Making sense of exactly-once semantics

Flavio Junqueira

Real-time streams powered by Apache Kafka.

About me

- **Confluent**
  - ✓ Infrastructure Engineer
  - ✓ Kafka Core
- Apache Software Foundation (**ASF**)
  - ✓ Apache ZooKeeper, BookKeeper, Kafka
  - ✓ Apache Incubator
- Previously
  - ✓ **Yahoo! Research** and **Microsoft Research**

Why this talk…

Apache Flink

The problem of providing **exactly once guarantees** really boils down to determining what state the streaming computation currently is in … rewinding the stream source (for example with help of **Apache Kafka**) to the point when the snapshot was taken and hitting the play button again.

http://data-artisans.com/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink/

Apache Spark

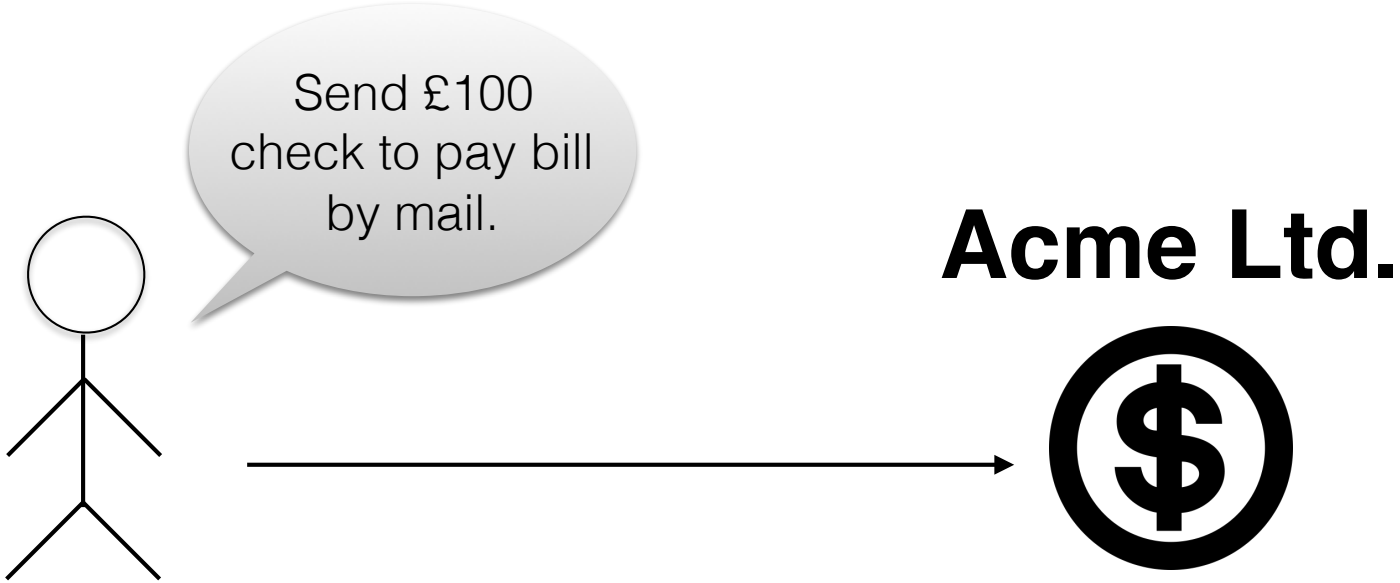
Exactly-once semantics: … we use **simple Kafka API** … each record is received by Spark Streaming effectively exactly once despite failures.

http://spark.apache.org/docs/latest/streaming-kafka-integration.html

Apache Kafka guarantees **at least once**

Towards exactly-once in Apache Kafka

Context


Send £100 check to pay bill by mail.
Acme Ltd.

The Daily L

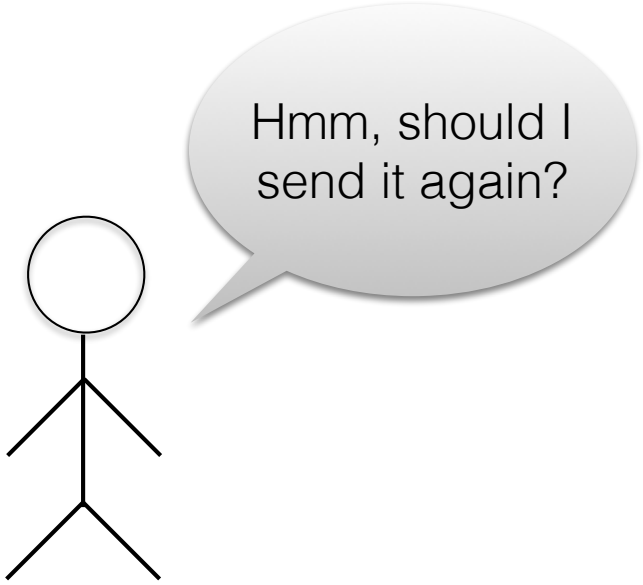
Sunday, August 30, 2012

Post Office on Fire

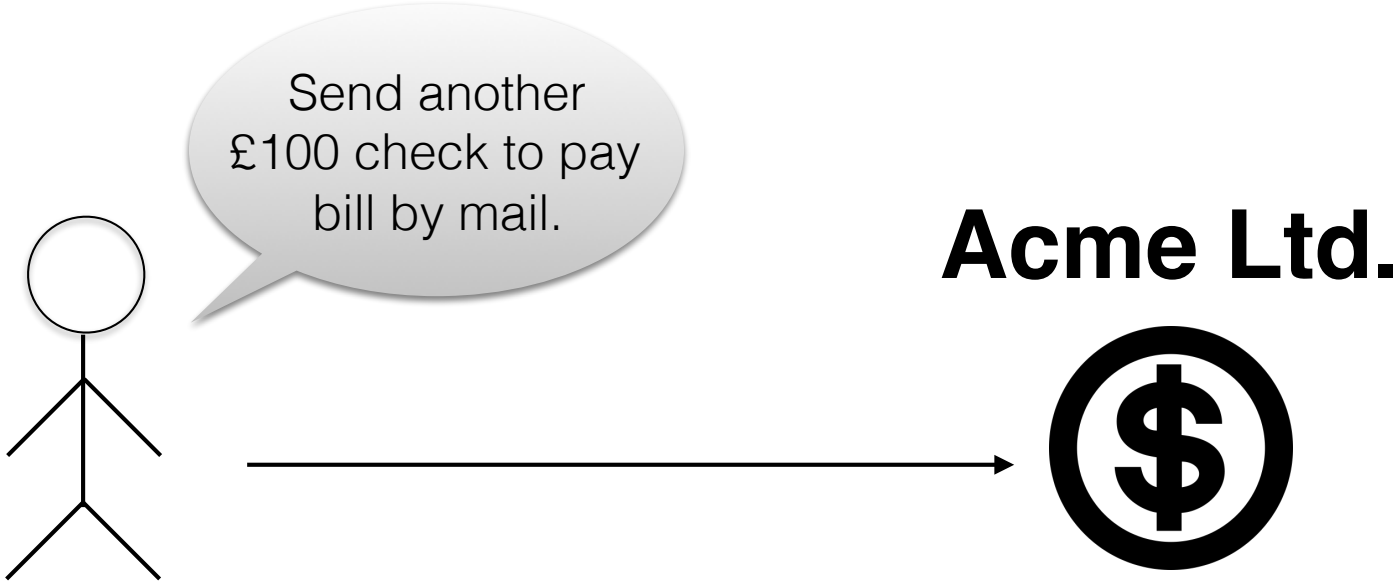
The main post office was set on fire yesterday by an unhappy customer who hasn't had his letters delivered correctly for the past 5 years. Witnesses said that the customer has started the fire while shouting "No more paper oppression!". The customer escaped and was identified by multiple other customers as a Strata London speaker. Unfortunately, multiple letters and parcels have been completely destroyed. The postal service apologizes for the inconvenience, but asks its customers to resend when possible.

Rer
foll
imp

The
that
rela
the
beh
of a
exp



Acme Ltd.


Send another £100 check to pay bill by mail.
Acme Ltd.

- Expect to pay only 100 and not 200
- Check name or customer id

Make actions idempotent

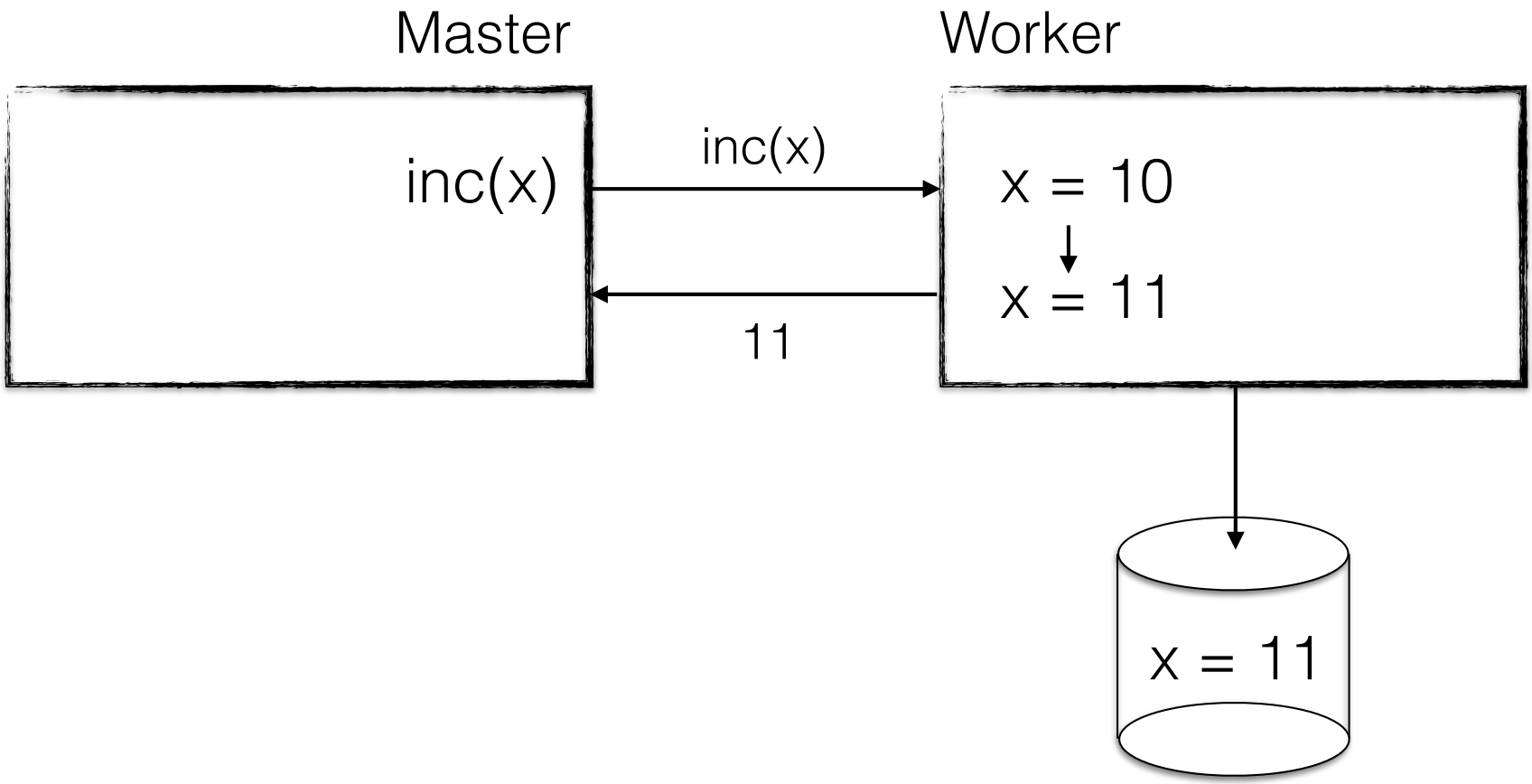
Remote Reference/Remote Operation Model

Table I. Reliability Semantics Survey.

