

# [544] Kafka Reliability

Tyler Caraza-Harter

# Learning Objectives

- describe how leader and follower replicas work in Kafka (to record messages, handle failover, etc.)
- apply the definition of "committed" messages to reason about when messages (a) are acknowledged to producers and (b) can be read by consumers
- configure Kafka and write producer/consumer code to achieve "exactly once semantics"

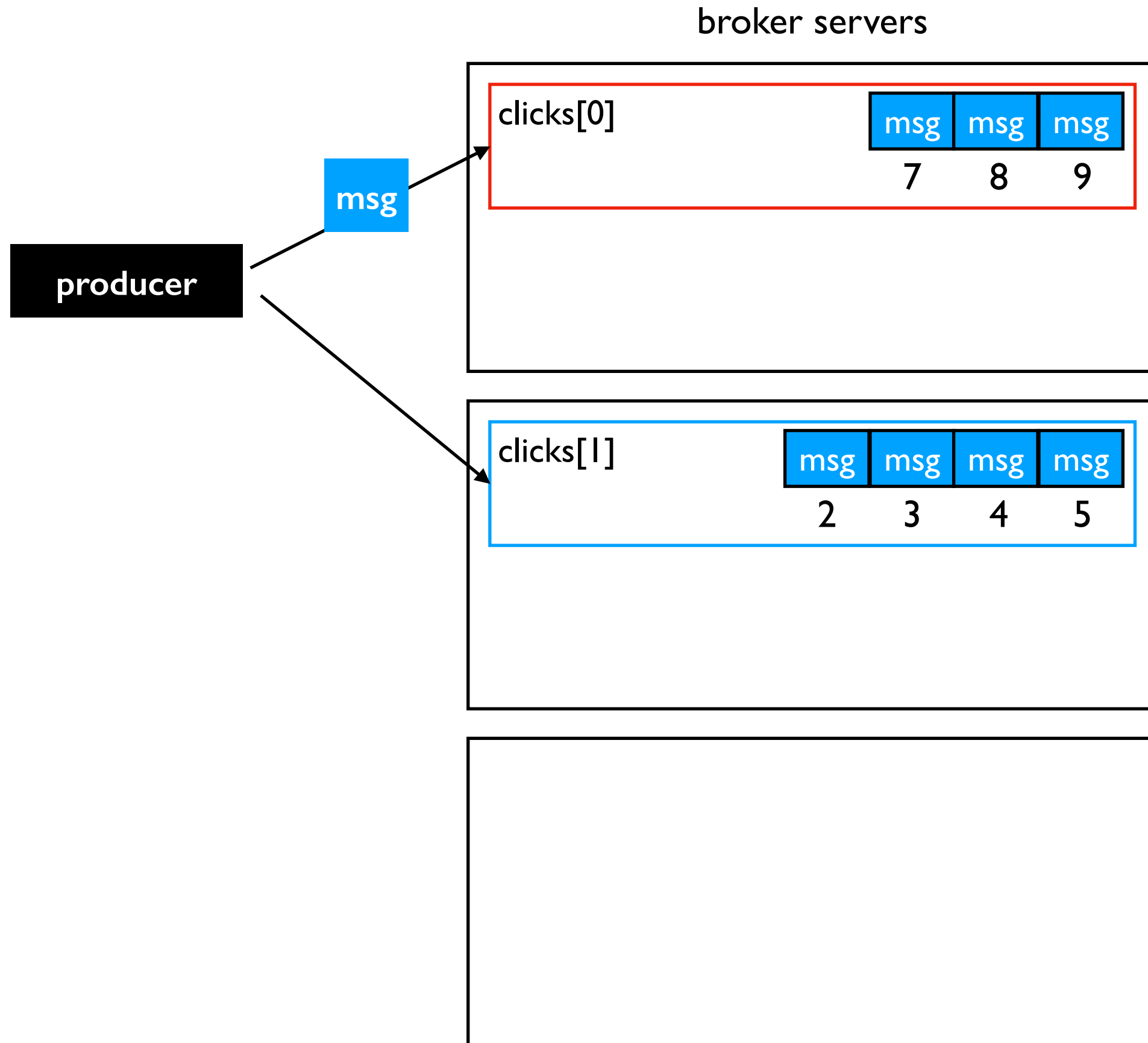
