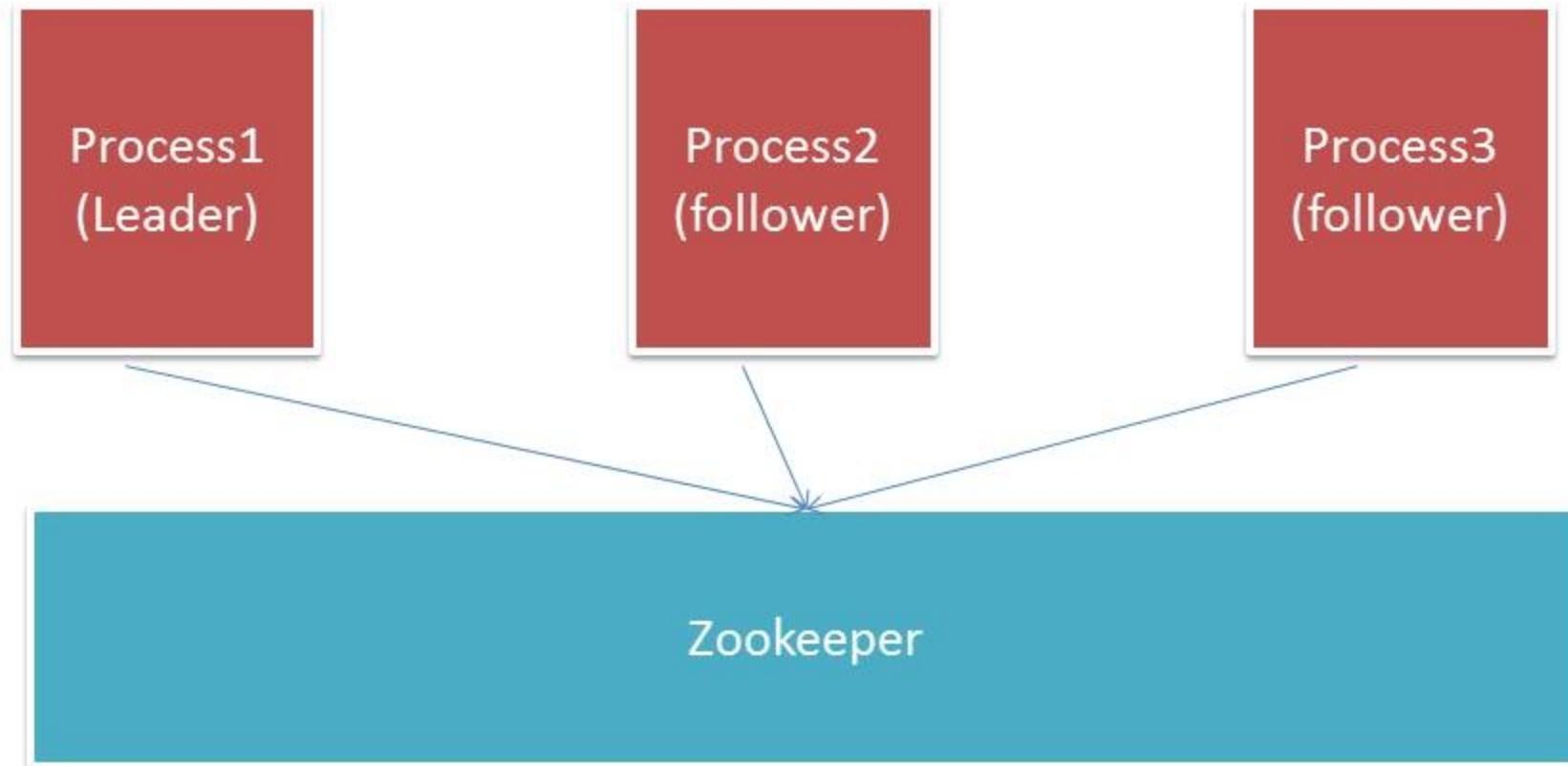


Welcome to the World of Distributed Messaging Queue

What is zookeeper?