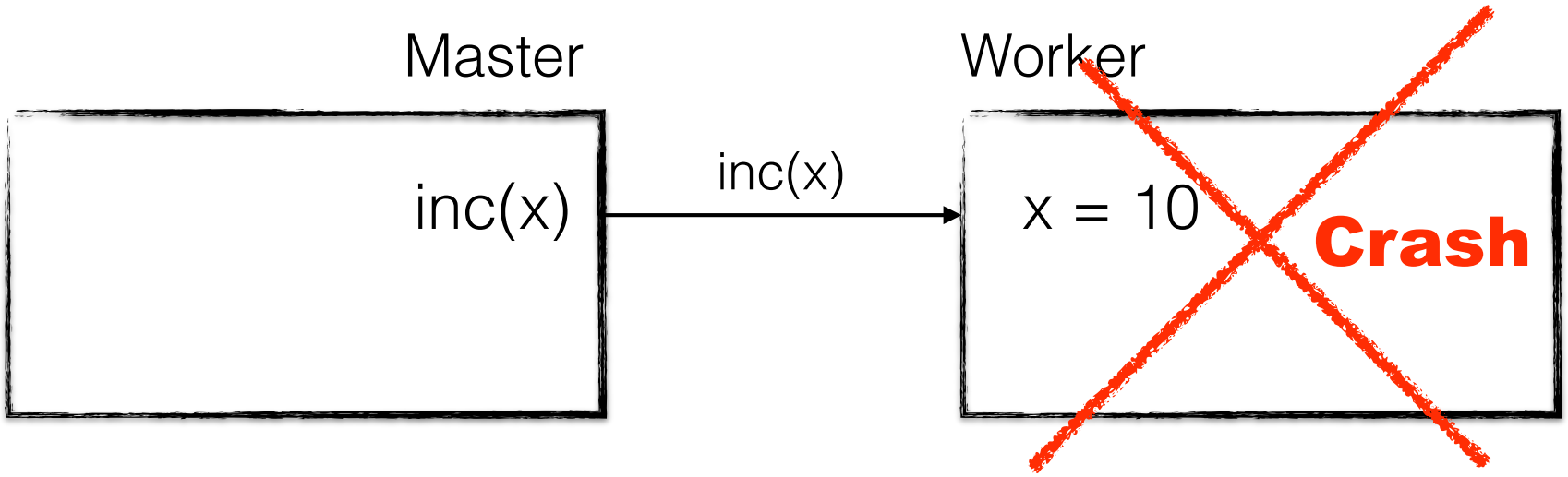
Protocol Class	Reference Semantics Under Different Failure Conditions				
	No Failures	Lost Packets	Lost Packets & Slave Failure	Lost Packets & Master Failure	Lost Packets, Master & Slave Failure
Maybe	op performed: 1 result-commit: 1	op performed: 0,1 result-commit: 0,1	op performed: 0,1 result-commit: 0,1	op performed: 0,1 result-commit: 0	op performed: 0,1 result-commit: 0
At-Least-Once	op performed: 1 result-commit: 1	op performed: ≧ 1 result-commit: ≧ 1	op performed: ≧ 0 result-commit: ≧ 0	op performed: ≧ 0 result-commit: 0	op performed: ≧ 0 result-commit: 0
Only-Once-Type-1	op performed: 1 result-commit: 1	op performed: 1 result-commit: 1	op performed: 0,1 result-commit: 0,1	op performed: 0,1 result-commit: 0	op performed: 0,1 result-commit: 0
Only-Once-Type-2	op performed: 1 result-commit: 1	op performed: 1 result-commit:1	op performed: 1 result commit:1	regular master process op performed: 1 result commit: 0 recoverable master process op performed: 1 result-commit: 1	regular master process op performed: 1 result-commit: 0 recoverable master process op performed: 1 result-commit: 1