# Outline: Kafka Reliability

Kafka Replication

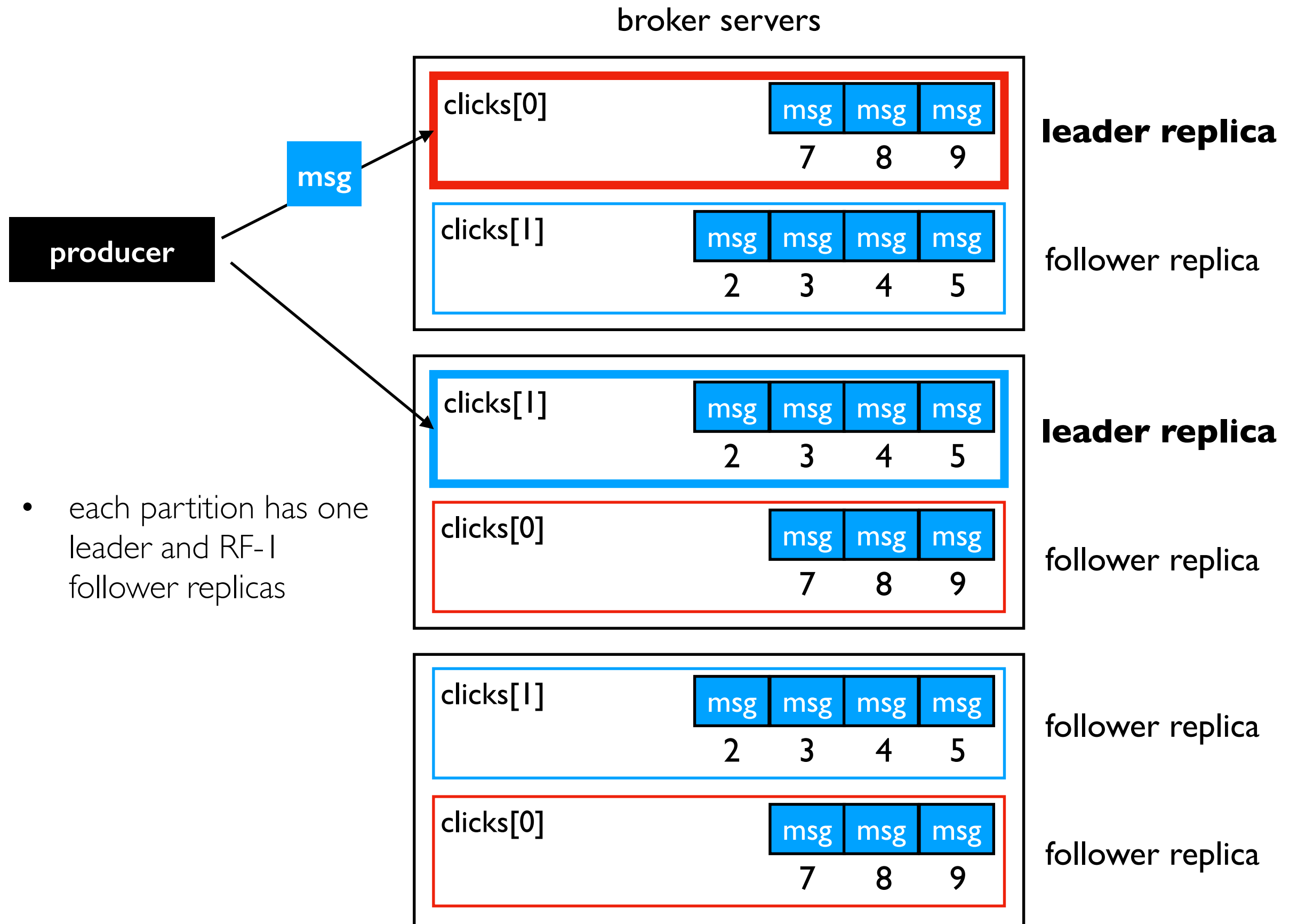
Fault Tolerance

Exactly-Once Semantics

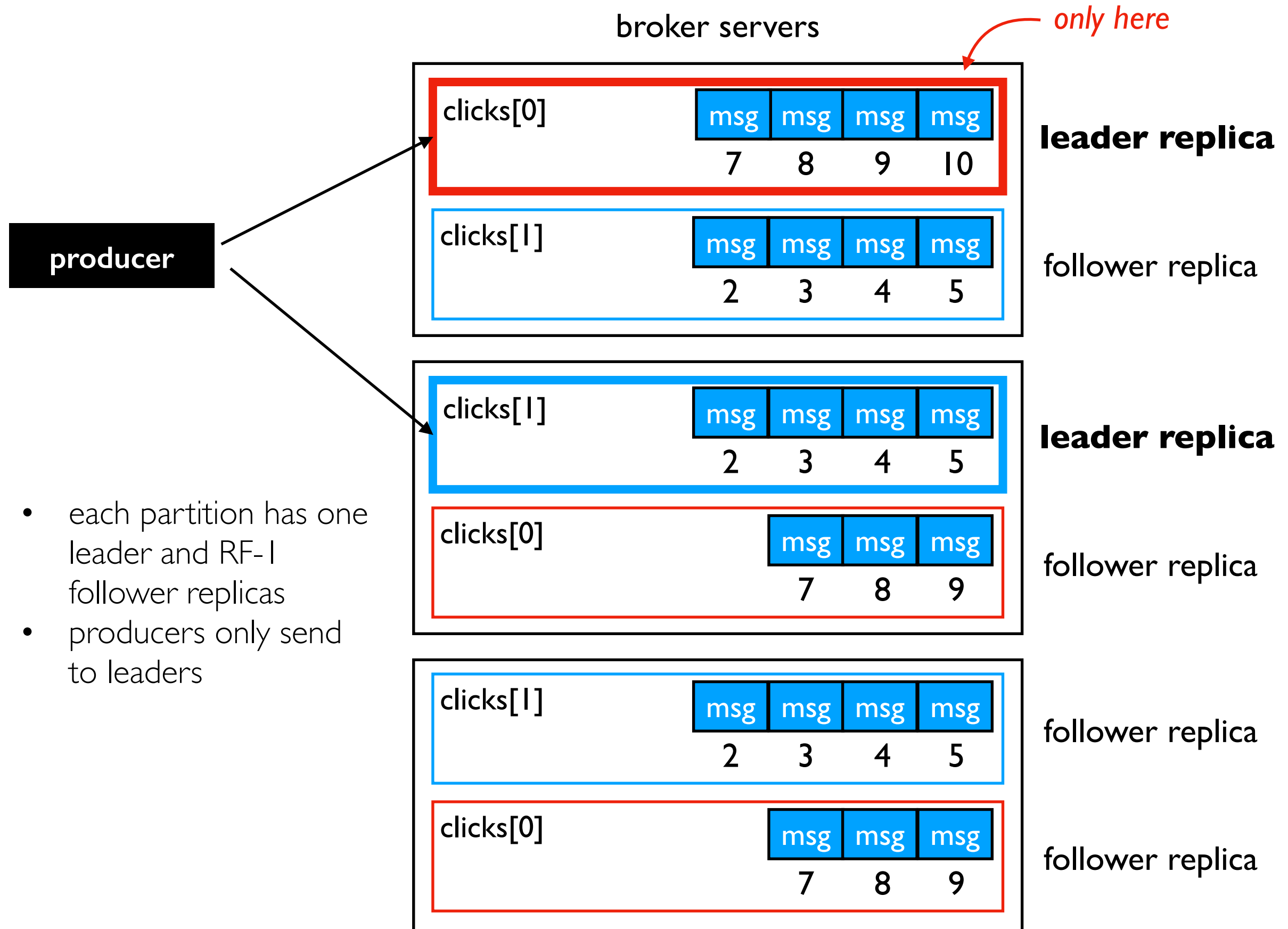
# Three brokers, 2 partitions, replication factor=1



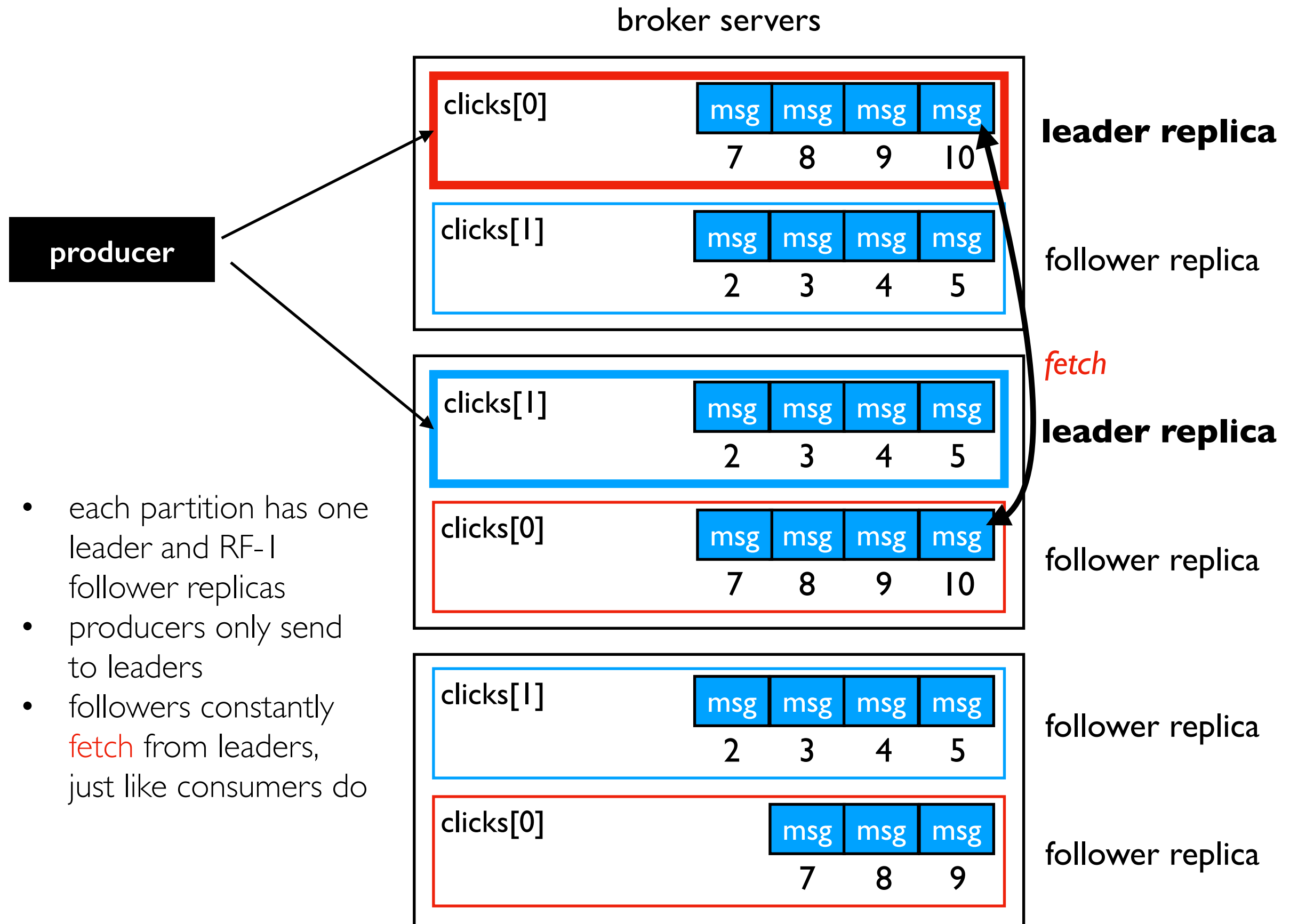
# Three brokers, 2 partitions, replication factor=3



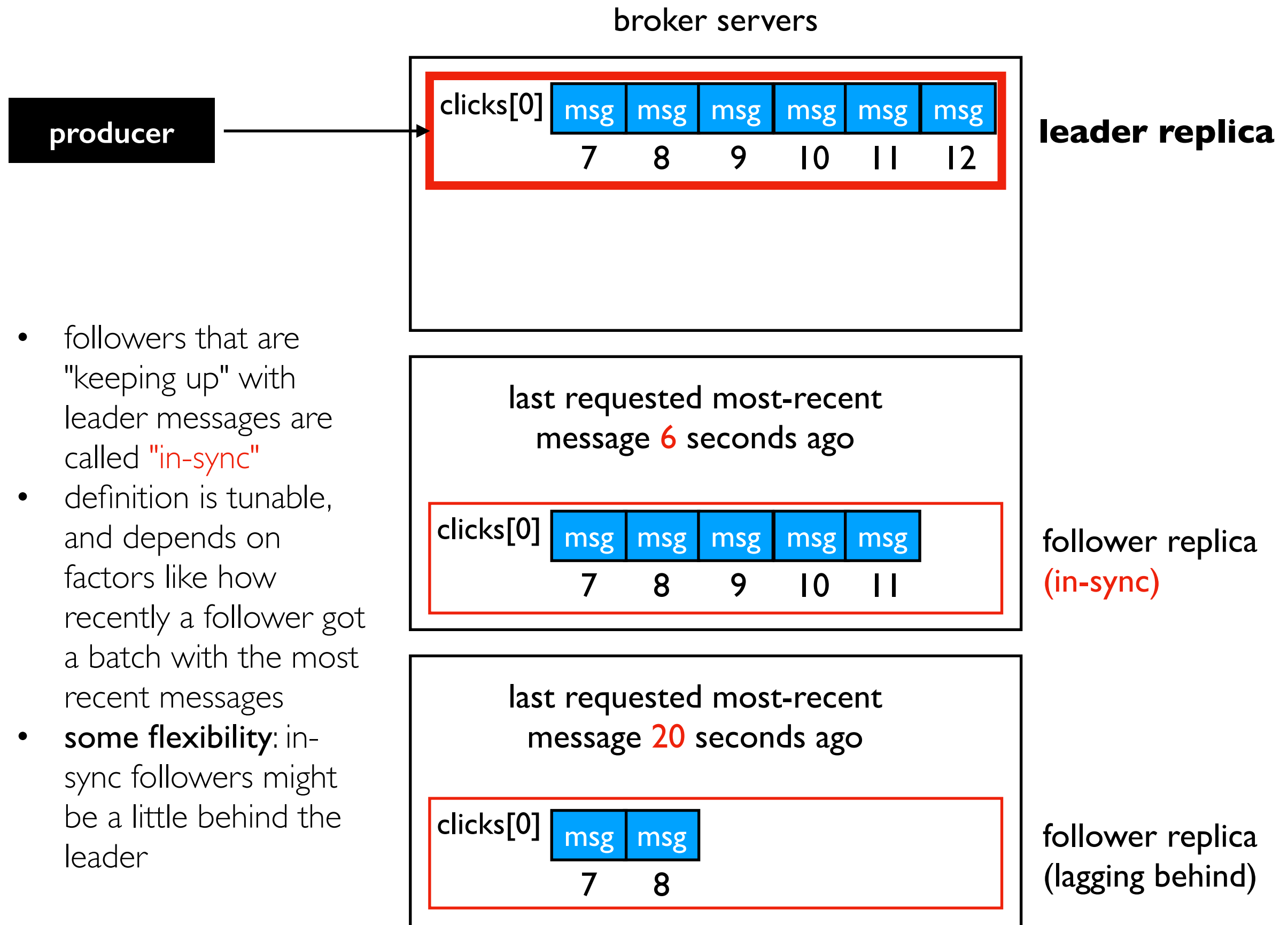
# Three brokers, 2 partitions, replication factor=3