- An open source, high performance coordination service for distributed application.
- Role of zookeeper
 - Configuration management
 - Distributed cluster management
 - Node join/leave
 - Leader election in distributed application
 - Distributed synchronization - Lock

Leader election

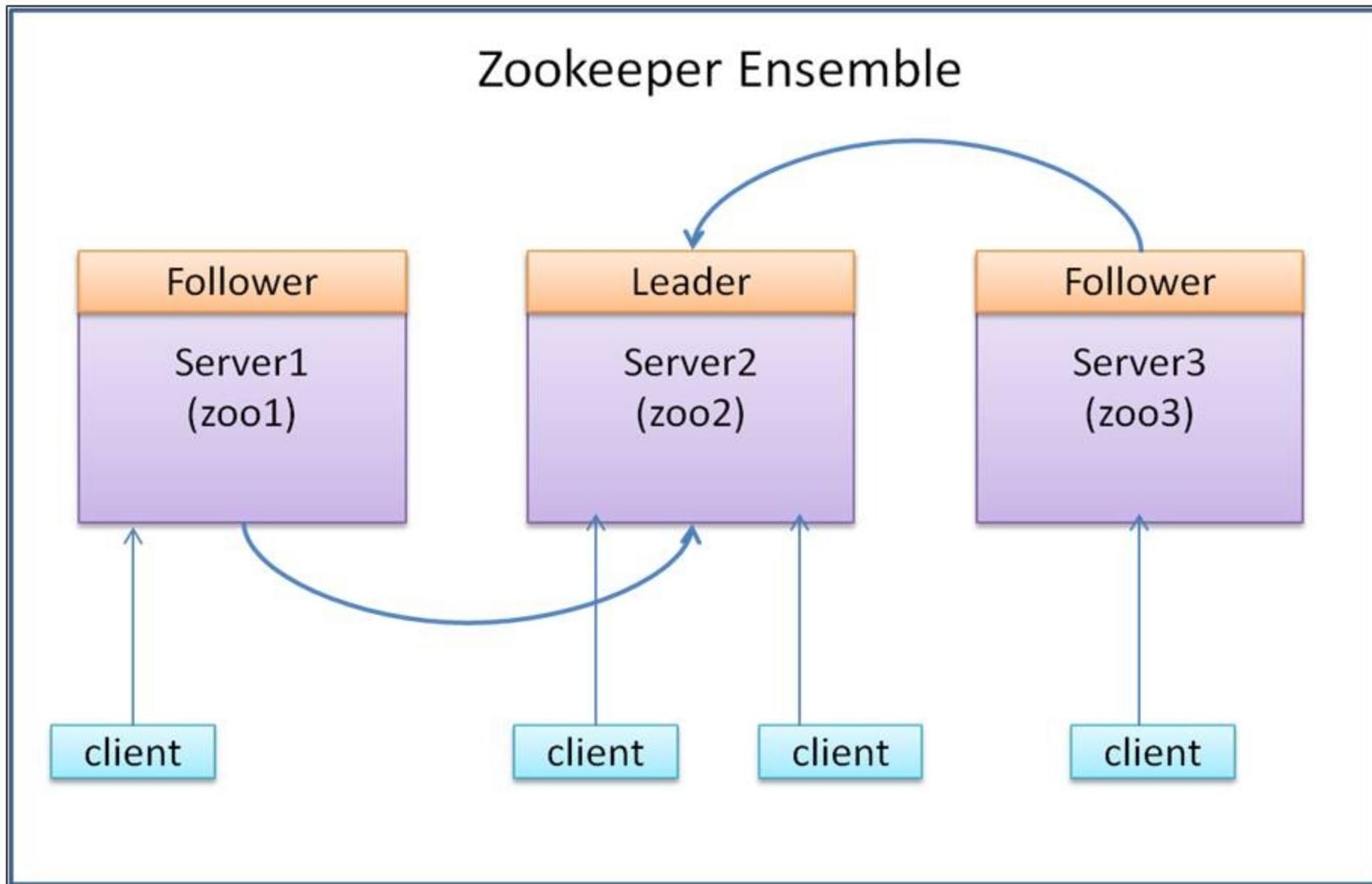


Applications and Companies which are using Zookeeper



- One node in the cluster acts as the leader, while the rest are followers.
- All write requests come from clients are forwarded to leader node while follower's nodes only handle the read request.
- Limitation of Zookeeper?
 - We can't increase the write performance of Zookeeper ensemble by increase the number of nodes because all write operations go through the leader.