A. Spector, *Performing Remote Operations Efficiently on a Local Computer Network*, CACM, April 1982

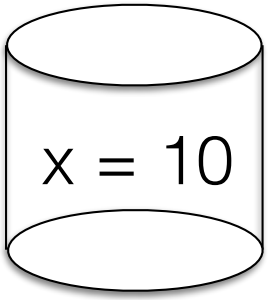
Remote Reference/Remote Operation Model



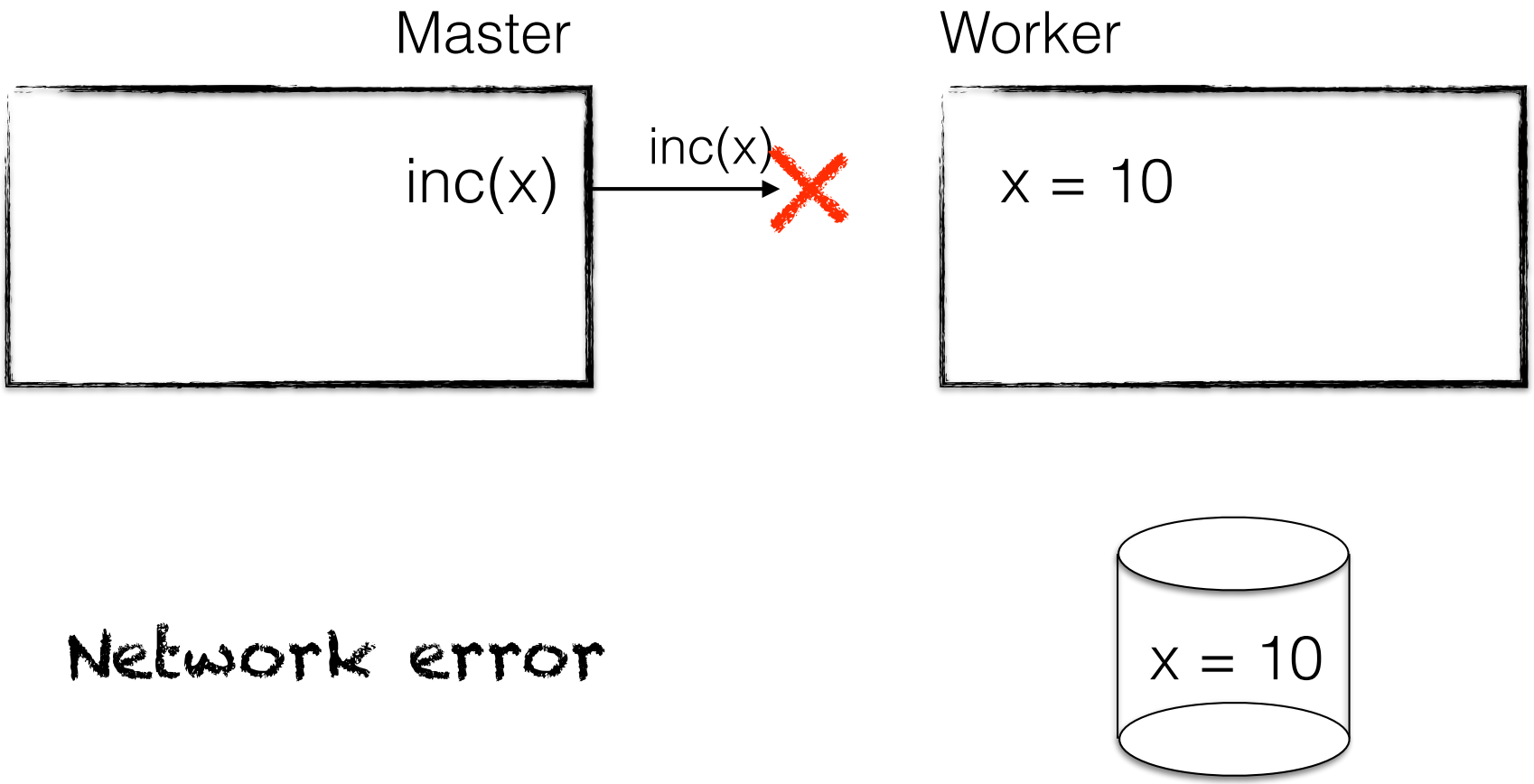
Remote Reference/Remote Operation Model



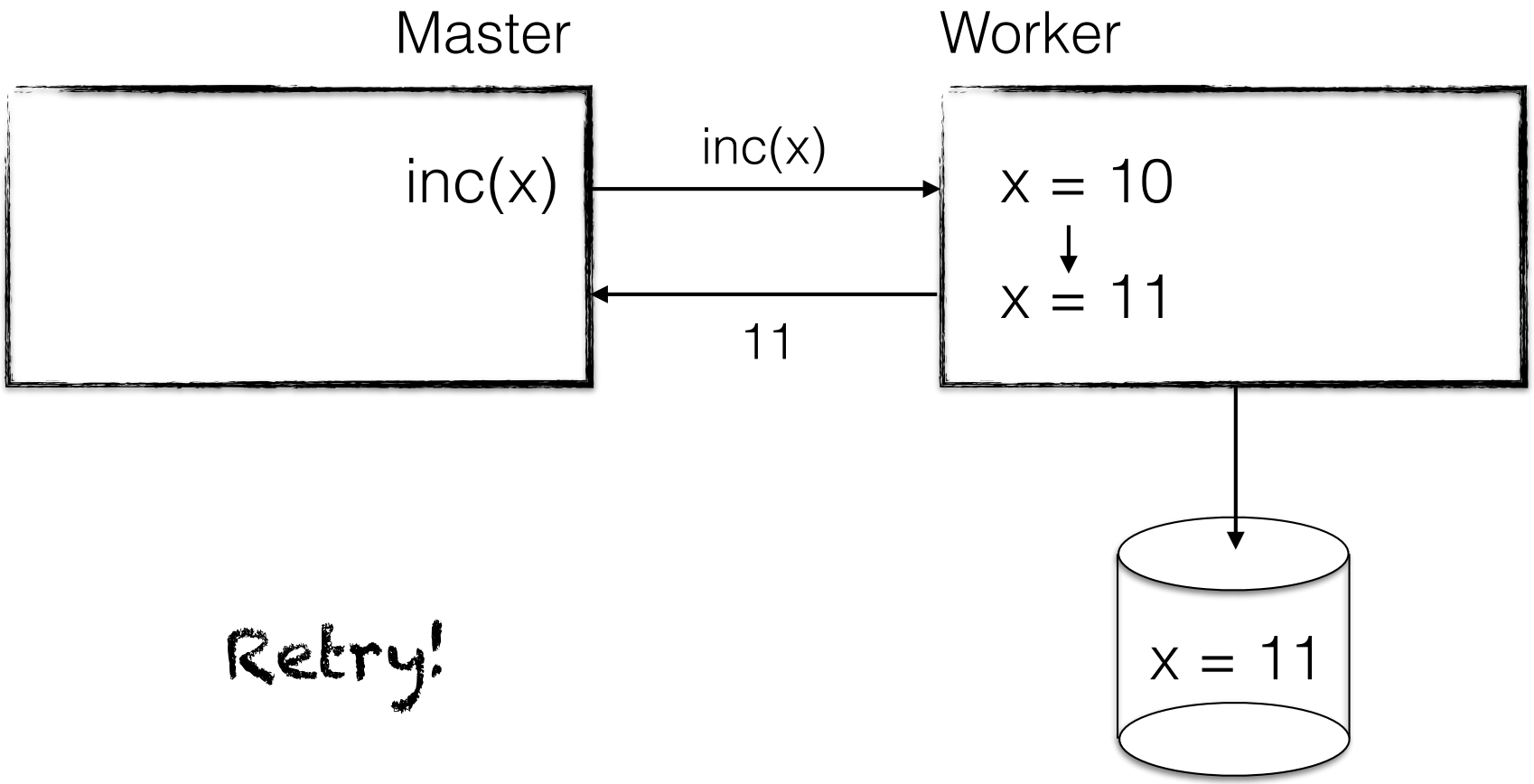
Worker has crashed



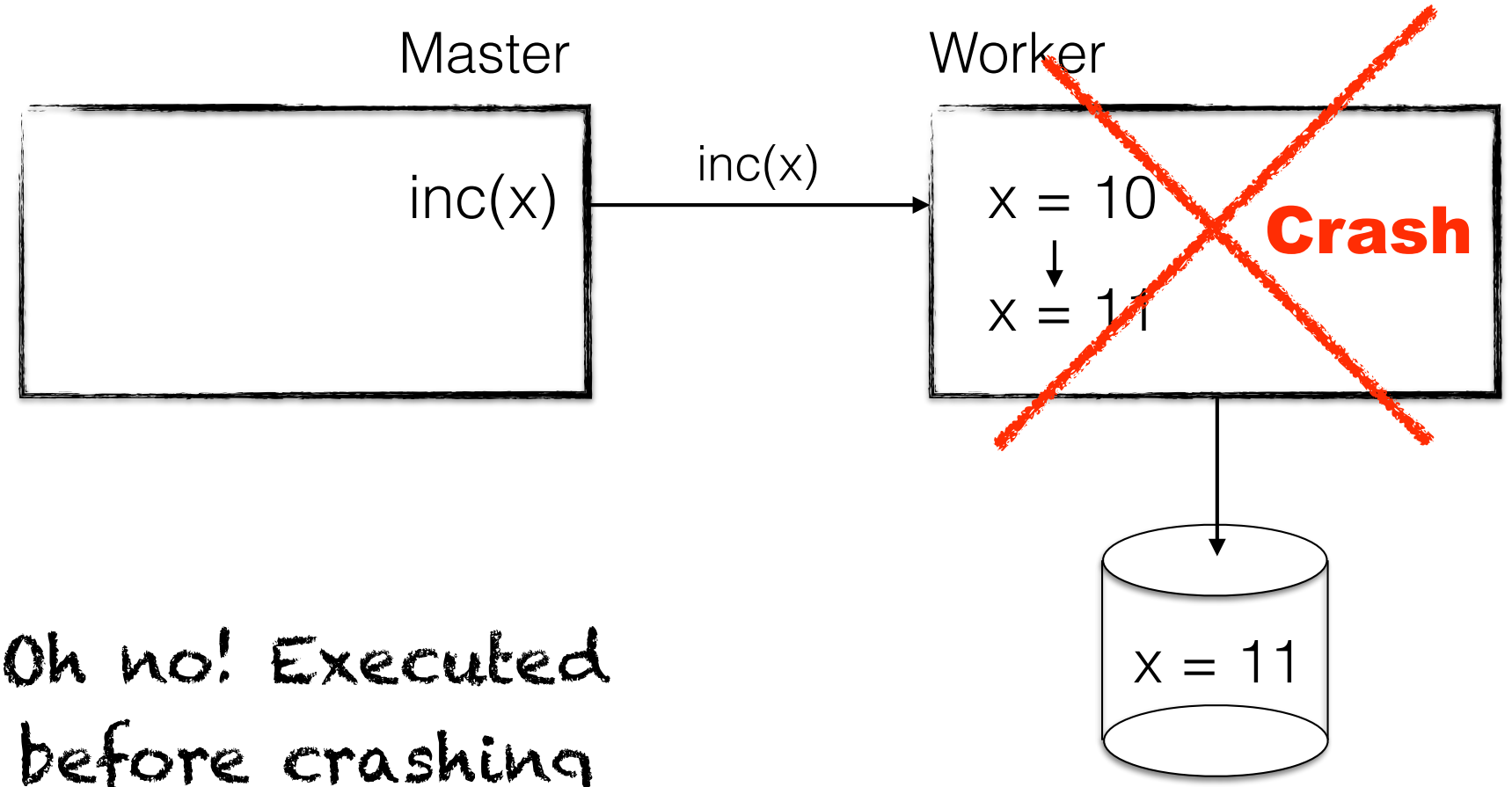
Remote Reference/Remote Operation Model



Remote Reference/Remote Operation Model

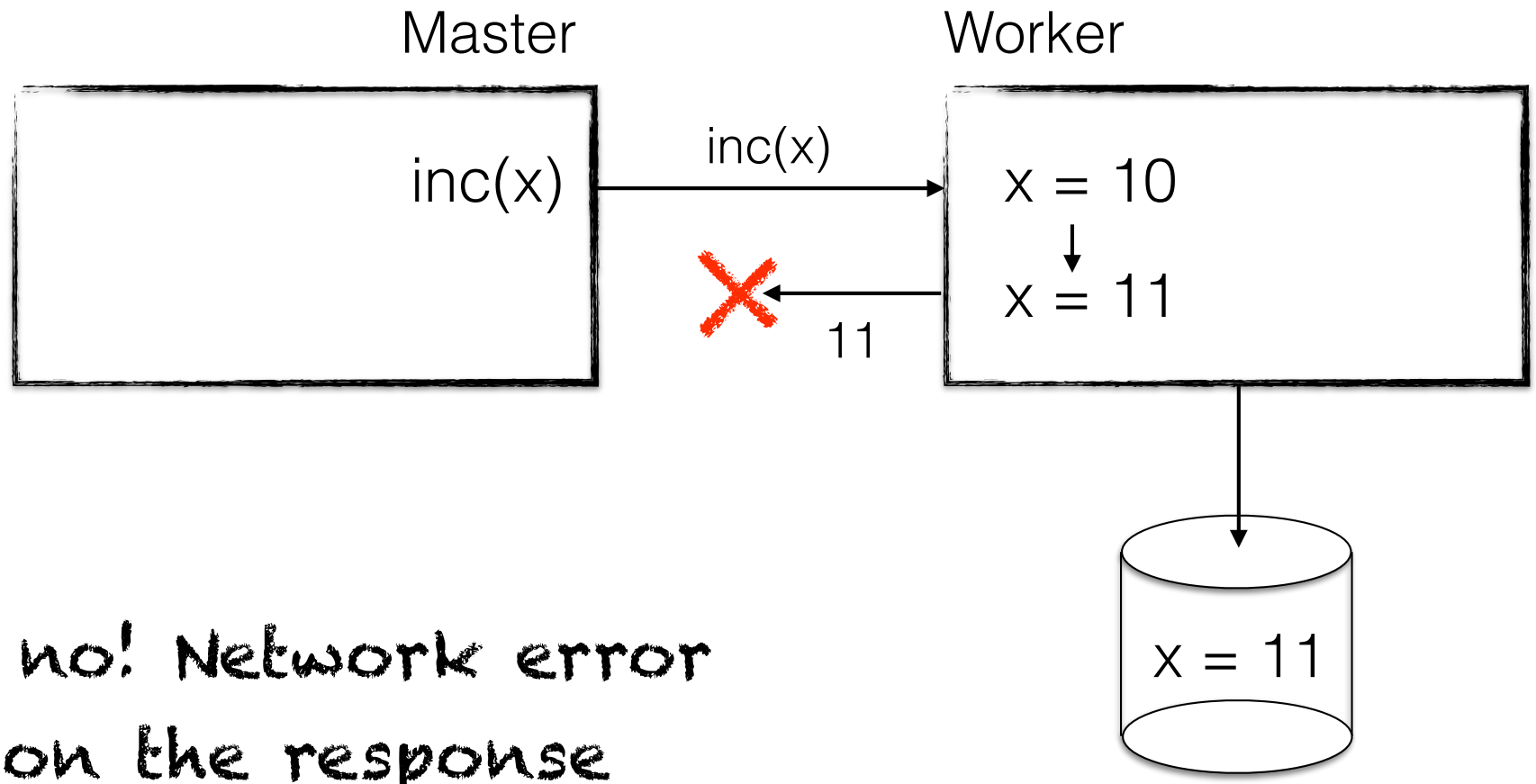


Remote Reference/Remote Operation Model



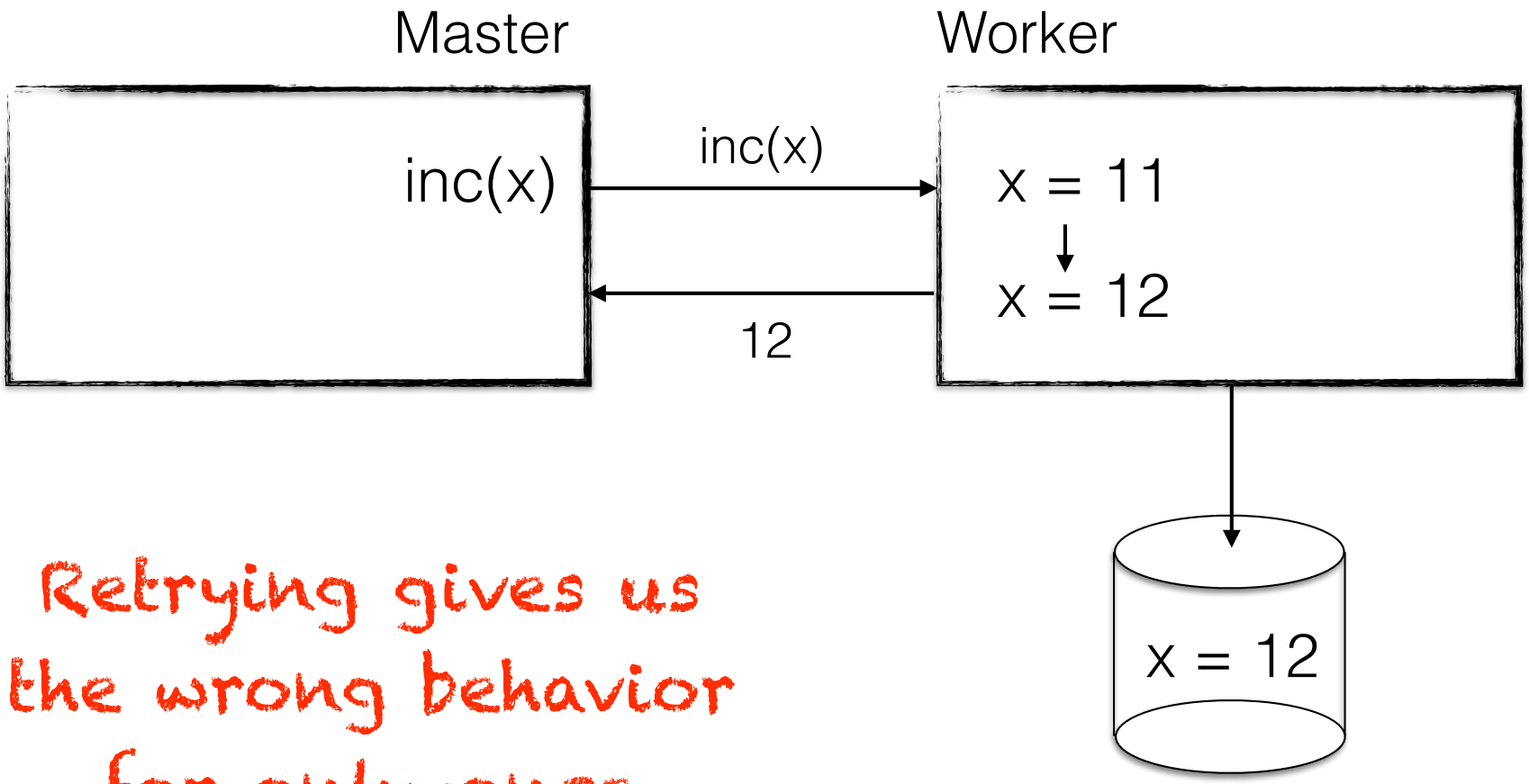