# Fetch Requests

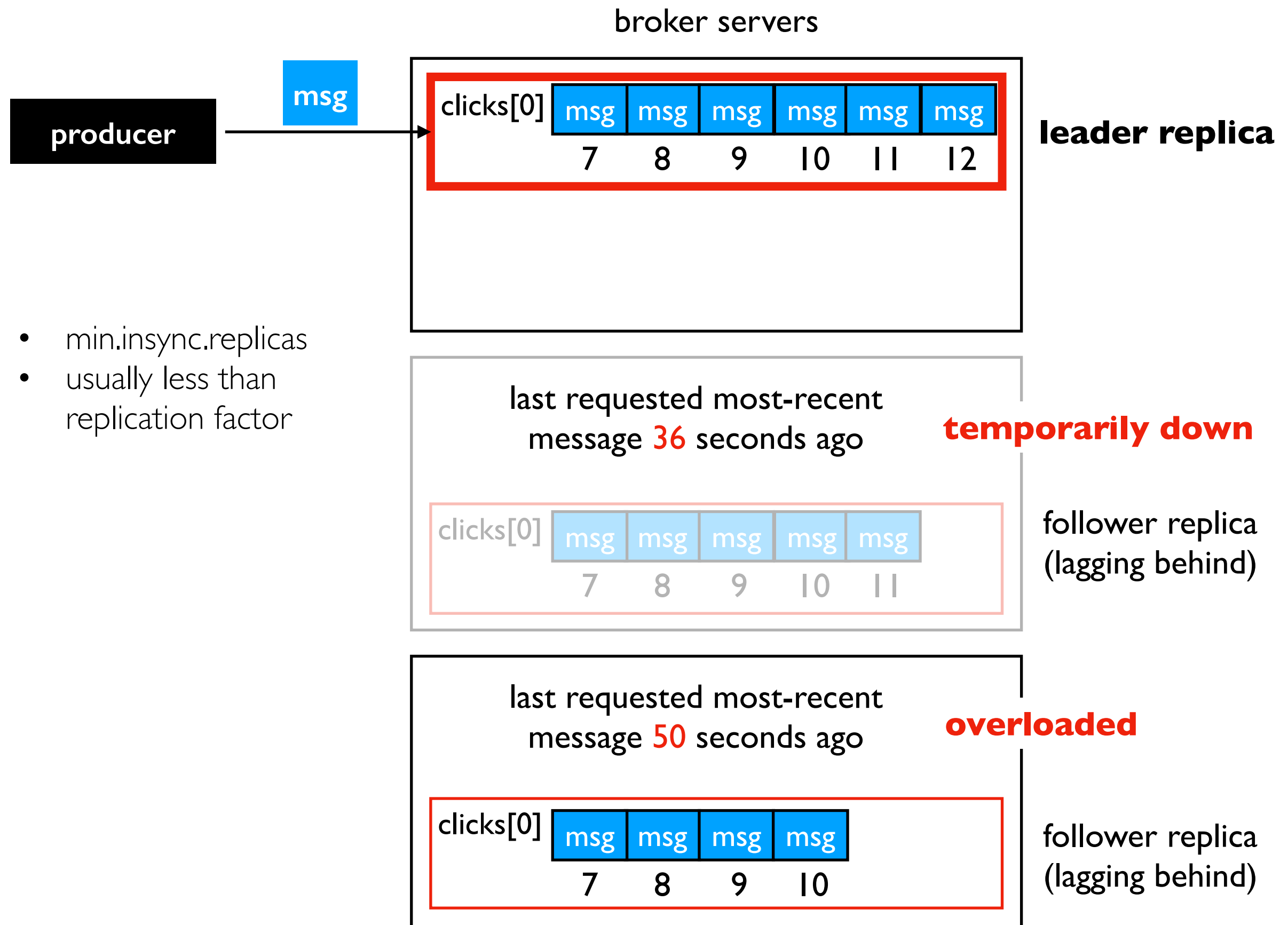


# Followers: In-Sync and/or Lagging Behind

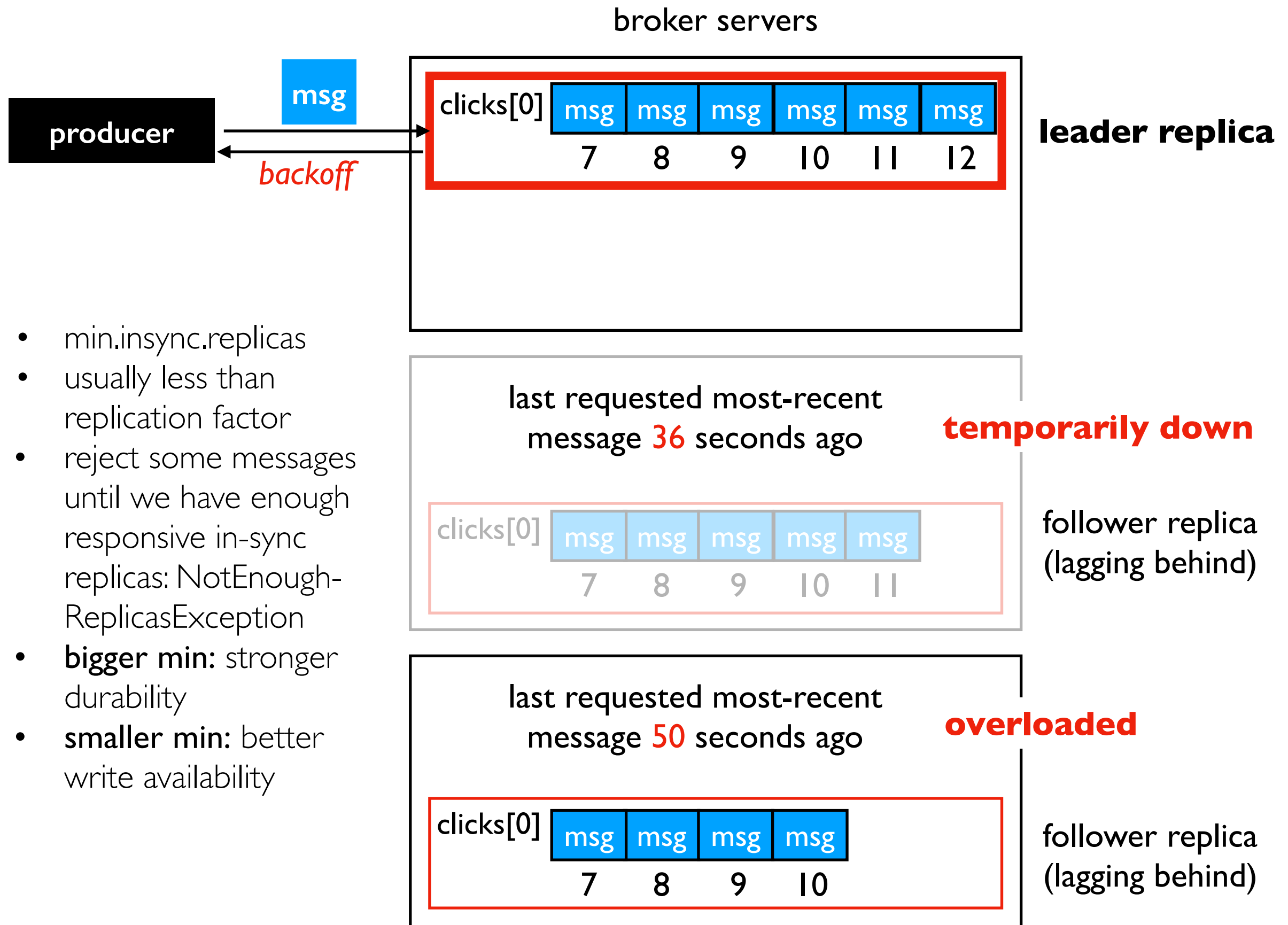




# Minimum In-Sync Replicas (Assume 2 Here)



# Backoff: Not Enough Replicas Exception



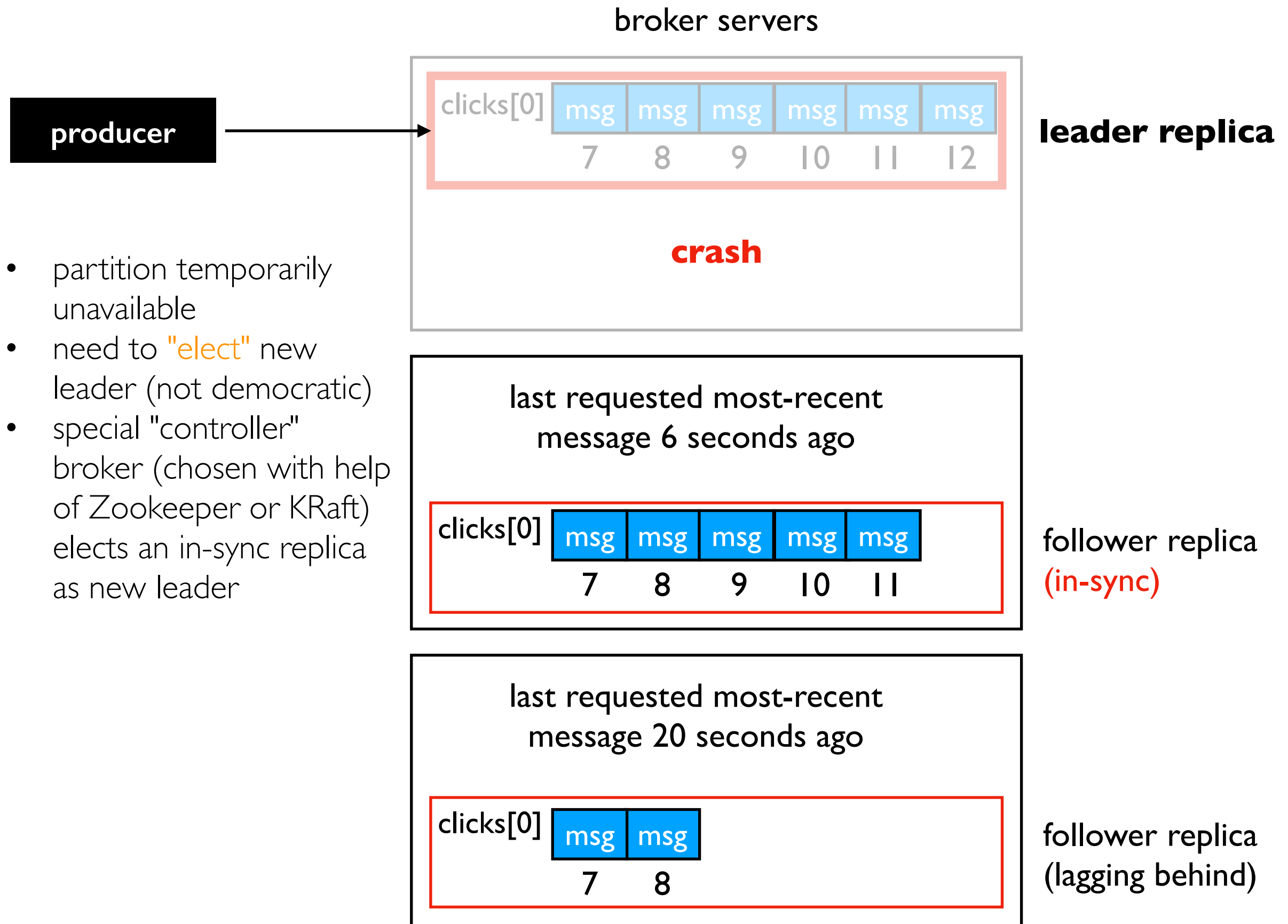
# Outline: Kafka Reliability

Kafka Replication

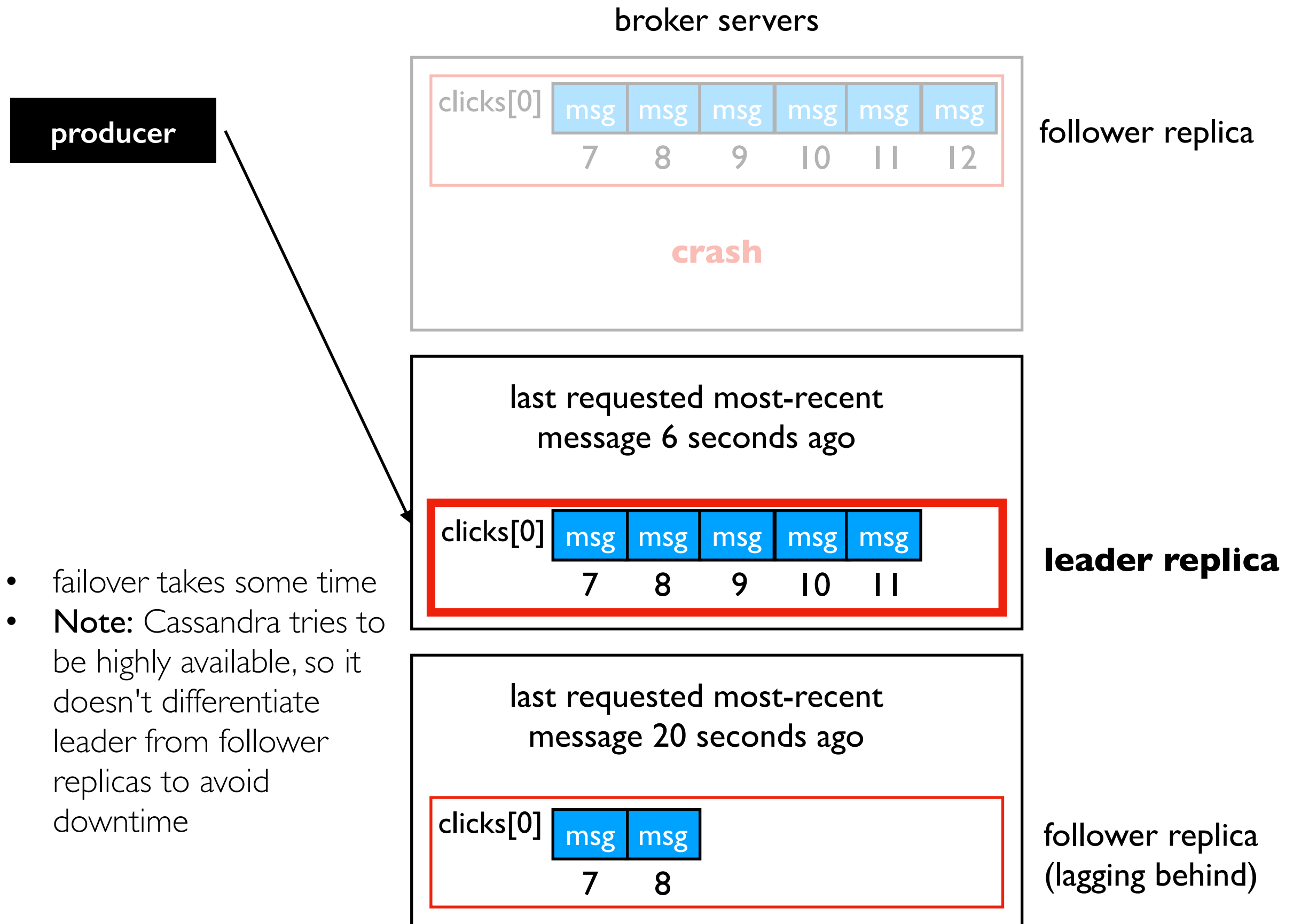
Fault Tolerance

Exactly-Once Semantics

# What if the leader fails? Elect a new one!



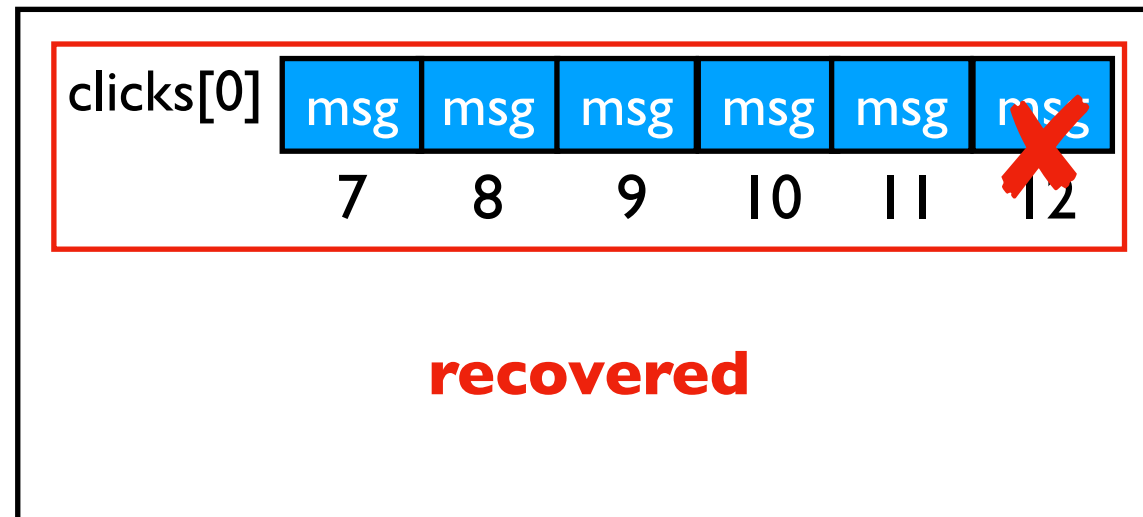
# Kafka Replica Failover



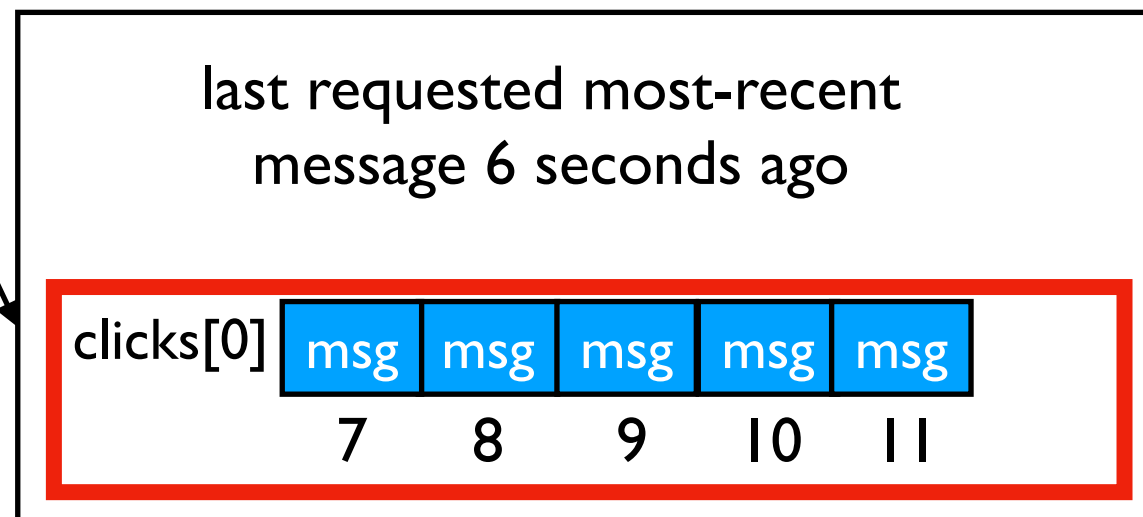
# Some Messages Seen by Old Leader Lost

broker servers

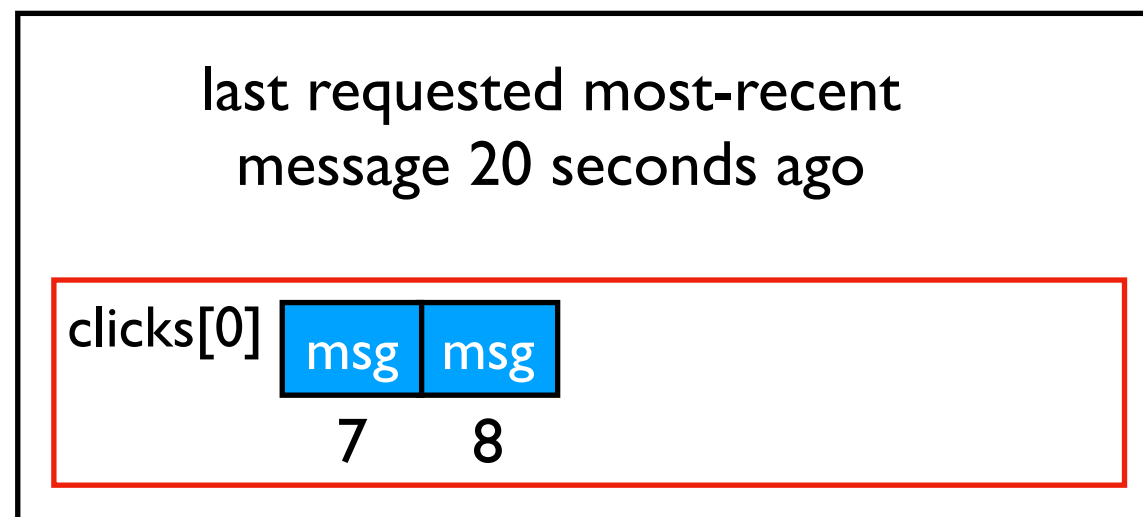
**producer**



follower replica