Zookeeper Architecture



1. Clients only connect to a single zookeeper server.
2. Client can read from any zookeeper node but writes go through the leader node.

- **dataDir**:- The directory to store the in-memory database snapshots and transactional log.
- **clientPort**:- The port used to listen for client connections.

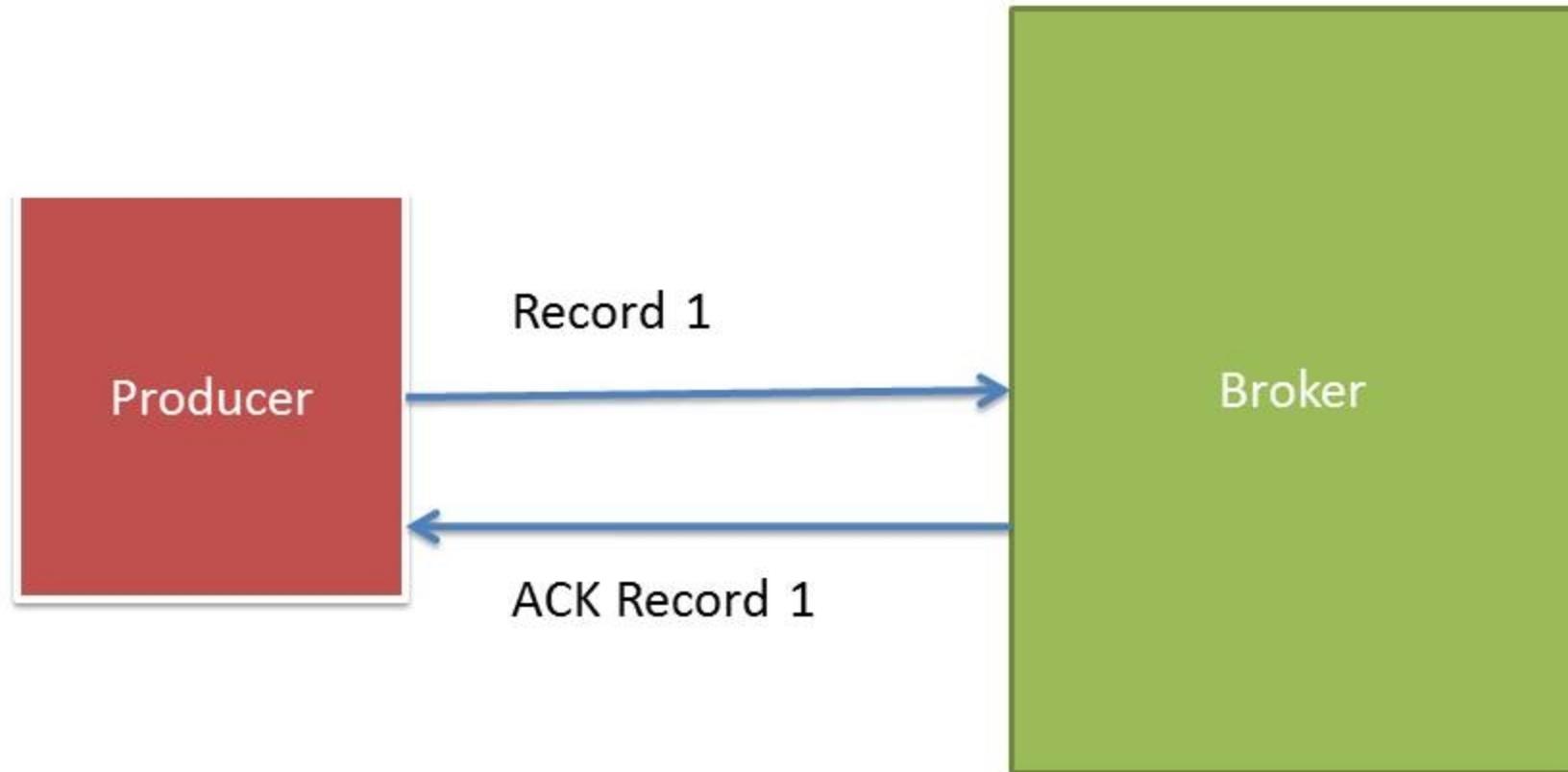
- Download the latest stable Zookeeper release from the Zookeeper site (<http://zookeeper.apache.org/releases.html>). At this moment the latest version is zookeeper-3.4.6.
- Set ZK_HOME environment variable on each machine
- Create the configuration file zoo.cfg at \$ZK_HOME/conf directory on each machine