Oh no! Executed before crashing

Remote Reference/Remote Operation Model



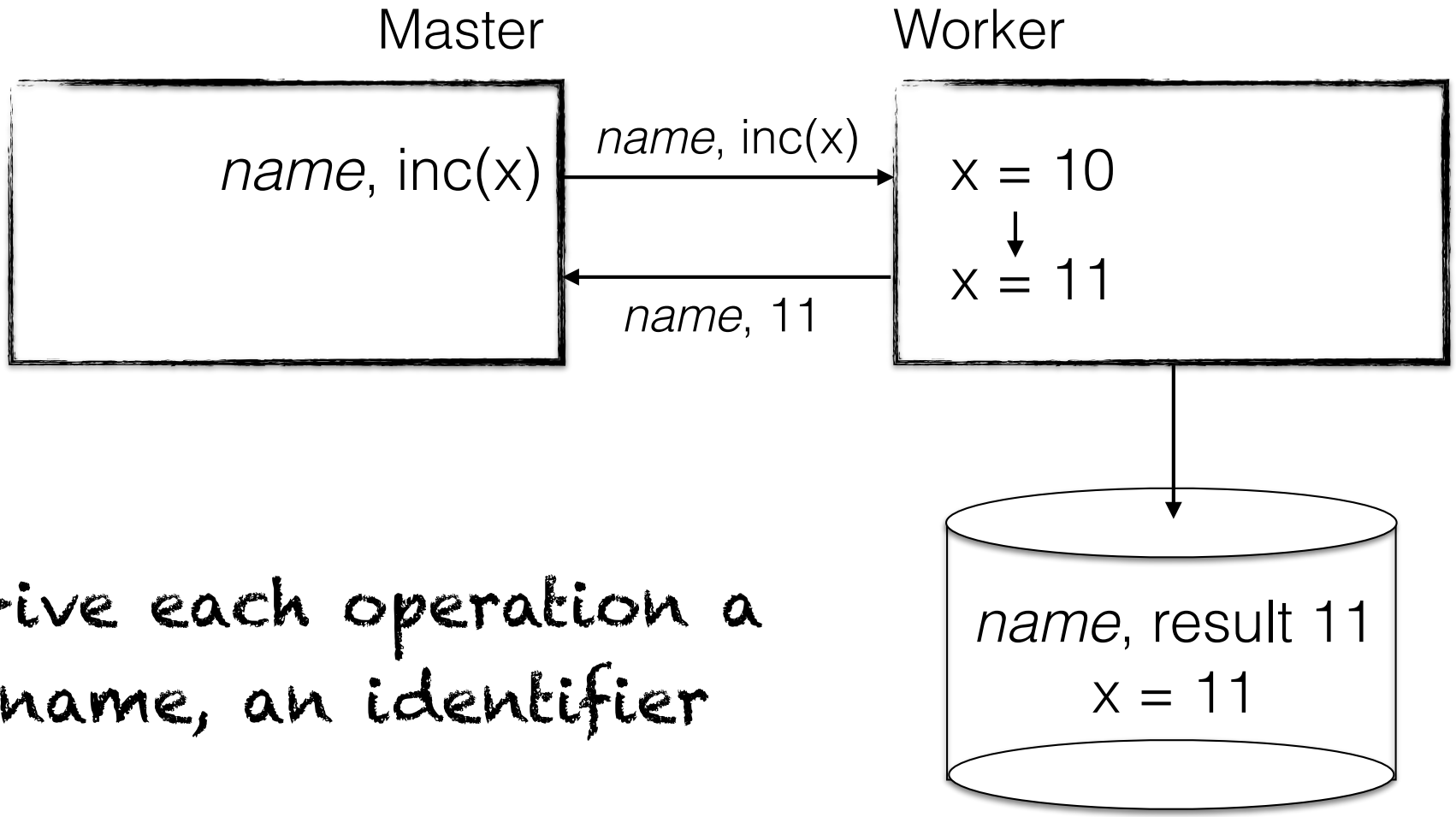
Oh no! Network error on the response

Remote Reference/Remote Operation Model



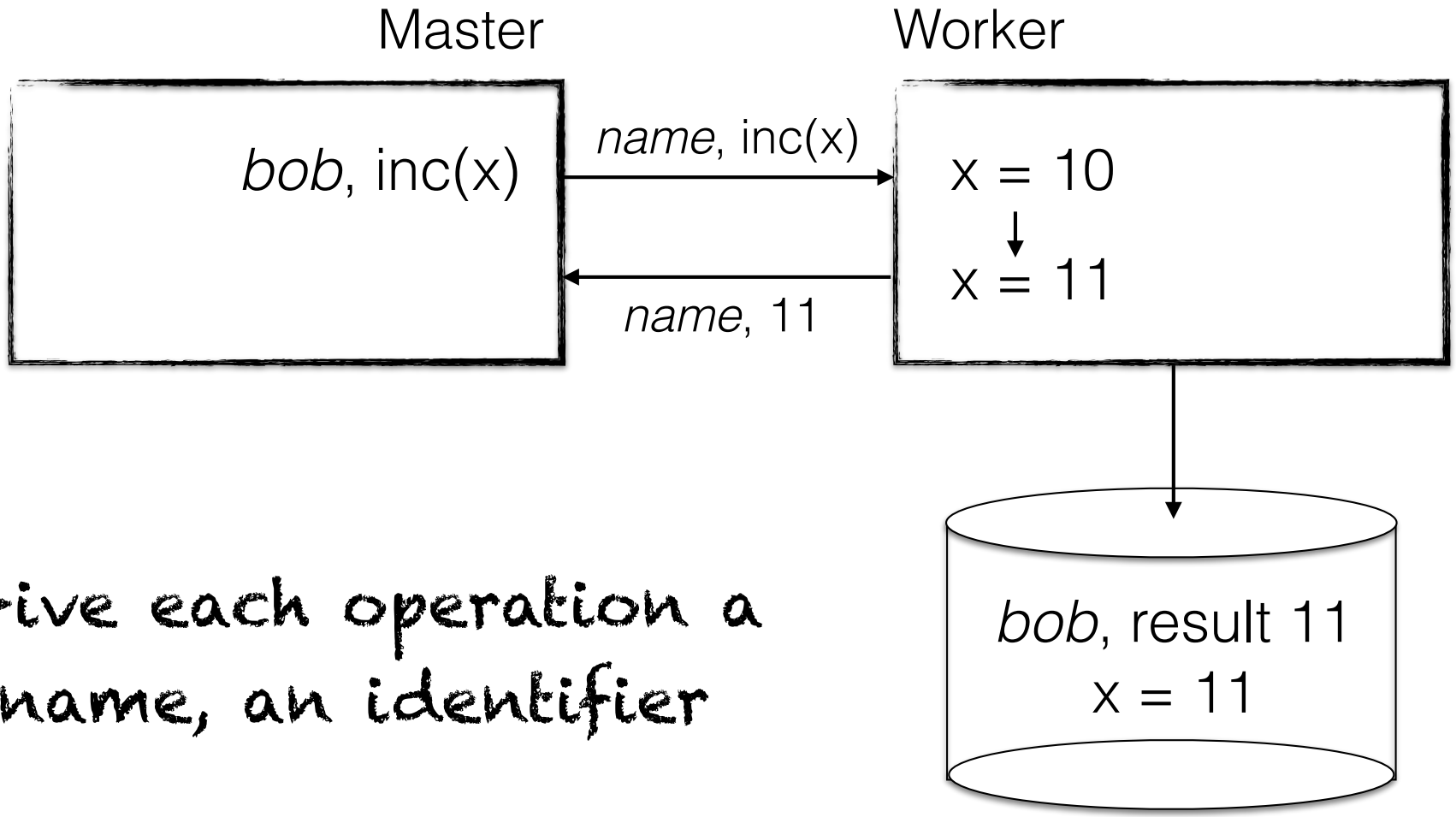
Retrying gives us the wrong behavior for only-once

Remote Reference/Remote Operation Model



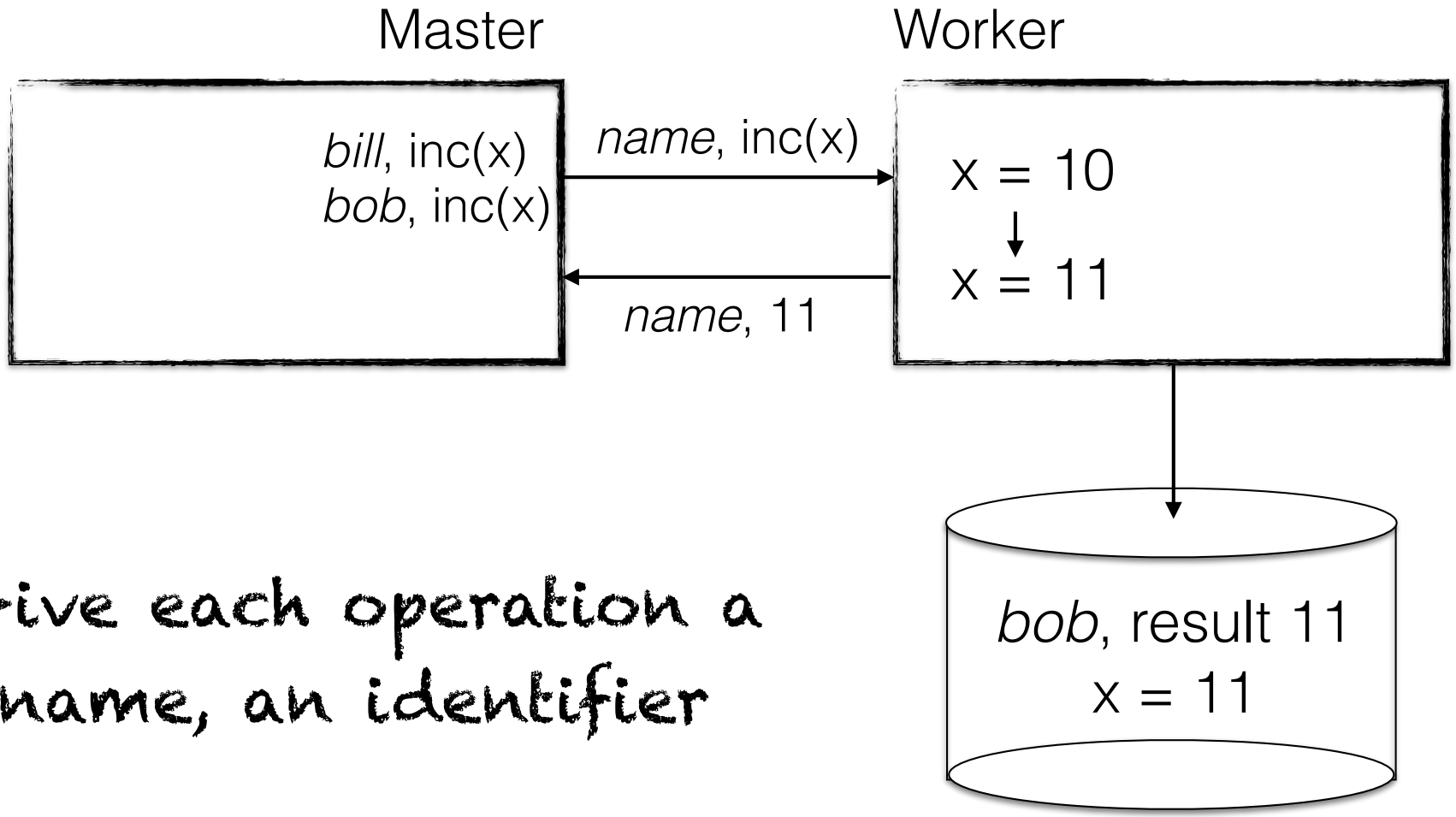
Give each operation a name, an identifier

Remote Reference/Remote Operation Model



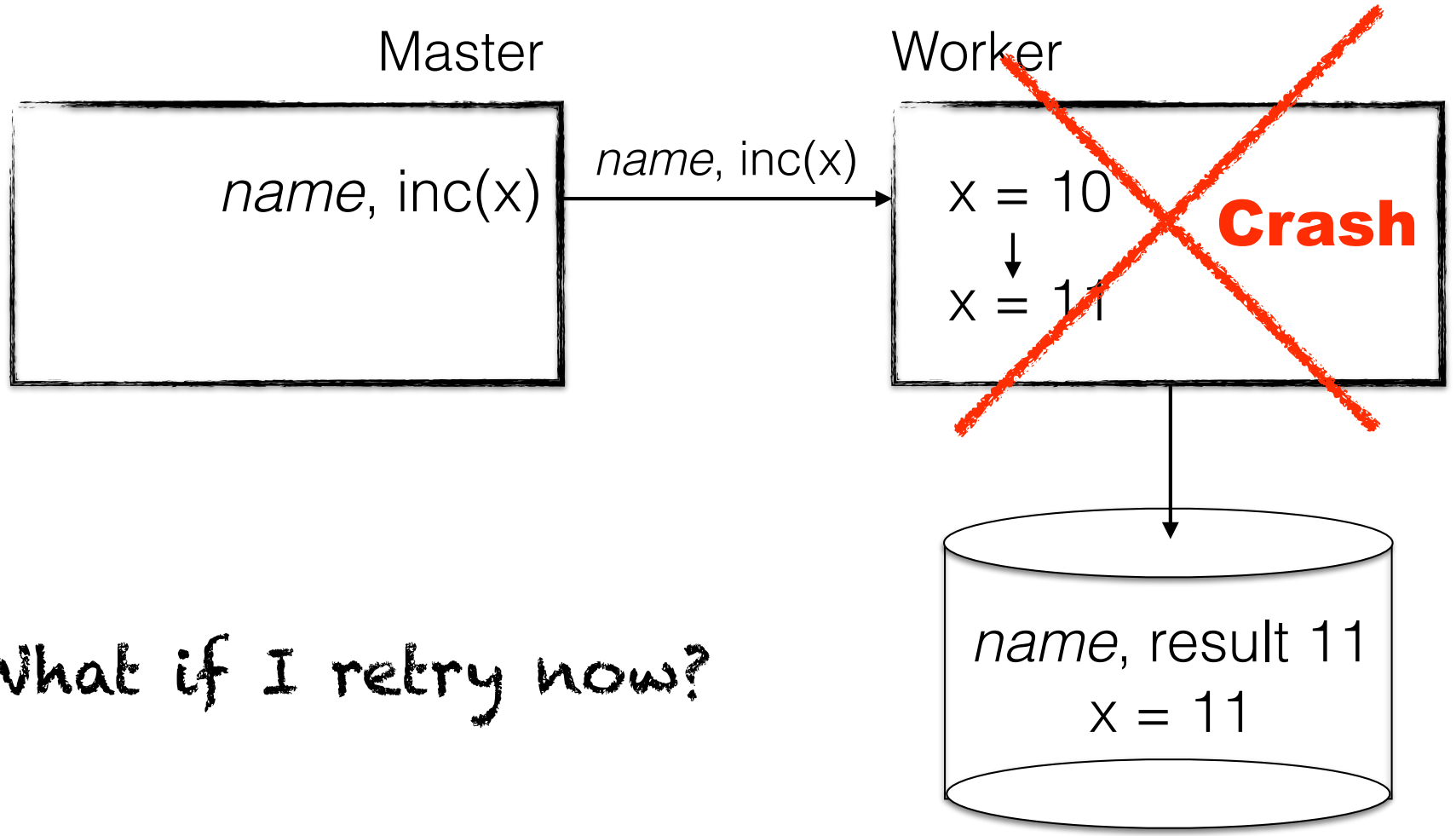
Give each operation a name, an identifier

