00000000000
msg-00000000215
msg-00014516809
segment file N
msg-02050706778
msg-02050706945
msg-02614516809

Message consumption

- Consumer always consumes messages from a particular partition sequentially
- Consumer acknowledges a particular message offset
  - He received all messages prior to that offset in the partition.



Message consumption

- Brokers keep sorted list of offsets
  - Including offset of the first message in every segment file
- Under the covers,
  - The consumer is issuing asynchronous pull requests to the broker to have a buffer of data ready for the application to consume.
  - Each pull request contains the offset of the message from which the consumption begins and an acceptable number of bytes to fetch
  - After a consumer receives a message, it computes the offset of the next message to consume and uses it in the next pull request

Stateless broker

- Broker does not keep track of who consumed what
  - It is the consumers who should keep track of what they have consumed
- But then how do you delete something if you are not sure that all consumers have already used it?
  - Retention policy
    - Your message is safe and sound for X time units (typically 7 days)
    - Most consumers consume their message daily, hourly or in real time

Stateless broker

- Does performance degrade with larger stored data size?
  - No since you consume using offsets, and files are kept within limits, e.g., 1 GB.
- A consumer can deliberately rewind back to an old offset and re-consume data.
  - Violates the common contract of a queue,
  - but proves to be an essential feature for many consumers.
  - For example, when there is an error in application logic in the consumer, the application can re-play certain messages after the error is fixed.

Distributed coordination

- Consumer groups
  - Those interested in the same topic(s)
- No coordination needed between consumer groups
- Decision 1:
  - a partition within a topic the smallest unit of parallelism
  - All messages from one partition are consumed only by <u>a single consumer within each consumer group</u>.
  - No locking and no state-maintenance overhead

Distributed coordination

- For the load to be truly balanced,
  - Many more partitions are needed in a topic than the consumers in each group.
  - Achieve this by over partitioning a topic.

Distributed coordination

- Decision 2:
  - No master (central) node
  - No worries about master failures
- Use ZooKeeper to facilitate the coordination

ZooKeeper usage in Kafka

- Detect the addition/removal of brokers and consumers
- Trigger a rebalance process in each consumer when a new broker/consumer added
- Maintaining the consumption relationship and keeping track of the consumed offset of each partition.

ZooKeeper usage in Kafka

- When each broker or consumer starts up
  - stores its information in a broker or consumer registry in Zookeeper.
- The broker registry contains the broker’s host name and port, and the set of topics and partitions stored on it.
- The consumer registry includes the consumer group to which a consumer belongs and the set of topics that it subscribes to.
- Each consumer group is associated with an ownership registry and an offset registry in Zookeeper.
  - The ownership registry has one path for every subscribed partition and the path value is the id of the consumer currently consuming from this partition
  - The offset registry stores for each subscribed partition, the offset of the last consumed message in the partition.