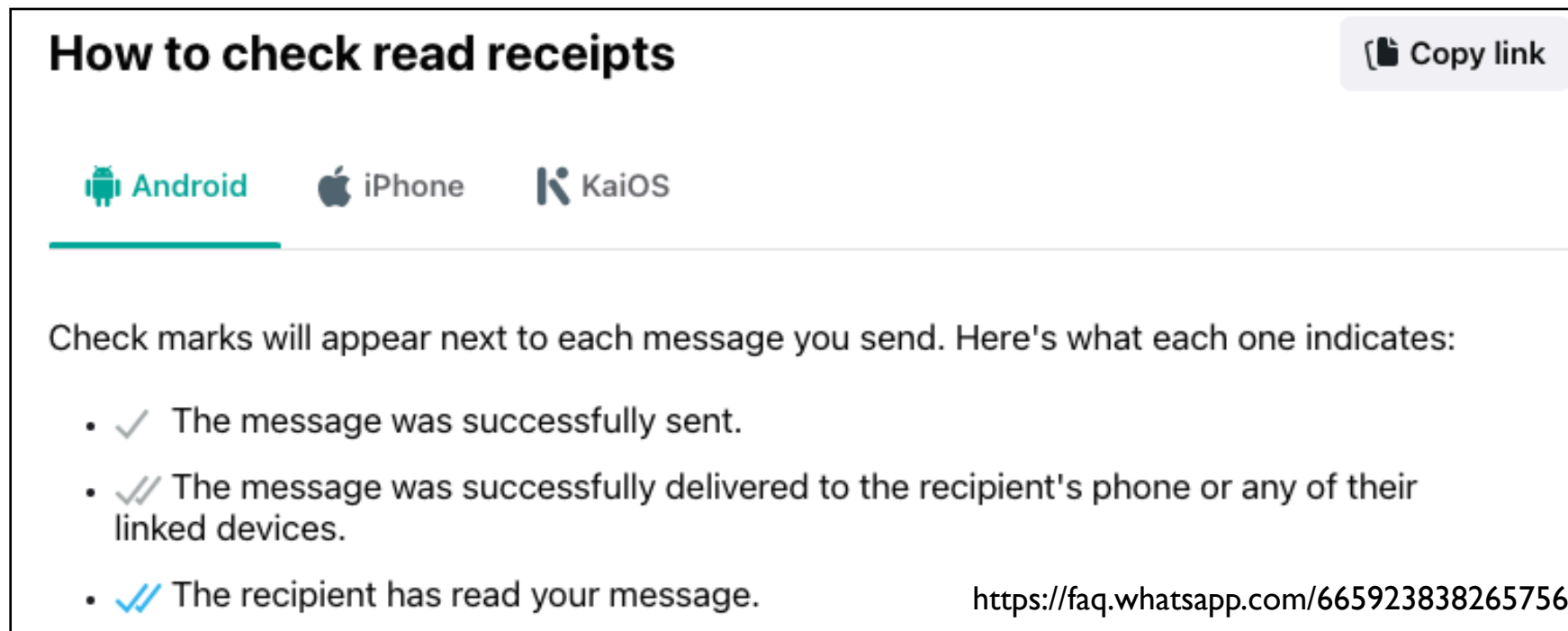
**leader replica**



follower replica  
(lagging behind)

- new leader decides what goes at each offset
- it probably wrote different messages at some offsets than what old leader wanted there
- old leader doesn't immediately get its job back upon recovery

# Review "Committed":WhatsApp Acks Example



these are examples of "acks" (acknowledgements)

In distributed storage systems/databases, an ack means our data is *committed*.

"Committed" means our data is "safe", even if bad things happen. The definition varies system to system, based on what bad things are considered. For example:

- a node could hang until rebooted; a node's disk could permanently fail