Remote Reference/Remote Operation Model



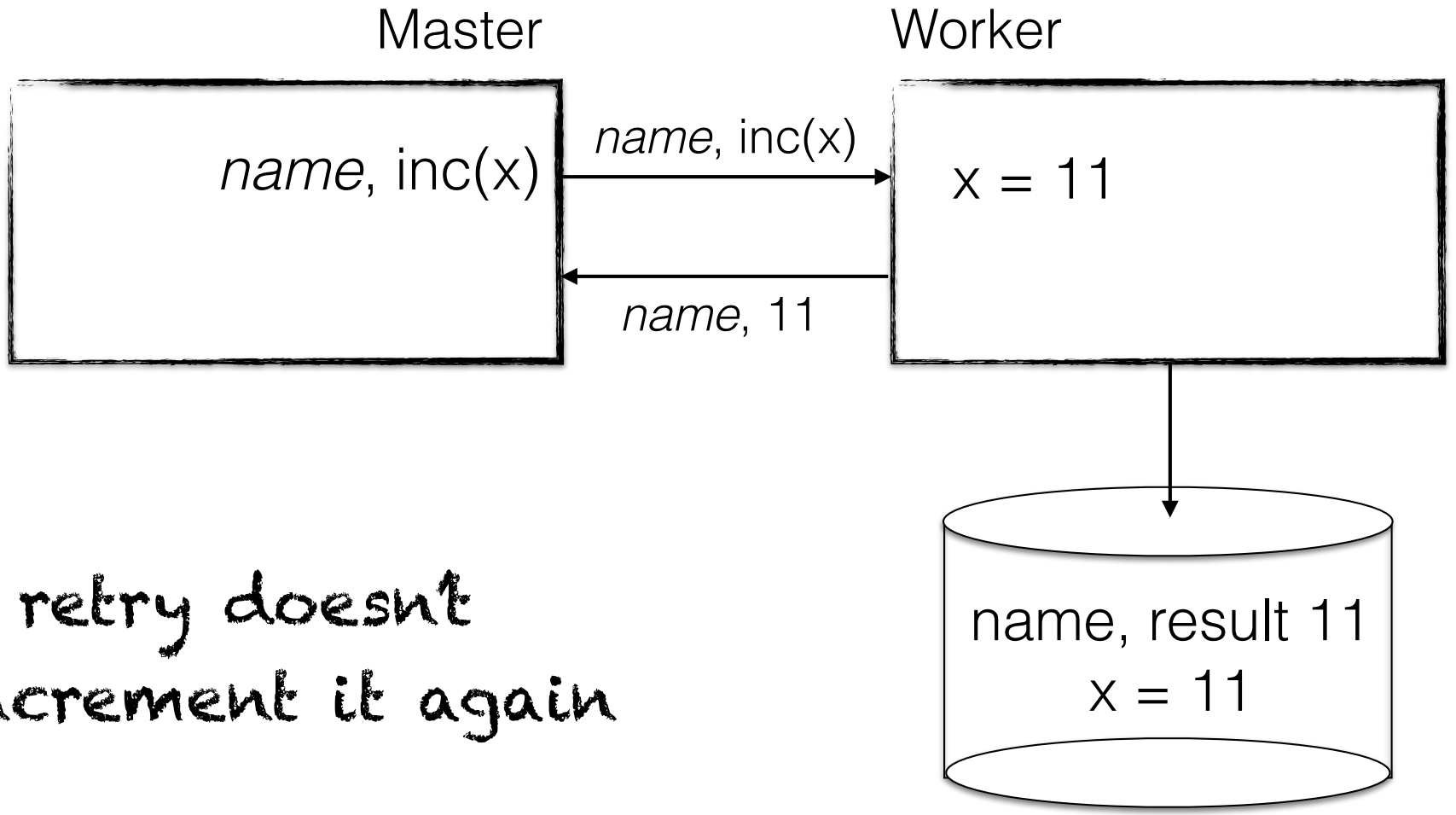
Give each operation a name, an identifier

Remote Reference/Remote Operation Model



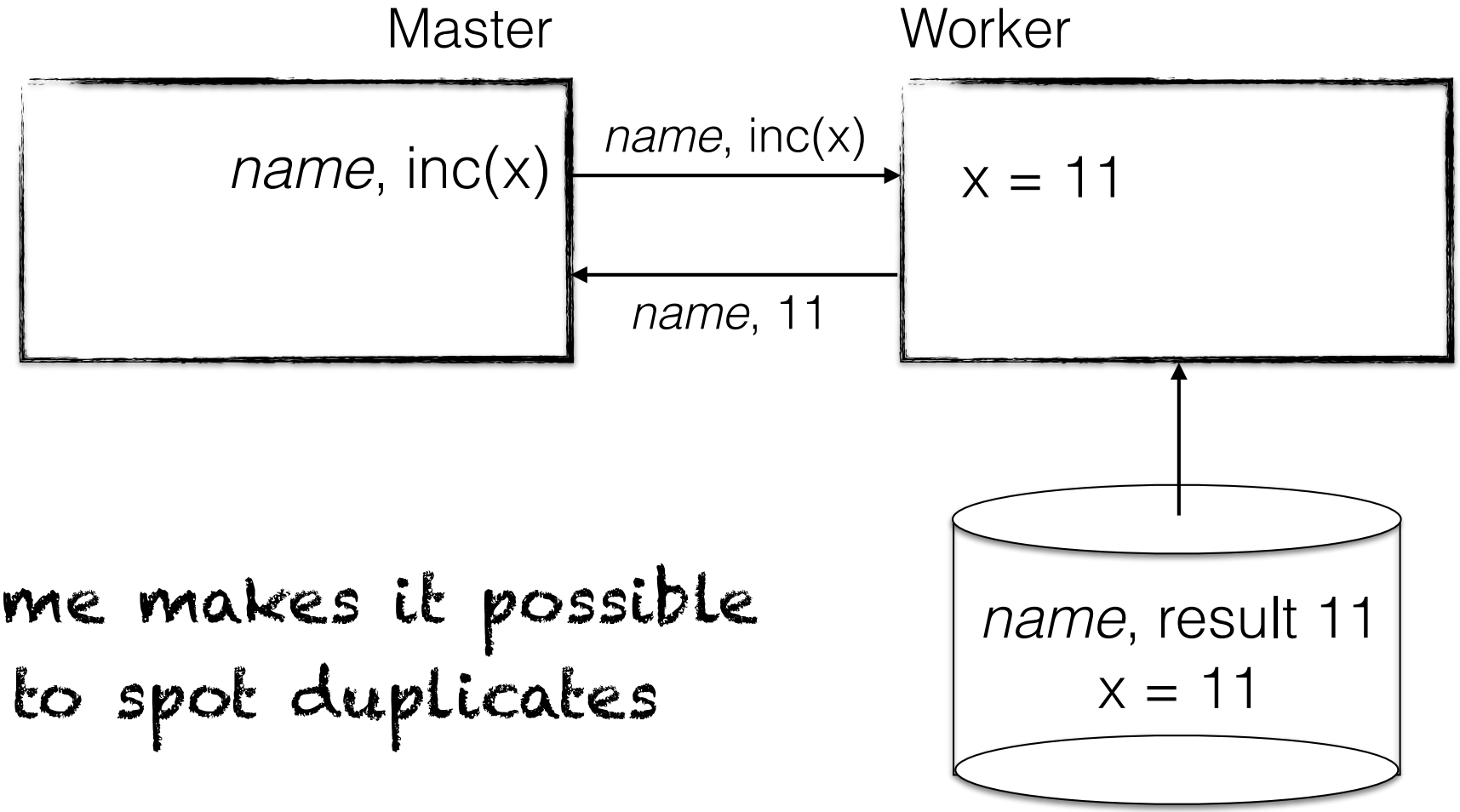
What if I retry now?

Remote Reference/Remote Operation Model



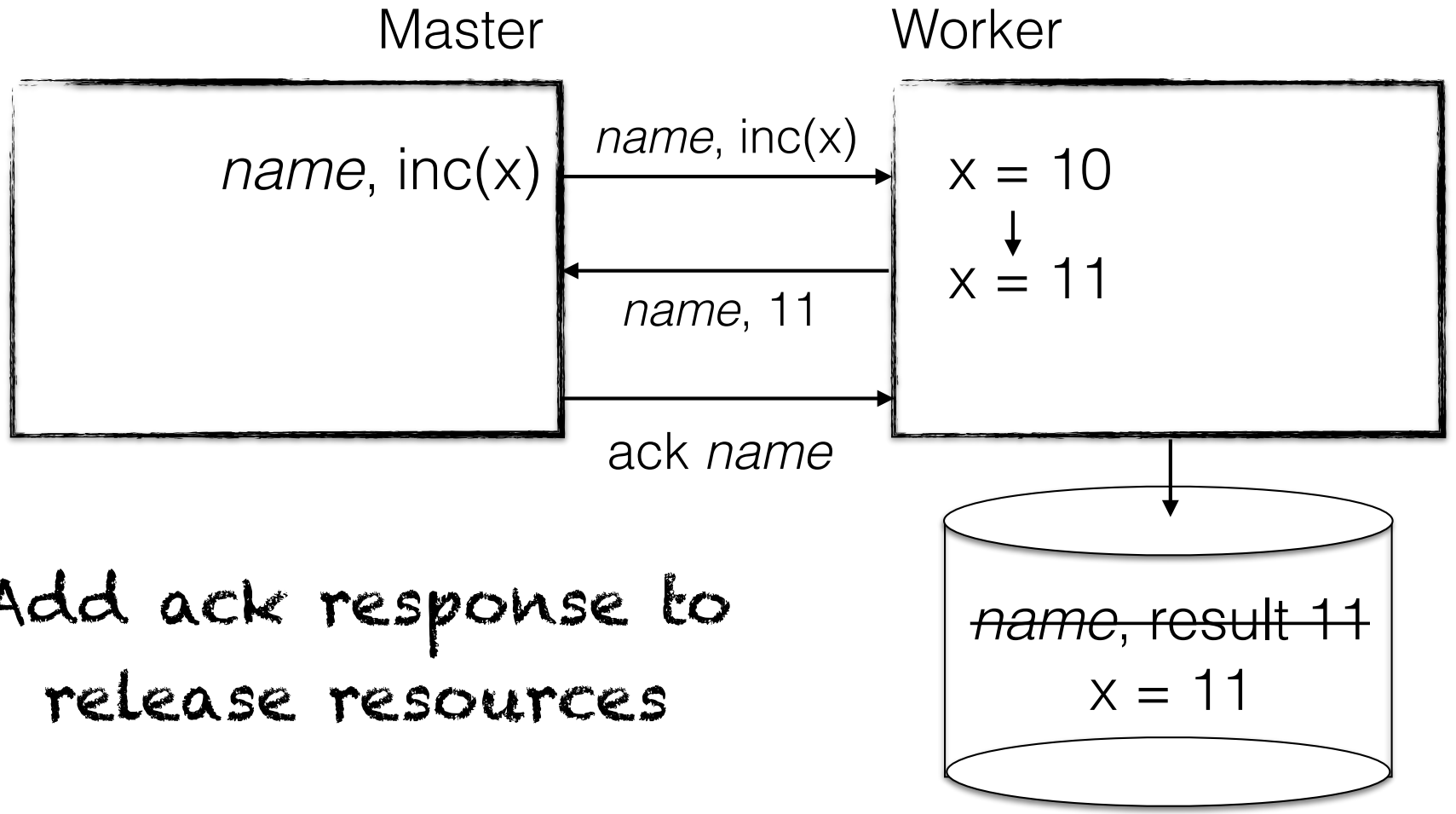
A retry doesn't increment it again

Remote Reference/Remote Operation Model



Name makes it possible to spot duplicates

Remote Reference/Remote Operation Model



Add ack response to release resources

Names

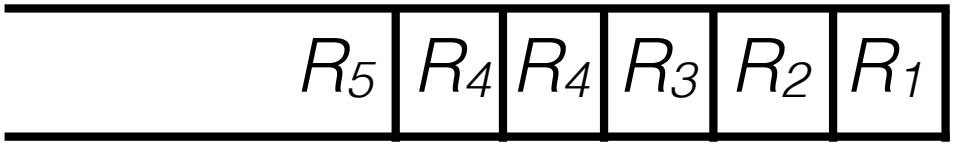
- Remote reference/Remote operation model
- Two levels
  - **Session**: logical connection between master and worker
  - **Sequence number**: order of request

What about large-scale?