- Add the following properties to zoo.cfg file on each machine:
tickTime=2000
dataDir=/var/zookeeper
clientPort=2181
initLimit=5
syncLimit=2
server.1=zoo1:2888:3888
server.2=zoo2:2888:3888
server.3=zoo3:2888:3888
- Where, zoo1, zoo2 and zoo3 are the IP's of Zookeeper nodes.

- Types of Producer
 - Sync
 - Async
- Sync : Producer push message in Kafka and wait for an acknowledgement.
- Async: Producer continuous push messages into Kafka without waiting for acknowledgement

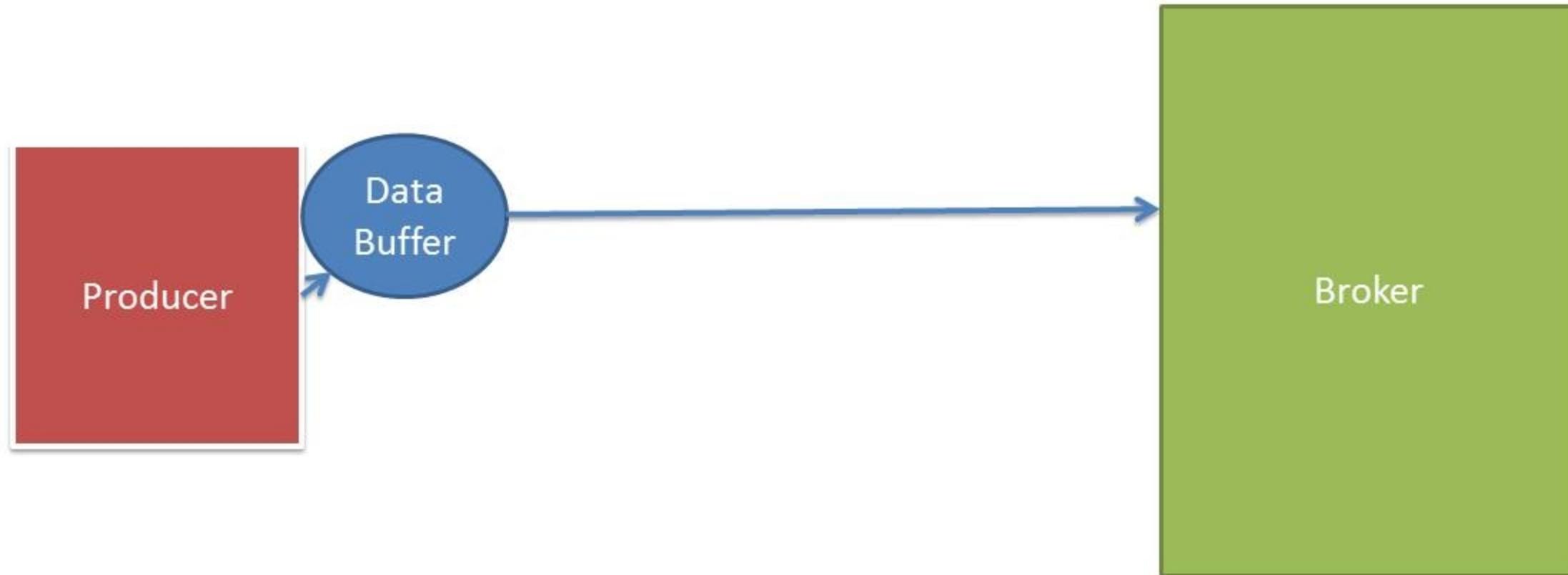
- The **producer.type** property is used to Define the producer type.
- Valid values are (1) **async** for asynchronous send and (2) **sync** for synchronous send.
- Data may loss in case of **async** producer.