- a rack could lose power; a data center could be destroyed

In Kafka's leader/follower replica design, what are some "bad things" we might worry about?

# Kafka: Committed Messages

Messages are "committed" when written to ALL in-sync replicas.

Depending on how many are in-sync, the strength of the guarantee can vary, but `min.insync.replicas` lets us specify a worst case.

If number of concurrent broker failures  $< \text{min.insync.replicas}$ , then our committed data is safe, even if the leader fails (because we can elect another in-sync replica, and all in-sync replicas have all committed data).

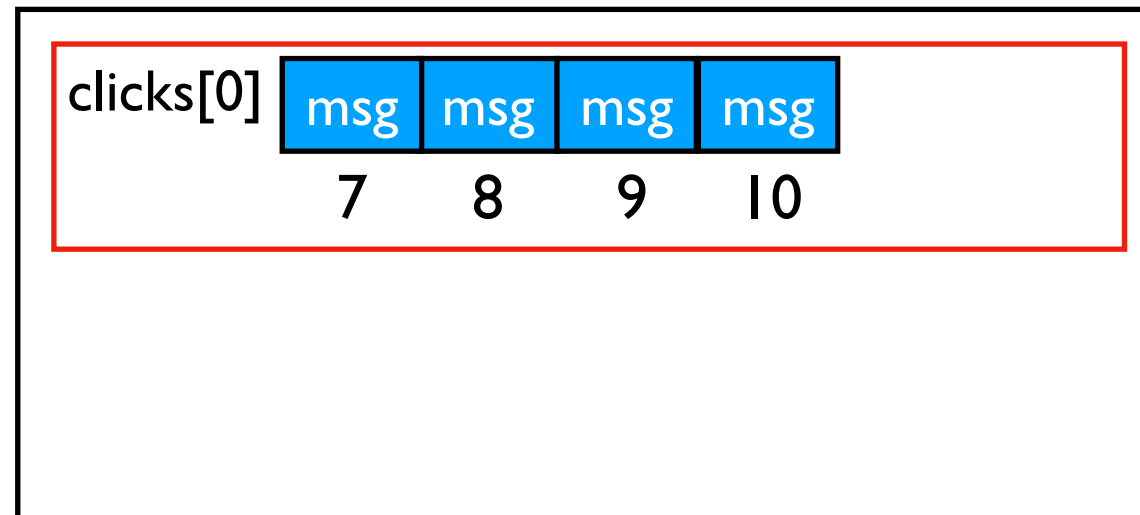


# Committed Messages

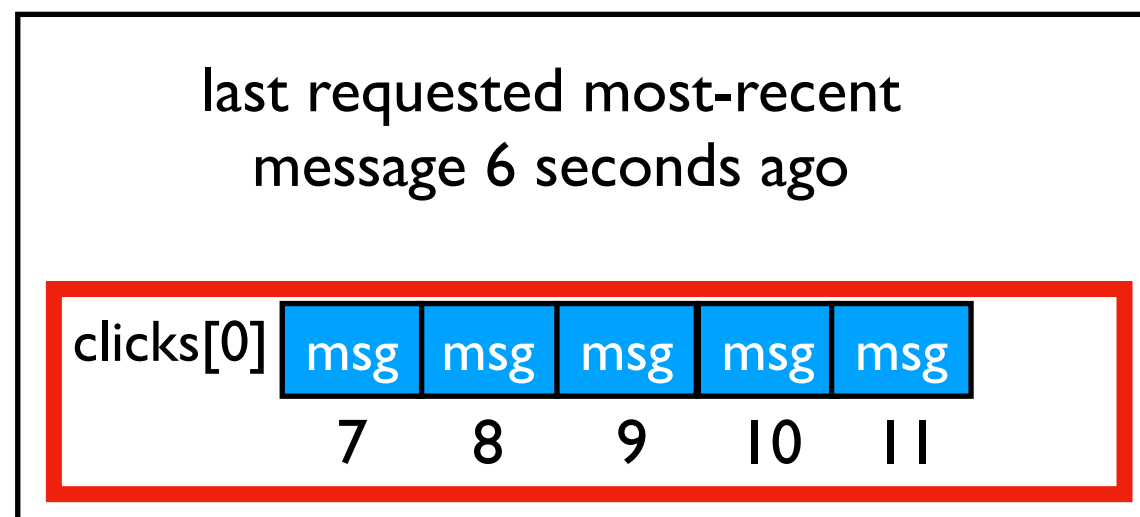
What is committed?