- Multiple servers and clients
- Servers can crash
- Objects can migrate
- Challenges
  - System-wide mechanism for assigning unique client identifiers
  - Completion record durability
  - Match retries
  - Garbage collection
- *Reusable Infrastructure For Linearizability* [Lee et al., SOSP'15]
  - Can't guarantee linearizability in the presence of retries

Consensus and Exactly once

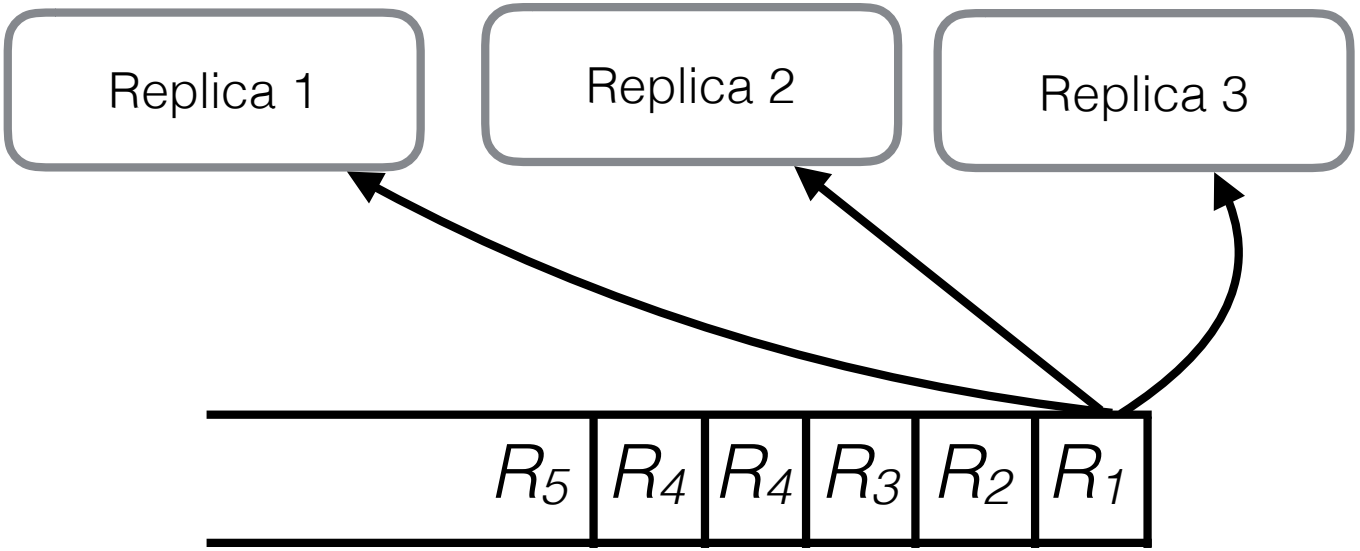
Agreement



Agreement on the sequence of requests to be executed

Agreement

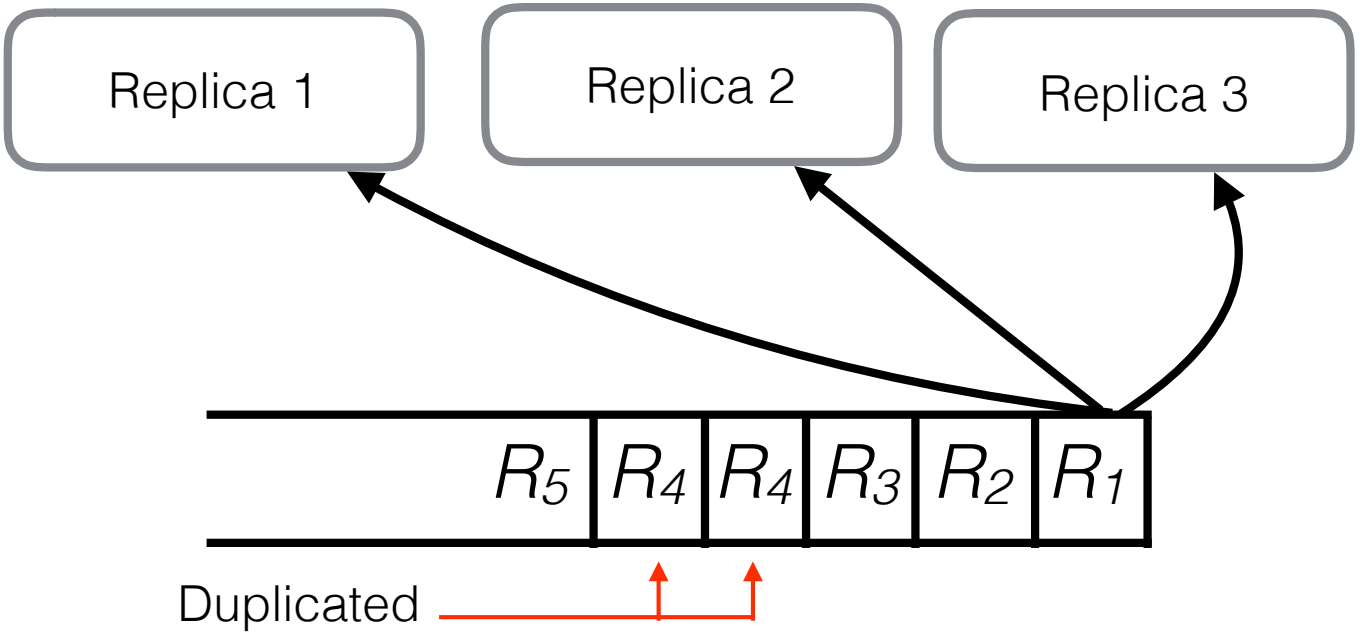
Replicas have a consistent state



Execute requests in the sequence agreed upon

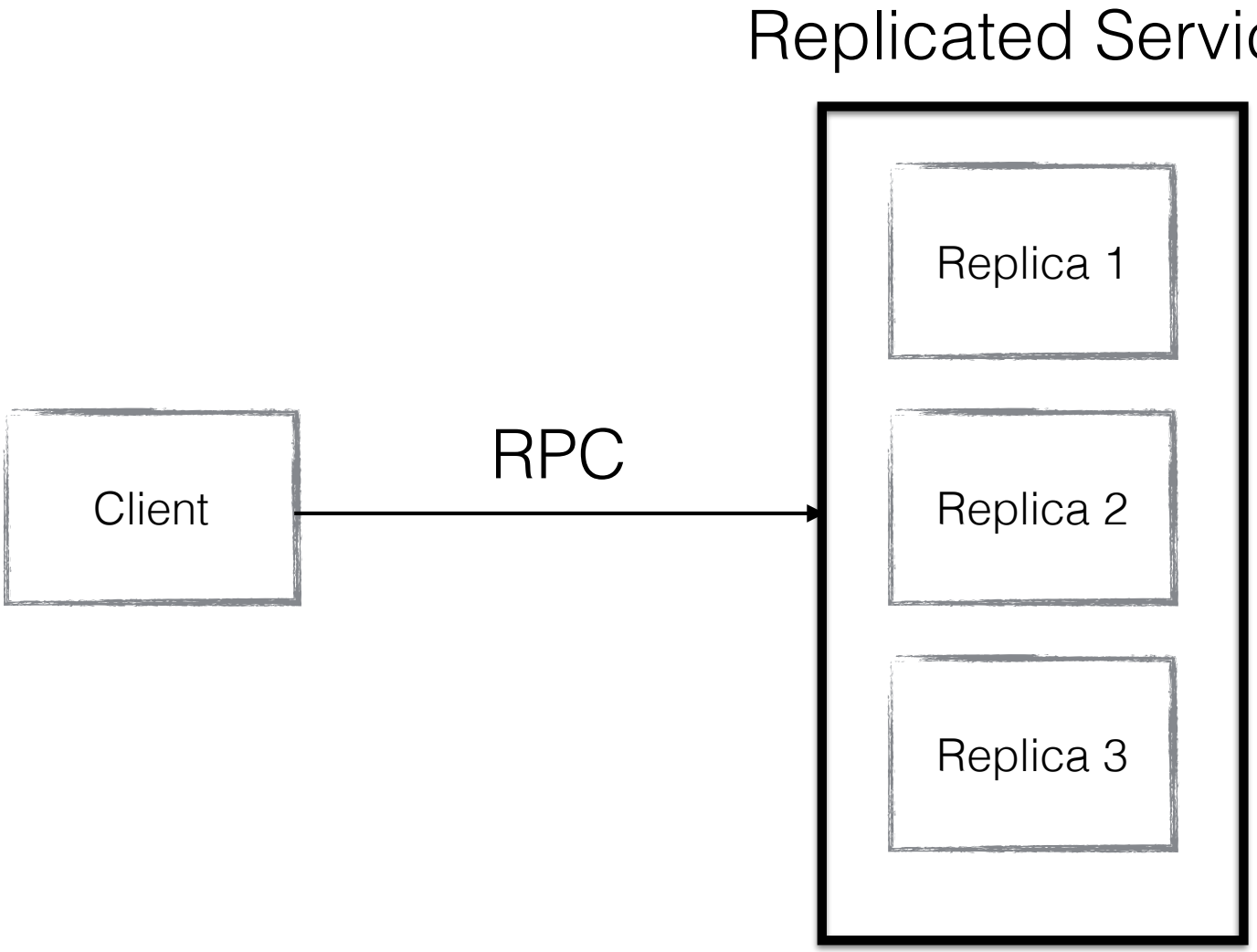
Agreement

Replicas have a consistent state