Sync Producer



- **Async** producer allow to batch the set of messages and store them in one go. By using, Async producer we can achieve the better throughput
- **Aysnc** Producer increase the latency.
 - How?
 - It buffers the data for certain amount of time, which will delay the processing of record.

Async Producer



- The property **batch.num.messages** define the number of message you want to send in one batch.
- The property **queue.buffering.max.ms** define the maximum time to buffer data. For example a setting of 100 will try to batch together 100ms of messages to send at once.
- What will happen? If user set both the properties:
 - The producer will wait until **batch.num.messages** size reach or **queue.buffer.max.ms** is reached.

- The property **queue.buffering.max.messages** define the maximum number of unsent messages that can be queued up the producer when using async mode before either the producer must be blocked or data must be dropped.
- The property **queue.enqueue.timeout.ms** define the amount of time to block before dropping messages when running in async mode and the buffer has reached **queue.buffering.max.messages**.
 - If set to 0 events will be enqueued immediately or dropped if the queue is full (the producer send call will never block).
 - If set to -1 the producer will block indefinitely and never willingly drop a send.

Can we batch the messages in Sync Producer?



- **Answer:** Kafka doesn't support batching in case of sync producer.
- We can perform the manual batching at producer end and sent the list of batches as single Kafka record.
- Data may lost while preparing the batching at producer end.
- We need to write the custom encoder/decoder to covert list into bytes and vice versa. As we need to serialize the data before sending over network

- **request.required.acks:**
 - 0, which means that the producer never waits for an acknowledgement from the broker.
 - 1, which means that the producer gets an acknowledgement after the leader replica has received the data.
 - -1, which means that the producer gets an acknowledgement after all in-sync replicas have received the data.
- **request.timeout.ms:**
 - The amount of time the broker will wait trying to meet the **request.required.acks** requirement before sending back an error to the client.

- Push data into Kafka
- Consumer to consume data from Kafka
- Again, start pushing data into Kafka and abruptly killed the producer
- Some data will be lost

- Push the batch of 10 records in each send request.
- We need to write the custom serializer/deserializer to serialize list into bytes.
- We can write serializer by implements the kafka.serializer.Encoder interface.

- All the replica which are in sync with leader is called In sync replica
- The leader keeps track of the set of "in sync" nodes. If a follower dies, gets stuck, or falls behind, the leader will remove it from the list of in sync replicas.
- How leader will identify, whether the replica is in-sync with leader or not?