- assume  $RF=3$  and minimum in-sync=2
- is message 8 committed?
- message 10?
- message 11?

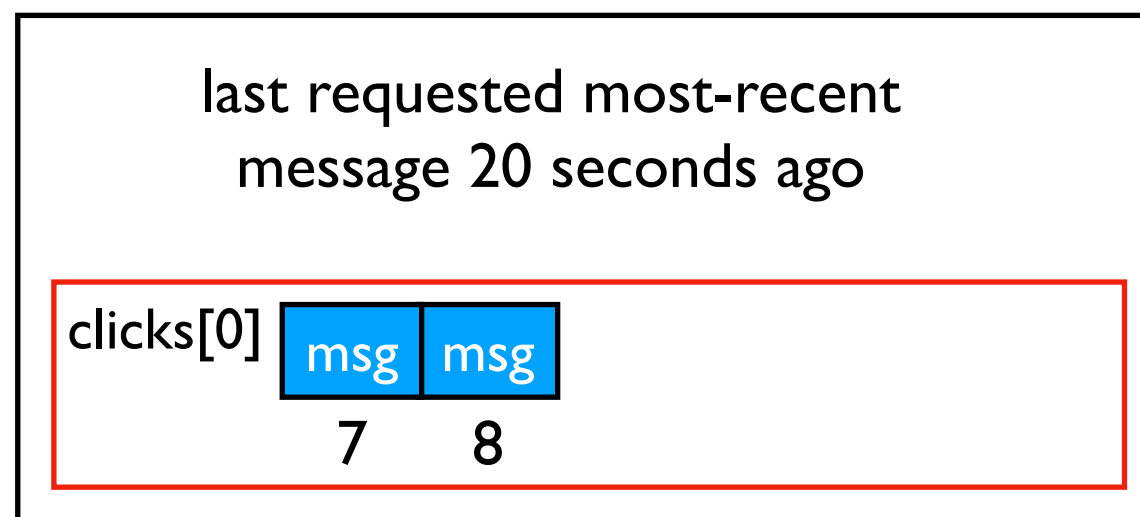
broker servers



follower replica  
(in-sync)



**leader replica**



follower replica  
(lagging behind)

## TopHat

# Working with Committed Data

How can we avoid "anomalies" (unexpected system behavior) by taking advantage of committed data?

# Example 1: Write Anomaly

## Scenario:

- producer writes a message
- produce receives an ACK (acknowledgement) from the broker
- consumers never see the message

**Cause:** maybe the leader sent an ACK back, then crashed, before replicating the message to the followers.

**How to avoid it?** *Use strong acks.*

## Consumer initialization:

- `KafkaProducer(..., acks=0)`  
**don't wait for leader to send back ACK**
- `KafkaProducer(..., acks=1)`  
**ACK after leader writes to its own log**
- `KafkaProducer(..., acks="all")`  
**ACK after data is committed (slowest but strongest)**

If you don't get an ACK that data is committed, usually best to retry in a loop (Kafka can be configured to do this for you).

# Example 2: Read Anomaly

## Scenario:

- a consumer reads a message
- there is an attempt to read the message again later (same consumer, or other)
- message is gone, or changed

**Cause:** maybe the message was consumed from the leader before it was replicated to the followers; then the leader crashed and the new leader doesn't have that message for future consumption.

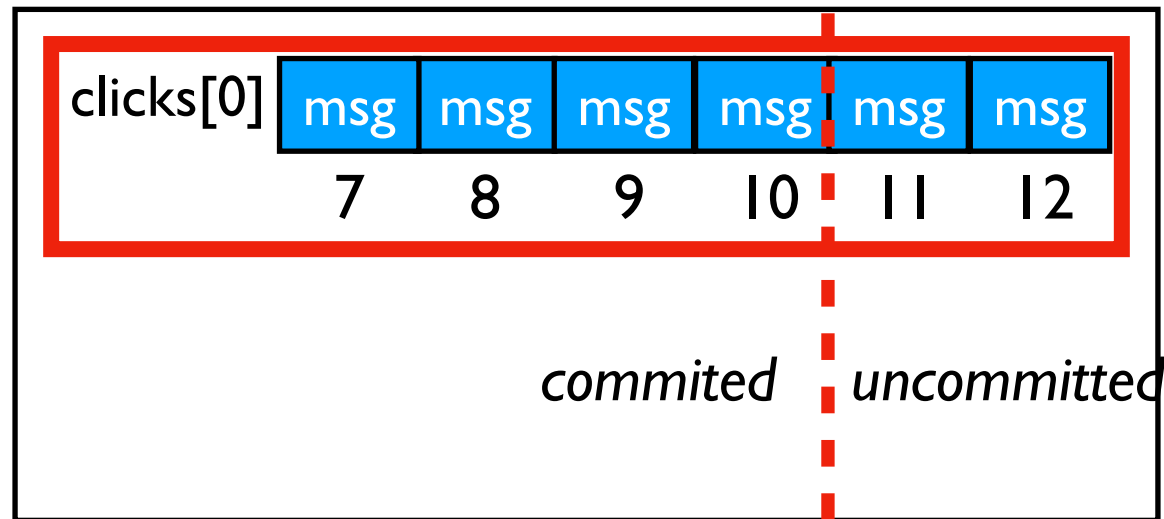
**How to avoid it?** *Never read un-committed data.*

The leader does this automatically.

# Fetch Behavior: Consumer vs. Follower

broker servers

leader replica

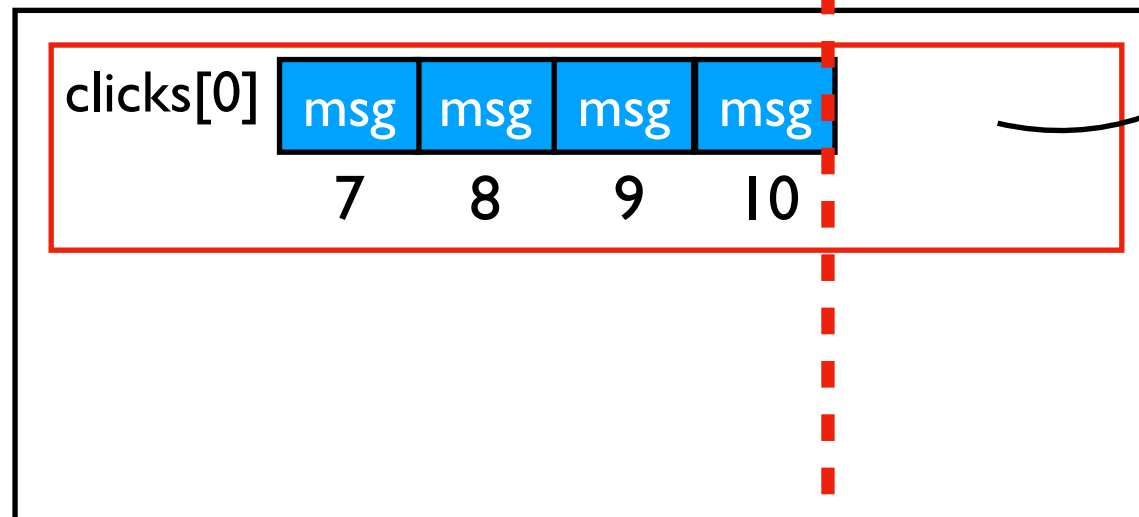


fetch

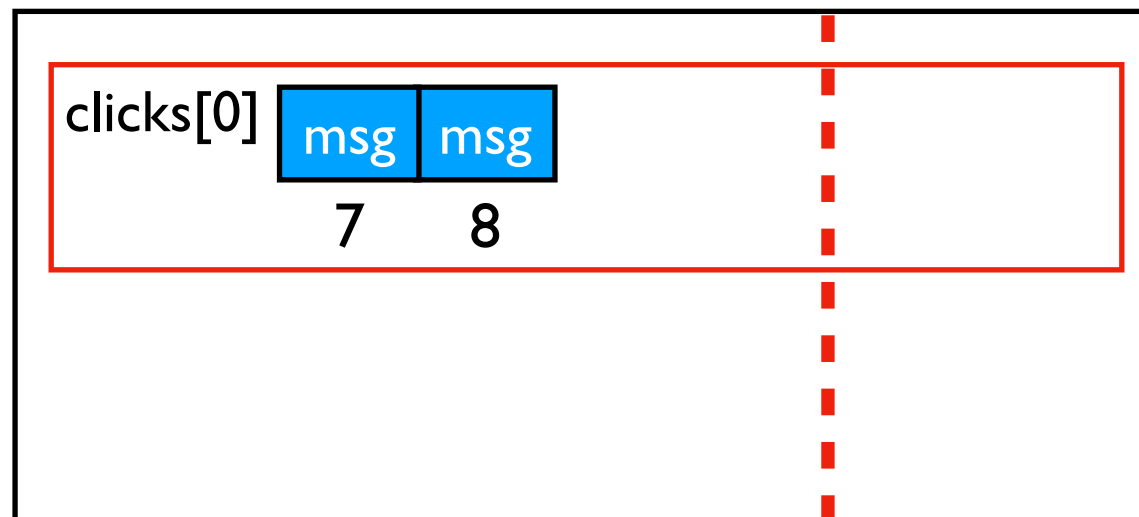
consumer

fetch

follower replica  
(in-sync)



follower replica  
(lagging behind)



- **consumer fetch:** leader **WILL NOT** send messages until it knows they are committed
- **follower fetch:** leader **WILL** send uncommitted messages

# Outline: Kafka Reliability

Kafka Replication

Fault Tolerance

Exactly-Once Semantics

# Semantics (Meaning)

## Dictionary

Definitions from [Oxford Languages](#) · [Learn more](#)



se·man·tics

*noun*

noun: **semantics**; noun: **logical semantics**; noun: **lexical semantics**

the branch of linguistics and logic concerned with meaning. There are a number of branches and

Programming Example:

- **Runtime bug:** the program crashed, there was clearly a problem
- **Semantic bug:** you need to understand the **meaning** of the results to say whether or not the program behaved correctly

In Systems:

- what does it **mean** when we get we get an **ACK**, or a **write returns**?
- the meaning depends on how we configured things...