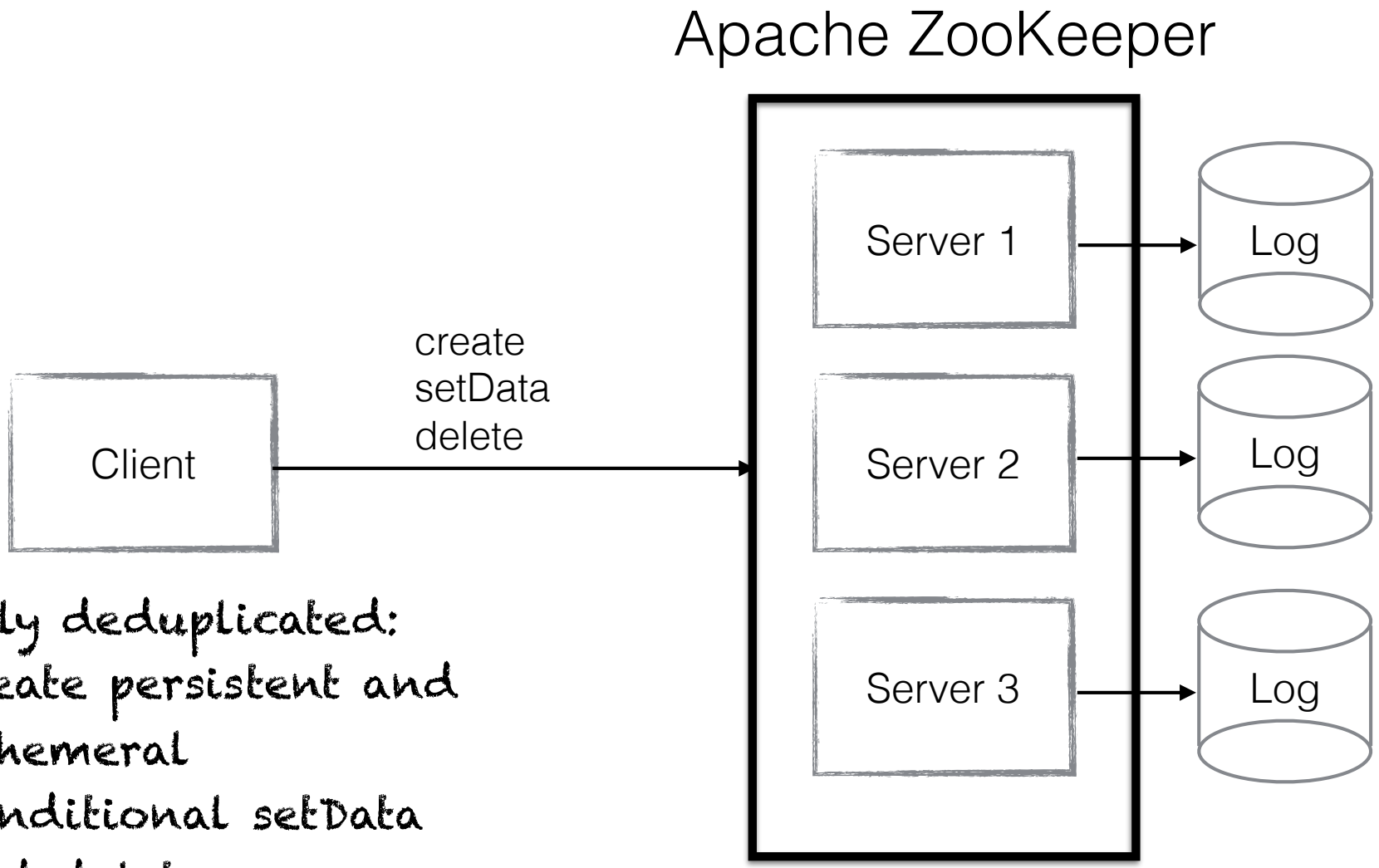


Replicas deduplicate by processing the sequence of requests

Using agreement in practice



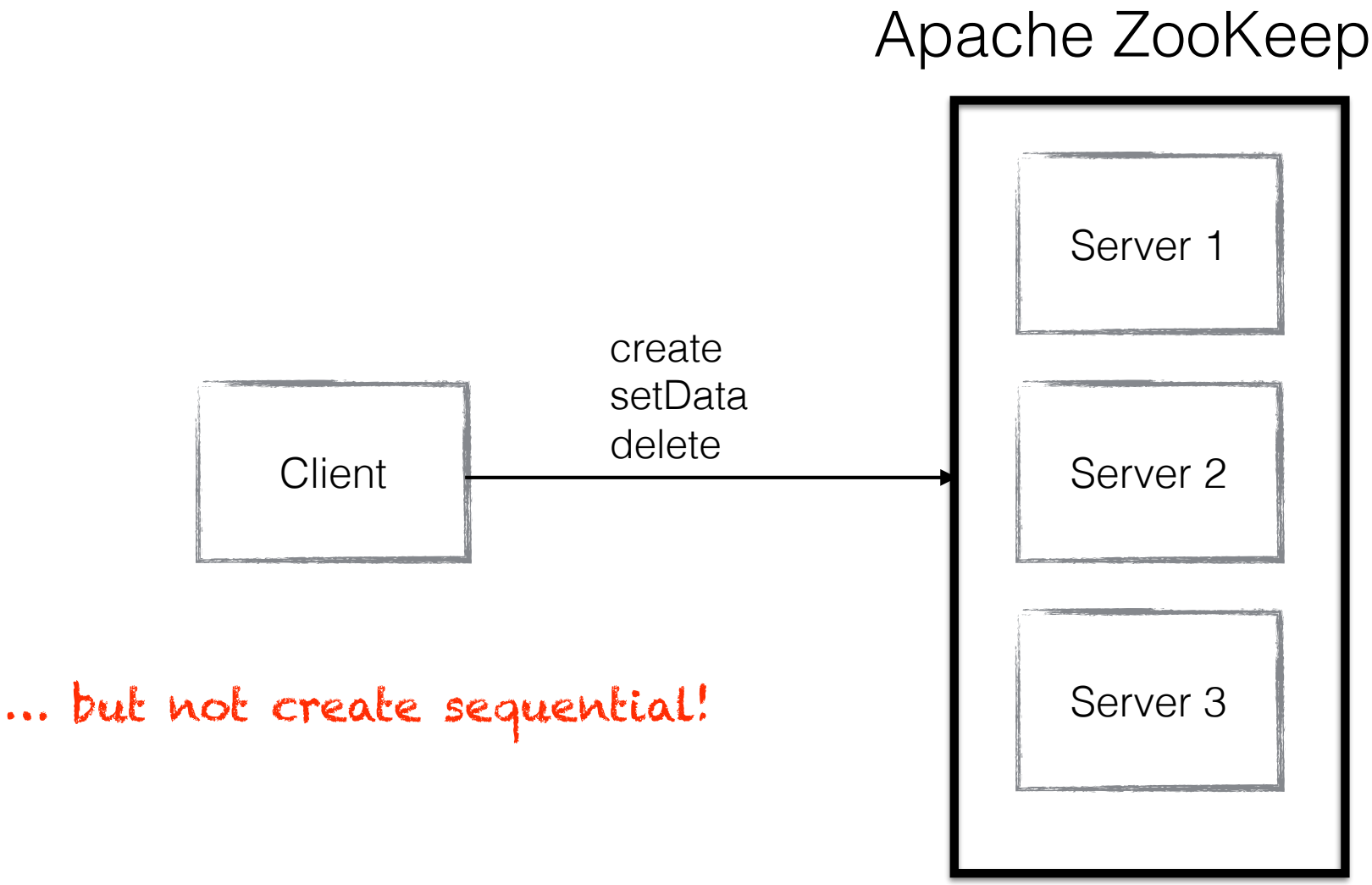
Using agreement in practice



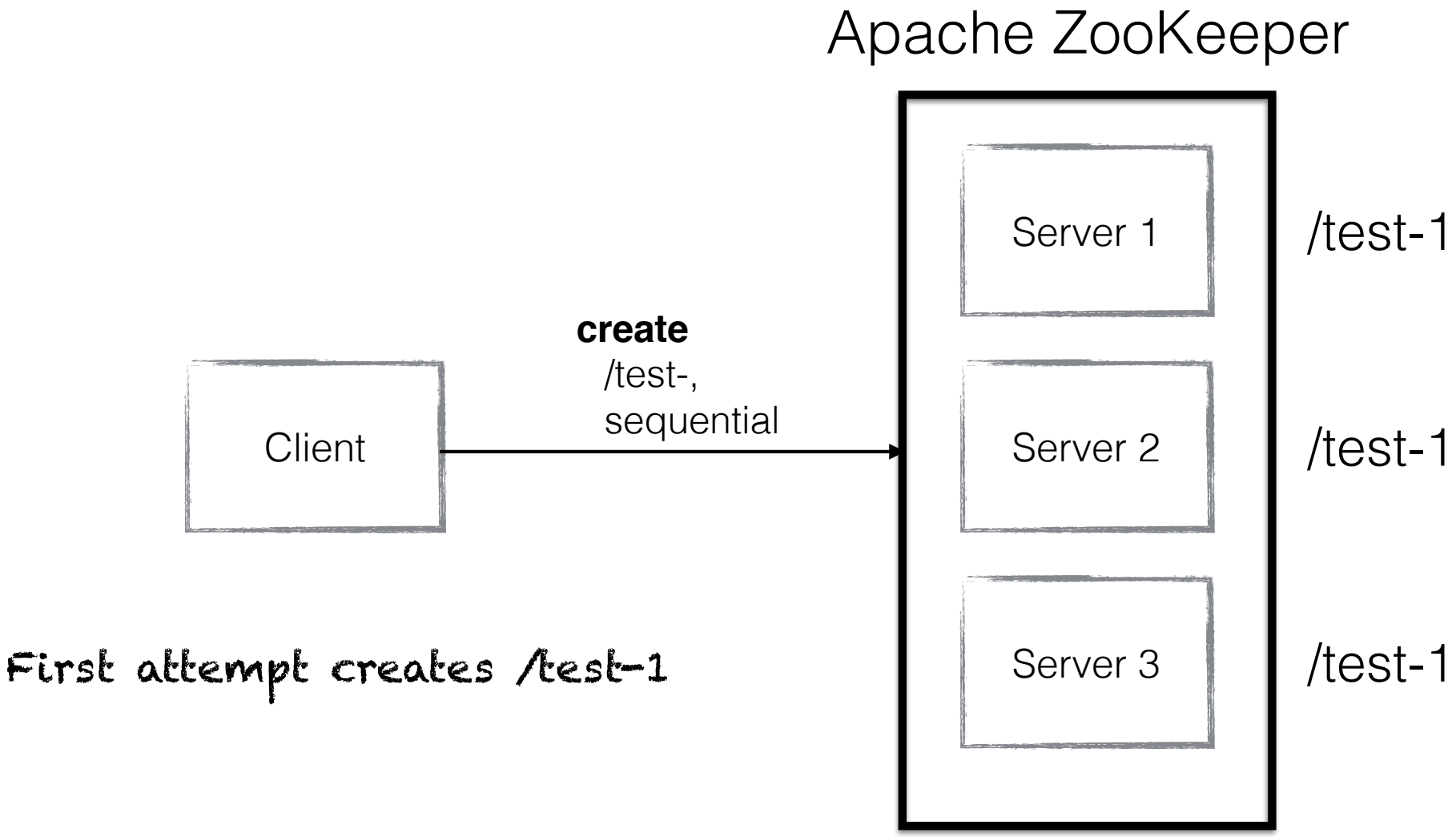
Naturally deduplicated:

- create persistent and ephemeral
- conditional setData and delete

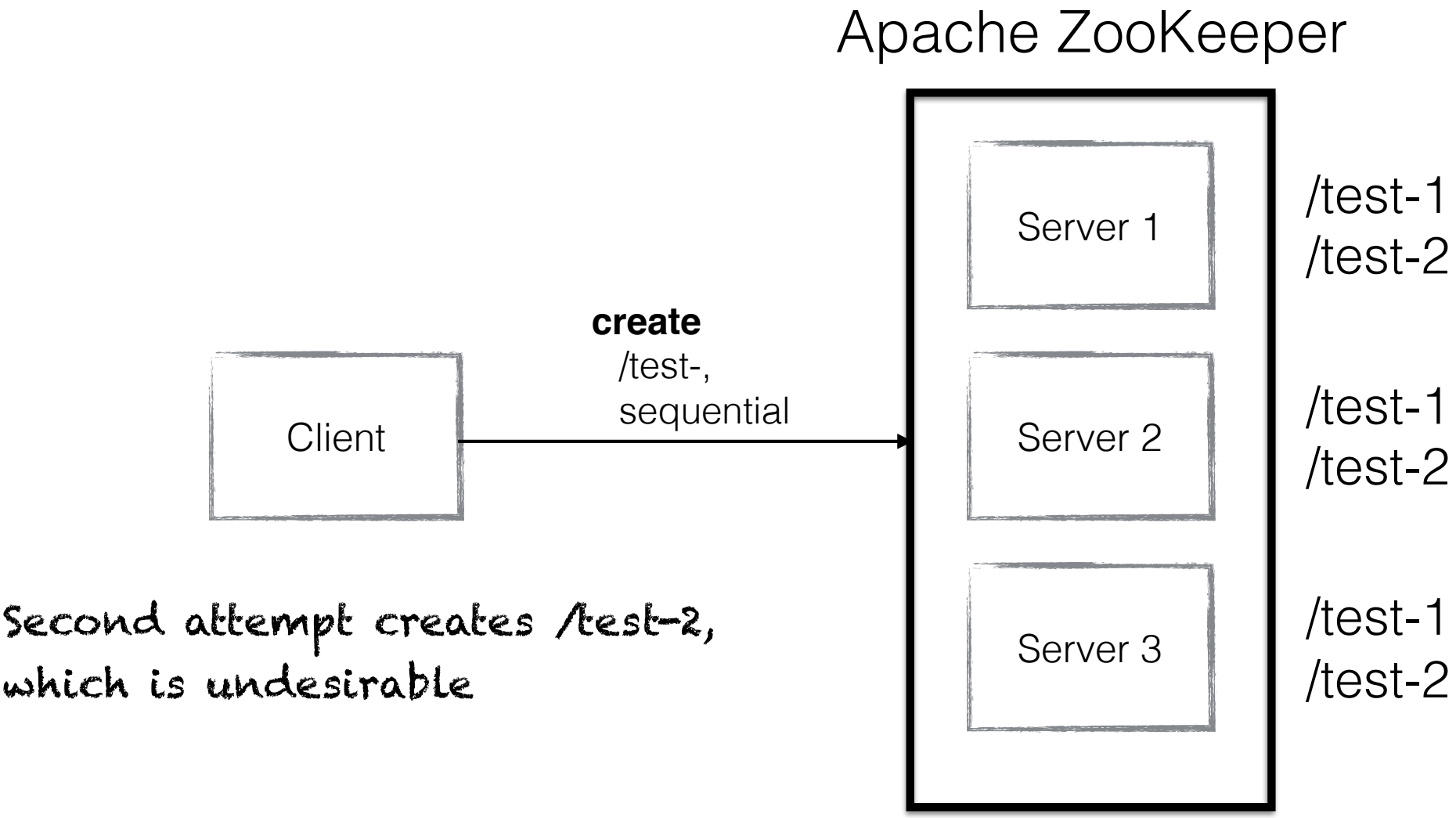
Using agreement in practice



Using agreement in practice



Using agreement in practice



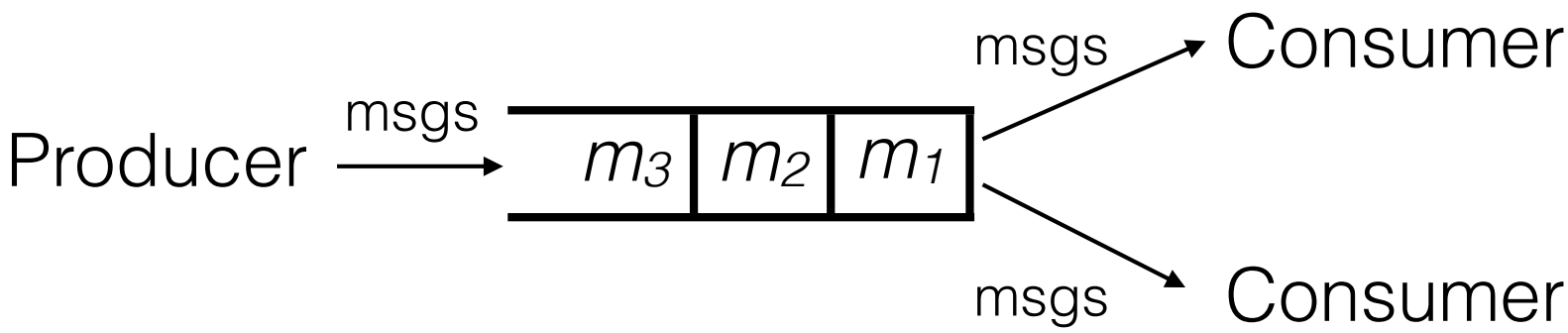
Retries

- ZooKeeper client currently does not retry
  - Error upon connection loss
- Possible by checking operation numbers

How does it help with building real-time message processing?