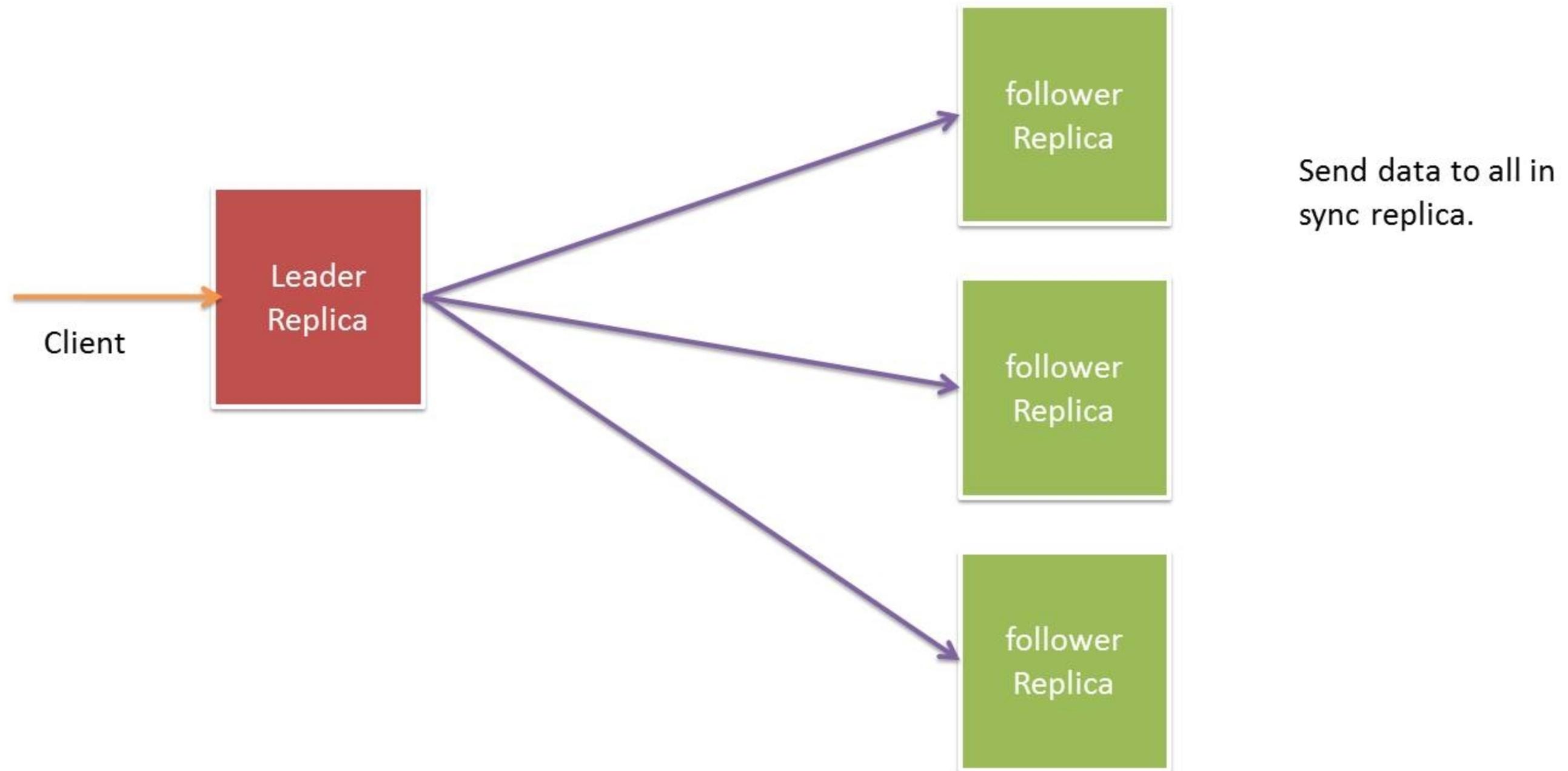
- **replica.lag.max.messages**
 - If a replica falls more than this many messages behind the leader, the leader will remove the follower from ISR and treat it as dead.
- **replica.lag.time.max.ms**
 - If a follower hasn't sent any fetch requests for this window of time, the leader will remove the follower from ISR (in-sync replicas) and treat it as dead.

- This ISR set is persisted to ZooKeeper whenever it changes. Because of this, any replica in the ISR is eligible to be elected leader.
- A message is considered "committed" when all in sync replicas for that partition have applied it to their log.
- Only committed messages are ever given out to the consumer. This means that the consumer need not worry about potentially seeing a message that could be lost if the leader fails.