# *At-most-once* semantics

```
producer = KafkaProducer(..., acks=1)
producer.send("my-topic", b"some-value")
```

With acks as 0 or 1 and no retry, a successful write means the data was recorded at most once (ideally once, but if the leader crashes at a bad time, maybe zero times).

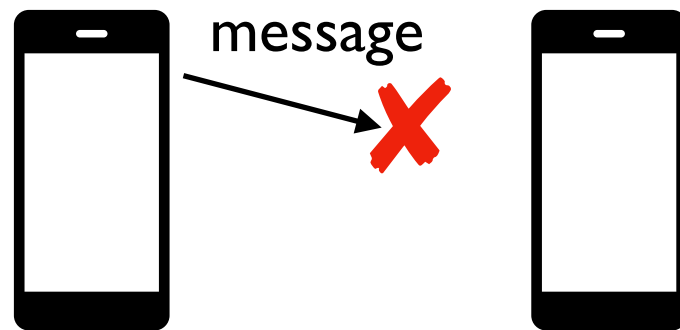


# Using strong ACKs and retry

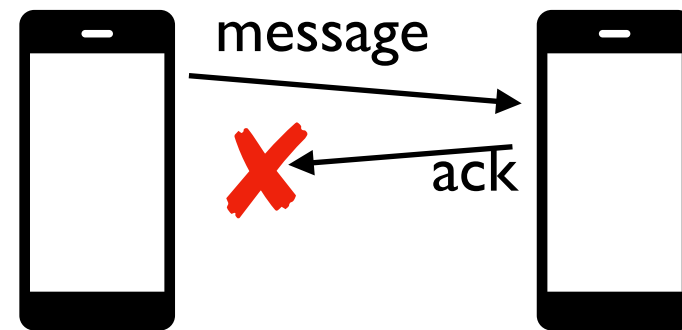
```
producer = KafkaProducer(..., acks="all", retries=10)
producer.send("my-topic", b"some-value")
```

Keep retrying until success (within reason -- for example, 10 times)

**Problem:** there are two reasons we might not get an ACK:



scenario 1



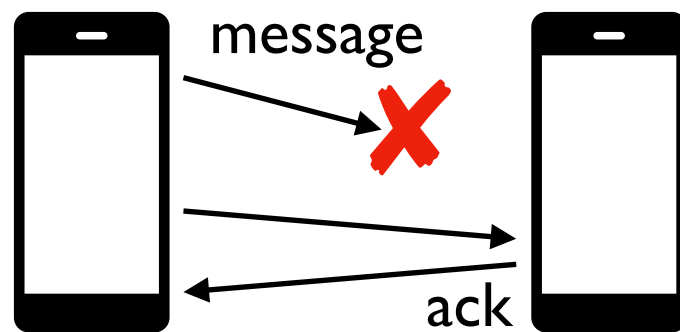
scenario 2

# Using strong ACKs and retry

```
producer = KafkaProducer(..., acks="all", retries=10)
producer.send("my-topic", b"some-value")
```

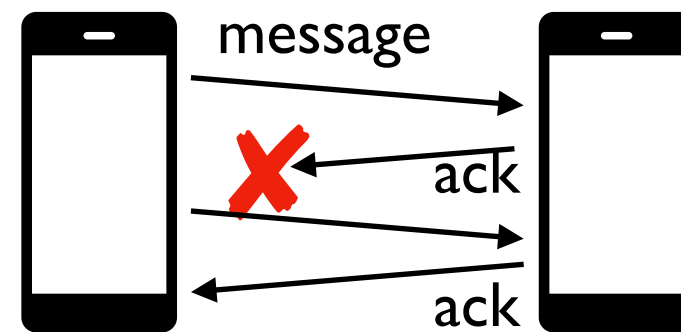
Keep retrying until success (within reason -- for example, 10 times)

**Problem:** there are two reasons we might not get an ACK:



scenario 1

*message written once*



scenario 2

*message written twice*

A strong ACK with retry provides **at-least-once semantics** because we're guaranteed 1+ messages upon success

# Are duplicate messages OK?

Yes, if they're **idempotent**.

"An operation is called **idempotent** when the effect of performing the operation multiple times is equivalent to the effect of performing the operation a single time"  
~ *Operating Systems: Three Easy Pieces*, by Arpaci-Dusseau

```
x = 0  
y = 0
```

```
def set_x(value):  
    global x  
    x = value
```

```
def inc_y(value):  
    global y  
    y += value
```

```
# if we just do once, is it the same?  
set_x(123)  
set_x(123)  
set_x(123)  
  
# if we just do once, is it the same?  
inc_y(3)  
inc_y(3)  
inc_y(3)
```

## TopHat

# Supressing Duplicates

With some cleverness, we can make ANYTHING idempotent.

```
y = 0
completed_ops = set()

def inc_y(value, operation_id):
    global y
    if not operation_id in completed_ops:
        y += value
        completed_ops.add(operation_id)

inc_y(3, 1251253)
inc_y(3, 1251253)      # no effect
inc_y(3, 1251253)      # no effect

inc_y(3, 9876)
inc_y(3, 9876)          # no effect

inc_y(1, 5454)
```

# Exactly-Once Semantics: Producer Side

Upon a successful write, the message will be considered **exactly once** (duplicates will be suppressed by brokers or consumers).

## Producer settings:

- `acks="all"`
- `retry=N`
- `enable.idempotence=True`

With idempotence enabled, producers automatically generate unique operation IDs and brokers suppress duplicates (this has an extra cost).

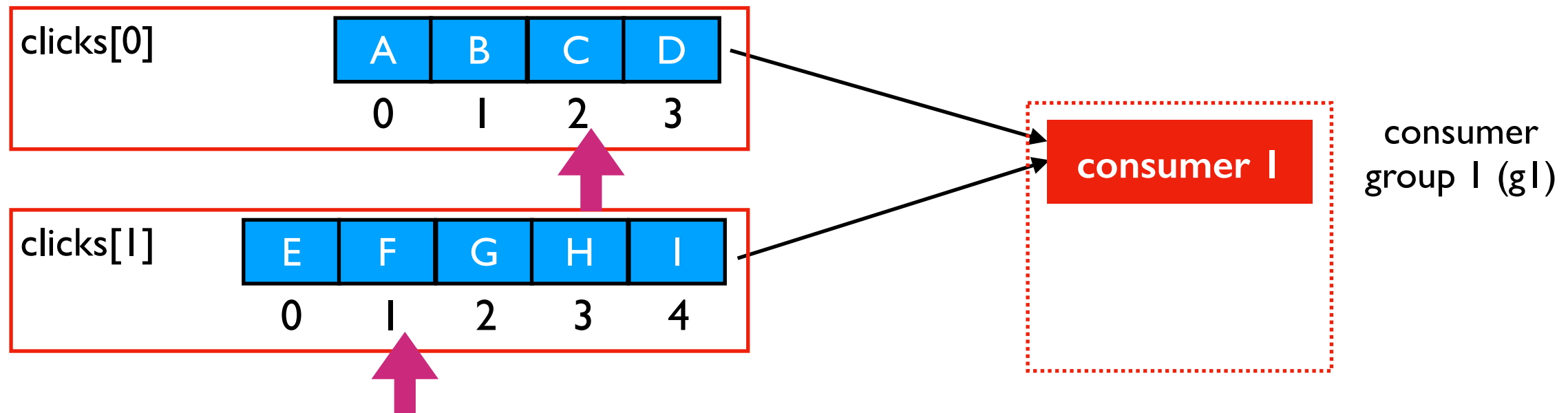
You can use `enable.idempotence` in Java, but the `kafka-python` package doesn't support it.:

- need to handle it yourself
- often, messages have a unique ID anyway, so consumers can ignore dups
- Example: weather stations that emit one record per day -- if a consumer sees a date for a station it has seen before, ignore it

# Exactly-Once Semantics: Consumer Side

```
c = KafkaConsumer("clicks",  
                  group_id="g1",  
                  ...)  
  
while True:  
    batch = c.poll(1000)  
    ...
```

## Topic Partitions



Suppose consumer dies and is replaced by another in the same group

- don't want replacement to miss any messages
- don't want replacement to repeat any processing