Kafka usage at LinkedIn

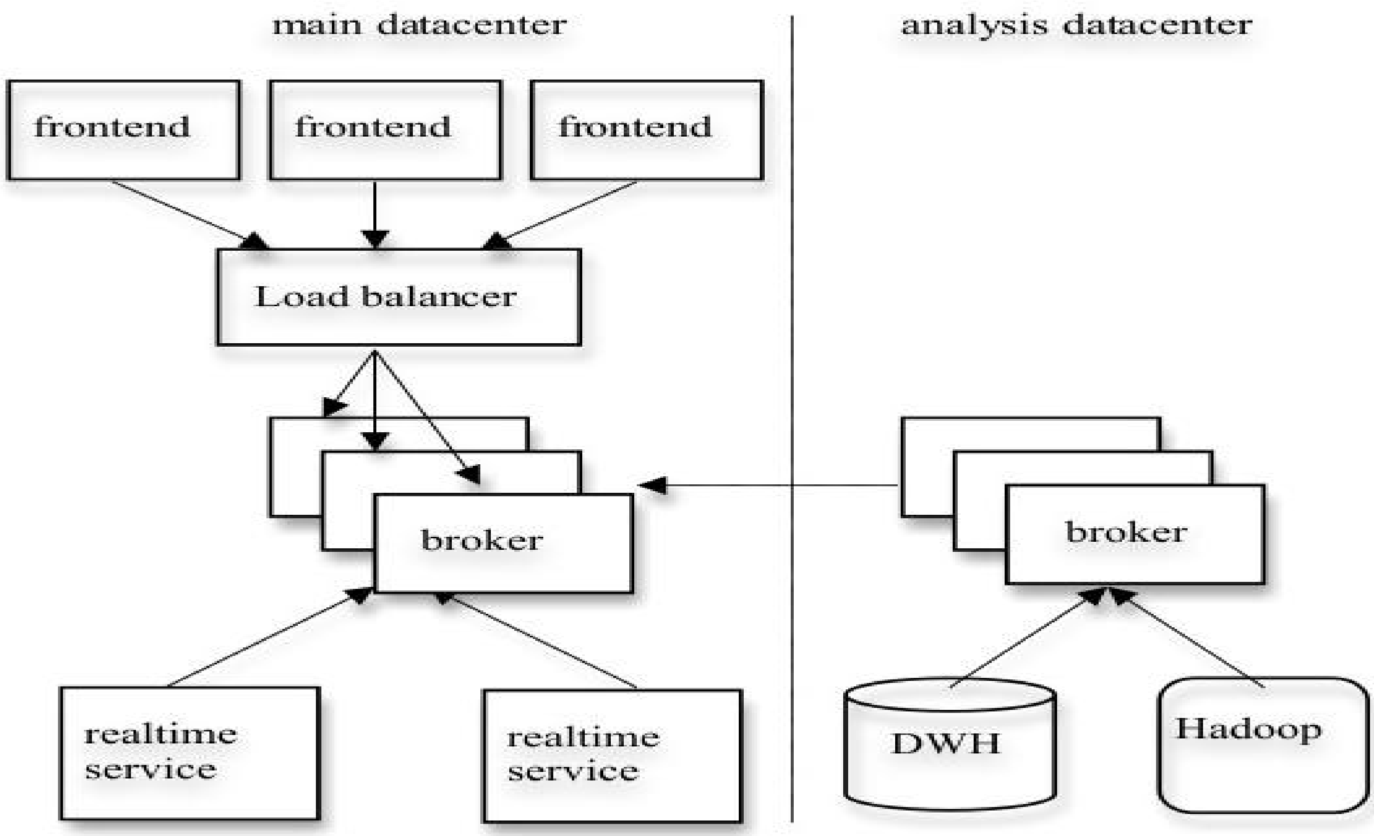


Figure 3. Kafka Deployment

Kafka usage at linkedIn

- One Kafka cluster co-located with each datacenter
- The frontend services generate various kinds of log data and publish it to the local Kafka brokers in batches
-

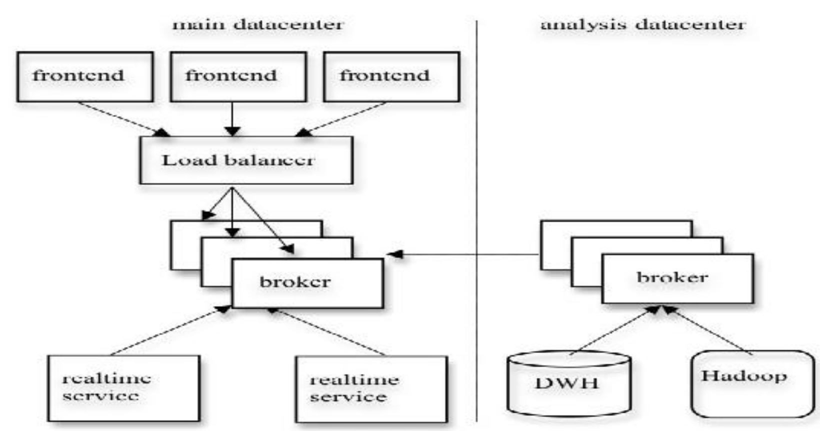


Figure 3. Kafka Deployment

Kafka usage at linkedIn

- Another deployment in an analysis center
- Kafka accumulates hundreds of gigabytes of data and close to a billion messages per day (2011).

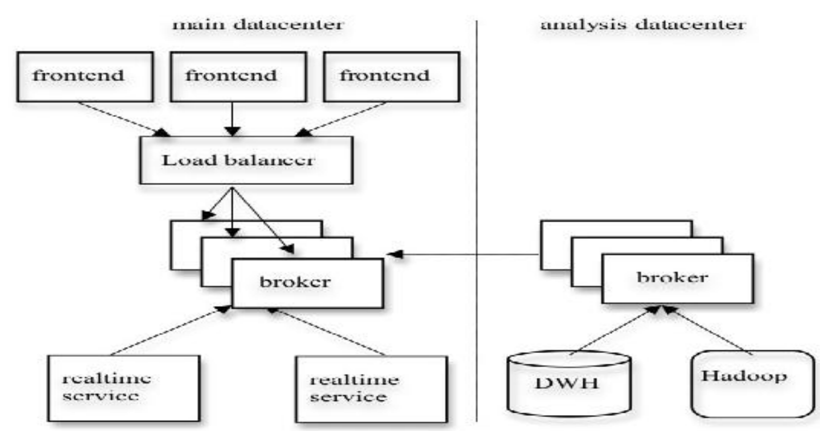


Figure 3. Kafka Deployment

Questions?