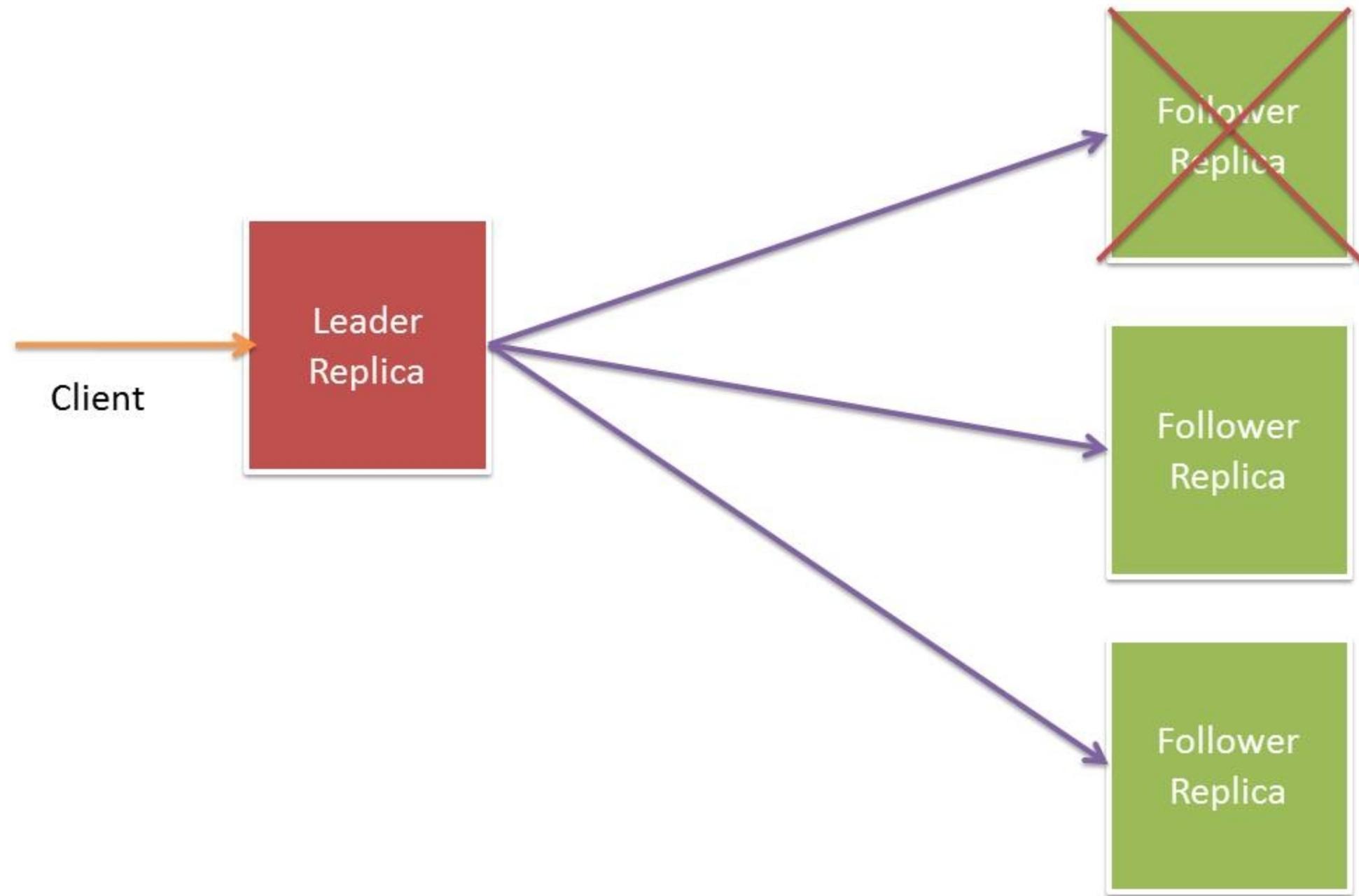
- There are two typical approaches of maintaining strongly consistent replicas:
 - The primary-backup approach
 - The quorum-based approach

- Primary Approach:
 - In primary-backup replication, the leader waits until the write completes on every replica in the group (ISR) before acknowledging the client. **If one of the replicas is down, the leader drops it from the current group and continues to write to the remaining replicas.** A failed replica is allowed to rejoin the group if it comes back and catches up with the leader.