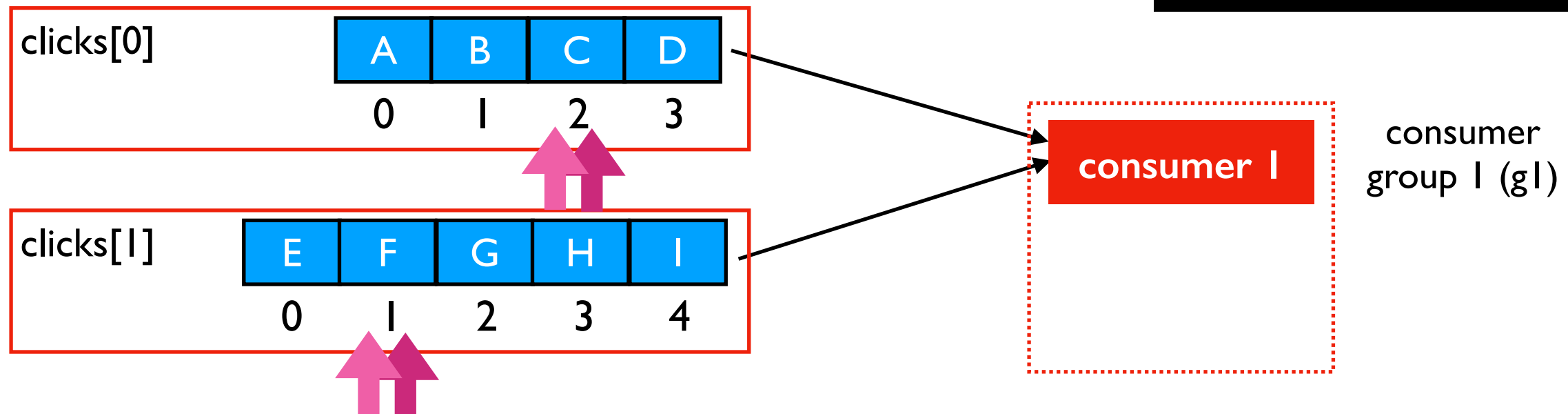
	g1 offsets
clicks[0]	2
clicks[1]	1

# Exactly-Once Semantics: Consumer Side

```
c = KafkaConsumer("clicks",  
                  group_id="g1",  
                  enable_auto_commit=True,  
                  auto_commit_interval_ms=5000,  
                  ...)  
  
while True:  
    batch = c.poll(1000)  
    ...
```

**Note! Committing messages and committing read offsets are two different ideas.**

## Topic Partitions



	g1 offsets
clicks[0]	2
clicks[1]	1

*Kafka*



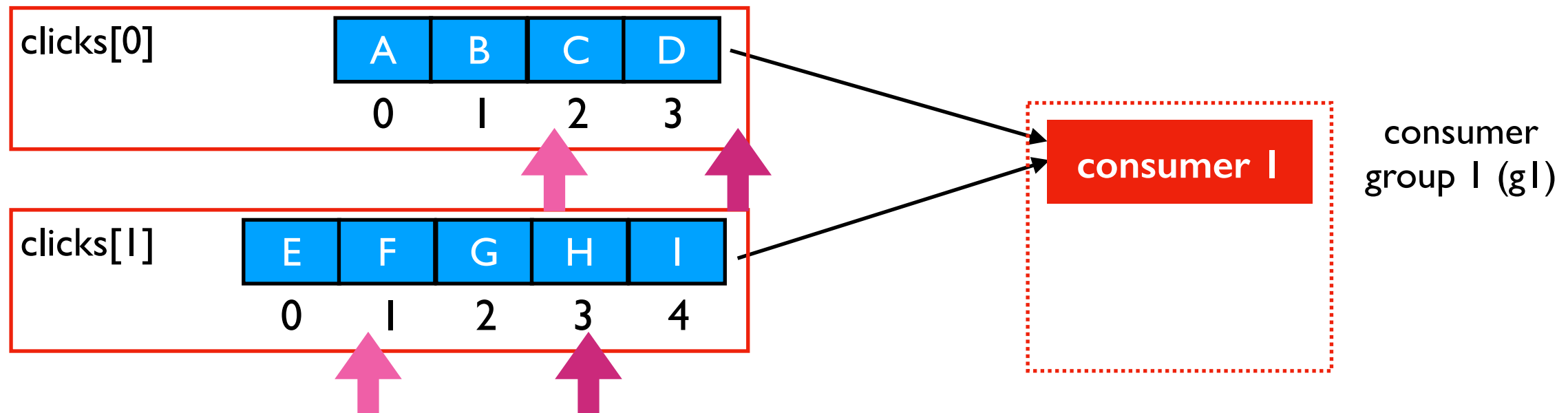
	g1 offsets
clicks[0]	2
clicks[1]	1

*consumer*

# Exactly-Once Semantics: Consumer Side

```
c = KafkaConsumer("clicks",  
                  group_id="g1",  
                  enable_auto_commit=True,  
                  auto_commit_interval_ms=5000,  
                  ...)  
  
while True:  
    batch = c.poll(1000)  
    ...
```

## Topic Partitions



	g1 offsets
clicks[0]	2
clicks[1]	1

*Kafka*

If we crash at a bad time, the offsets the next consumer gets from Kafka will only be approximately correct.

	g1 offsets
clicks[0]	4
clicks[1]	3

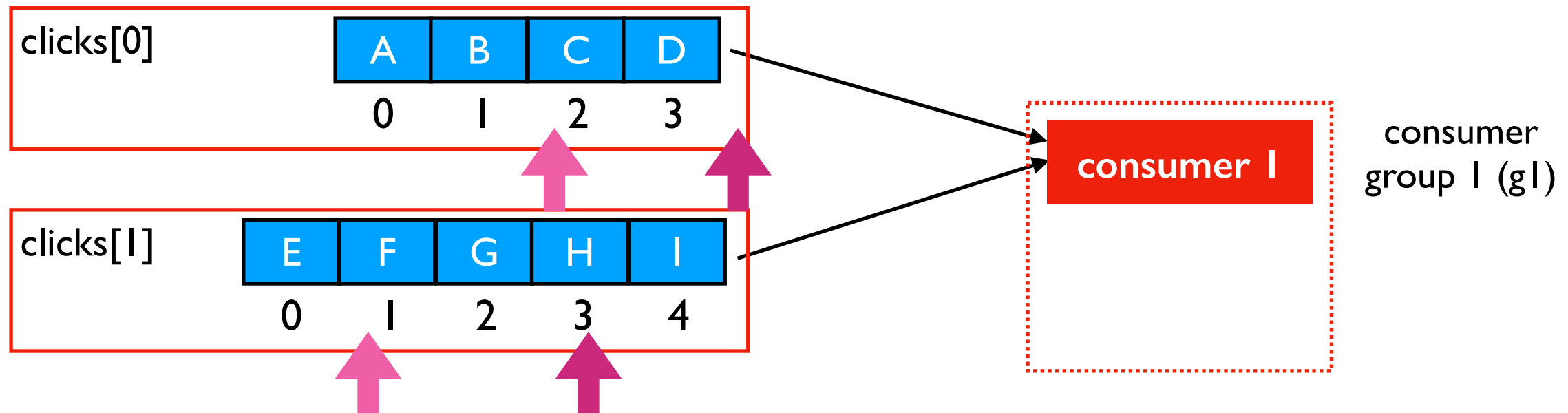
*consumer*



# Approach 1: Manually Commit Offsets

```
c = KafkaConsumer("clicks",  
                  group_id="g1",  
                  enable_auto_commit=False,  
                  ...)  
  
while True:  
    batch = c.poll(1000)  
    ...  
    c.commit() # manually commit read offsets
```

## Topic Partitions



	g1 offsets
clicks[0]	2
clicks[1]	1

*Kafka*

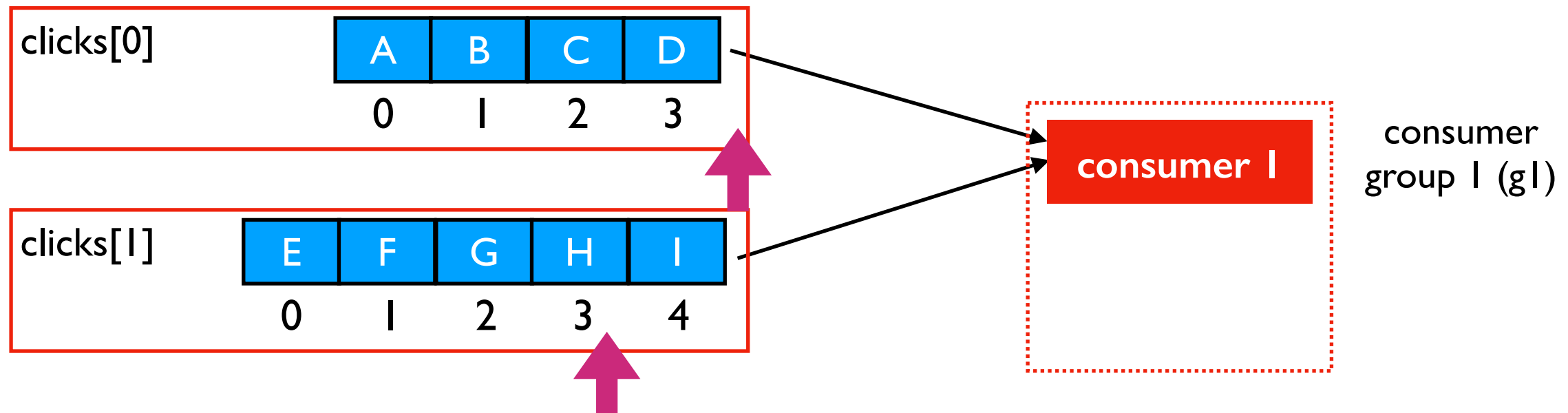
	g1 offsets
clicks[0]	4
clicks[1]	3

*consumer*

# Approach 2: Externally Save Commits

```
c = KafkaConsumer("clicks",
                  group_id="g1",
                  ...)
# TODO: seek to previous position
while True:
    batch = c.poll(1000)
    ...
    # TODO: write offsets to a DB or file
```

## Topic Partitions



	g1 offsets
clicks[0]	4
clicks[1]	3

*consumer*

# Conclusion

Every part of the system has a part to play in **reliability** and **exactly-once semantics**.

## Producer:

- requesting strong acks
- retry
- idempotence

## Broker:

- replicating data to followers
- failing over to new leader
- sending acks
- helping producer suppress duplicates
- keeping uncommitted data hidden from consumers

## Consumer:

- carefully handling read offsets
- sometimes suppressing duplicates (if not handled by producers+brokers)