Primary Replication

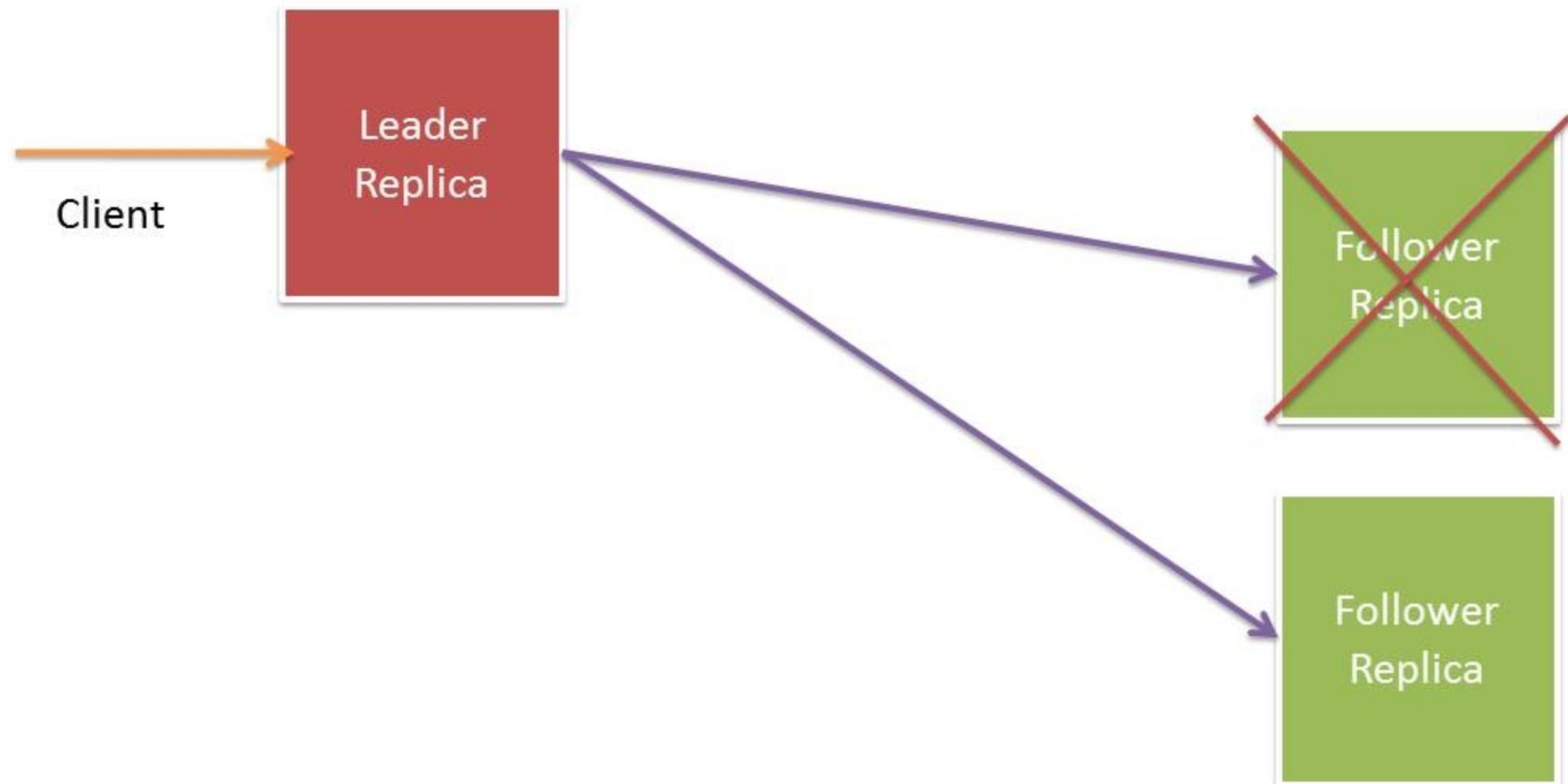


Primary Replication



- Send data to all in sync replica.
- If follower is down or lack behind with leader, then it will remove it from the list of ISR
- Sync Producer continue able to push data

Primary Replication



- Send data to all in sync replica.
- If follower is down or lack behind with leader, then it will remove form the list of ISR
- Sync Producer continue able to push data even single node is running

DataFlair Web Services Pvt Ltd

+91-8451097879

info@data-flair.com

<http://data-flair.com>