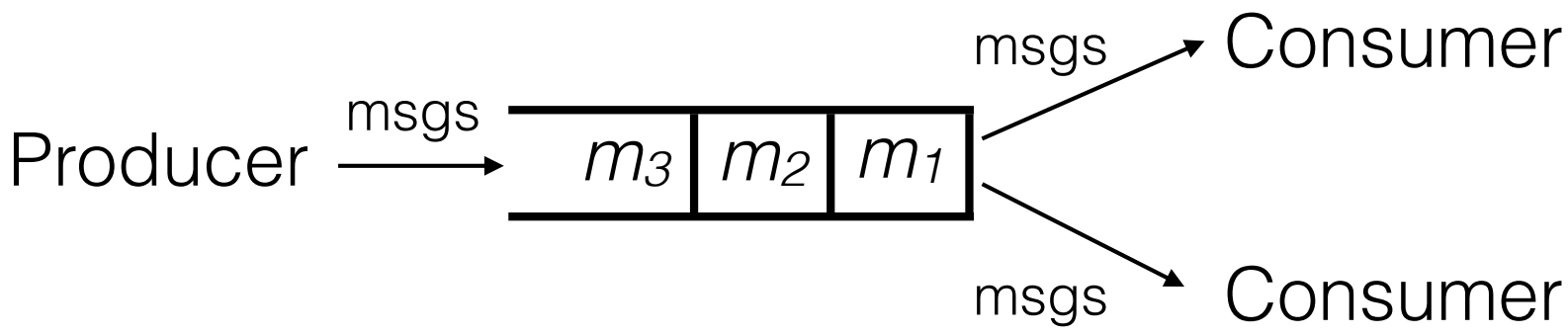
Exactly-once for Messaging

Messaging simplified



Messaging simplified

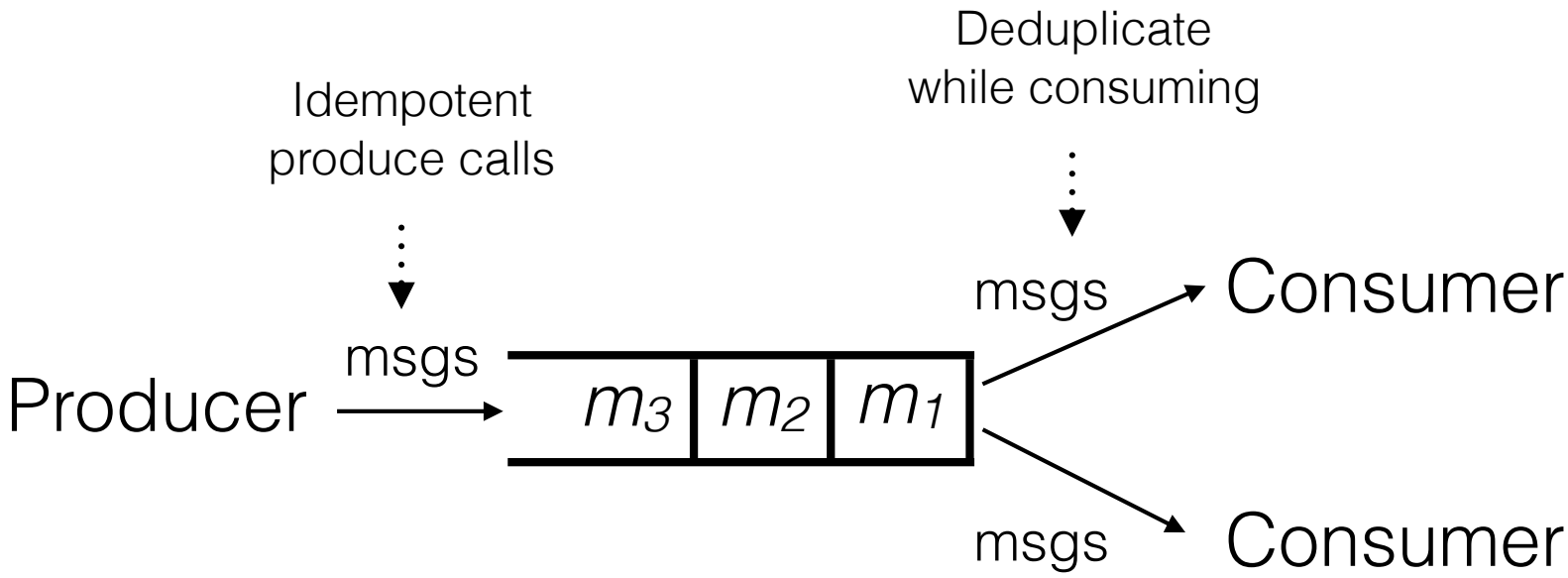
Apache Kafka



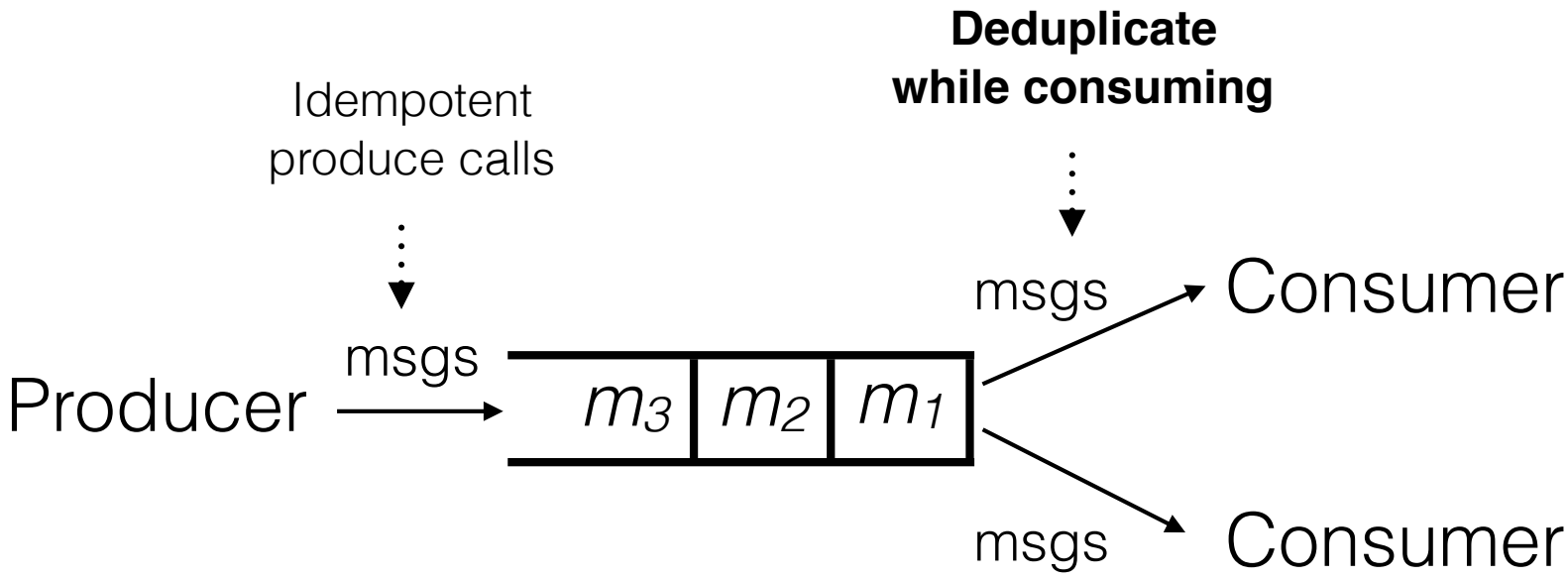
Messages are persisted and appended to a log

(see it as a replicated state machine)

Messaging simplified

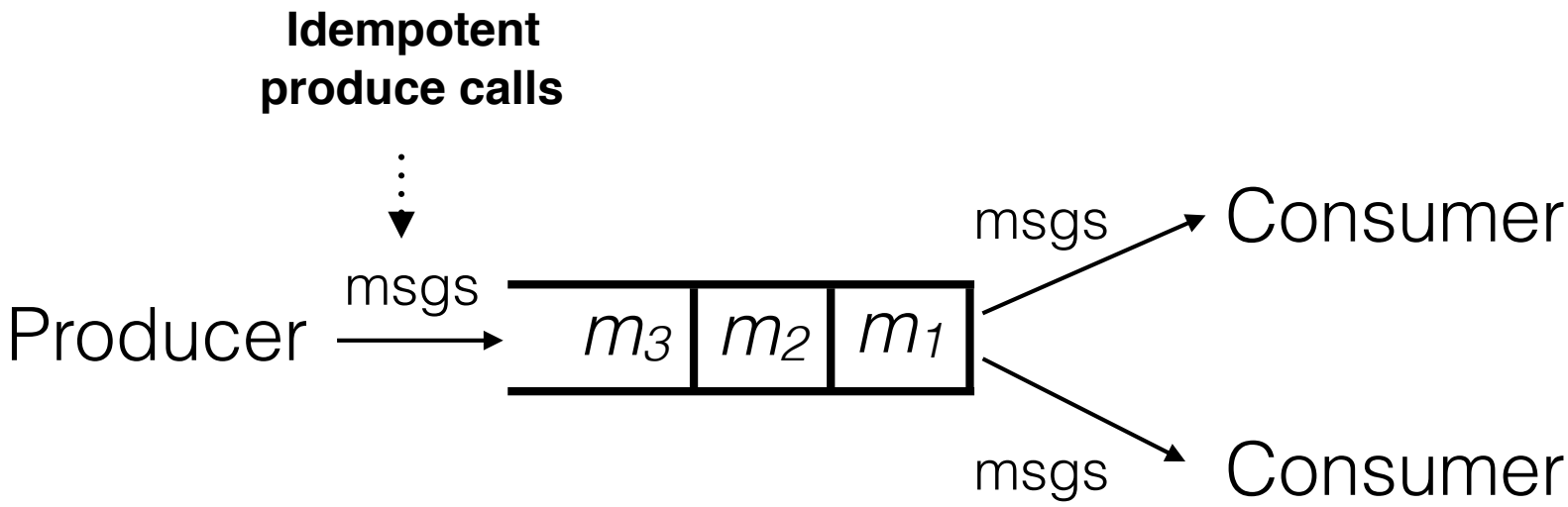


Messaging simplified



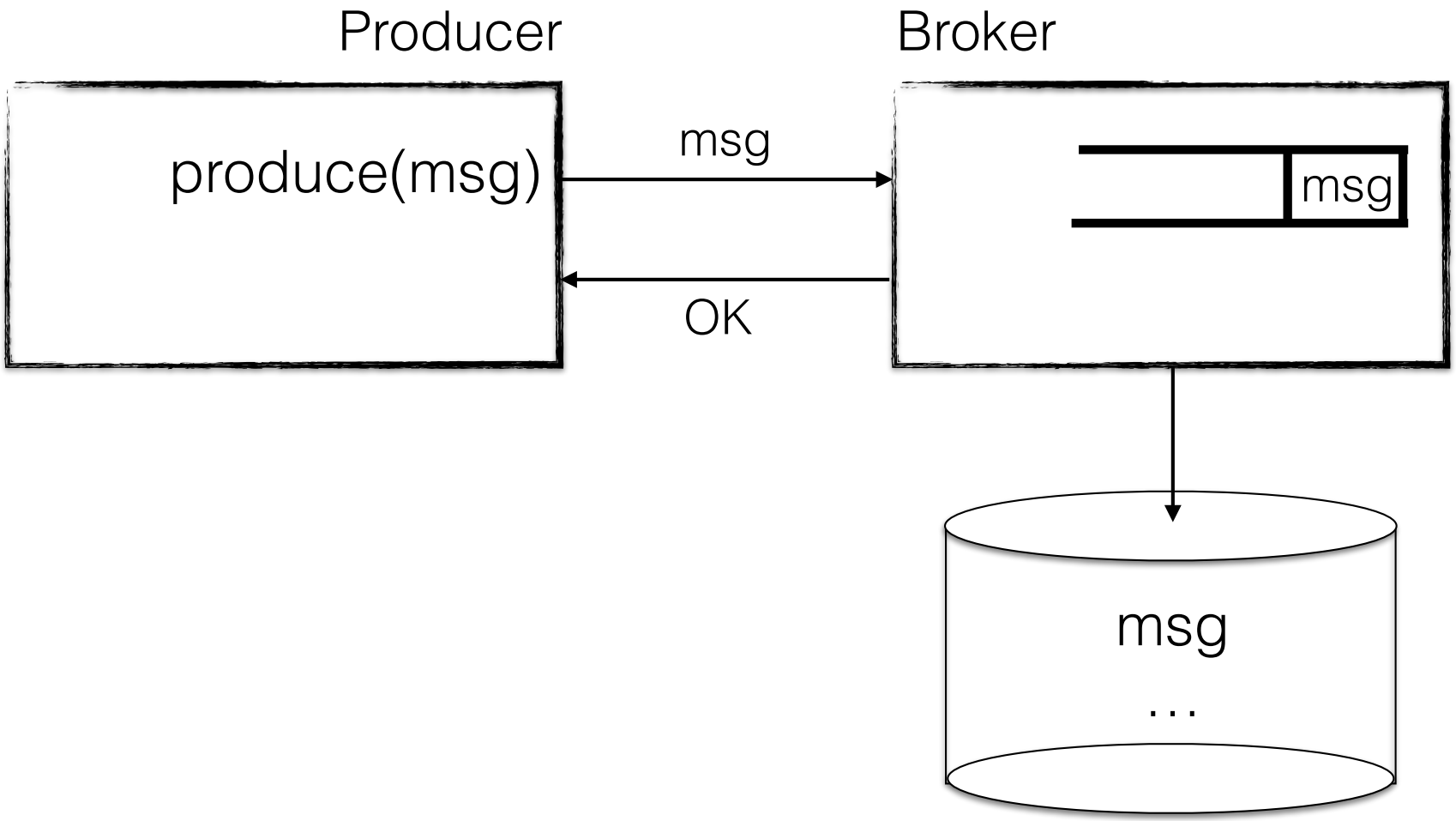
- ✓ If done by consumer only
  - ✓ difficult to resume from arbitrary position
- ✓ If broker checks for duplicates
  - ✓ Better off discarding when produced

Messaging simplified

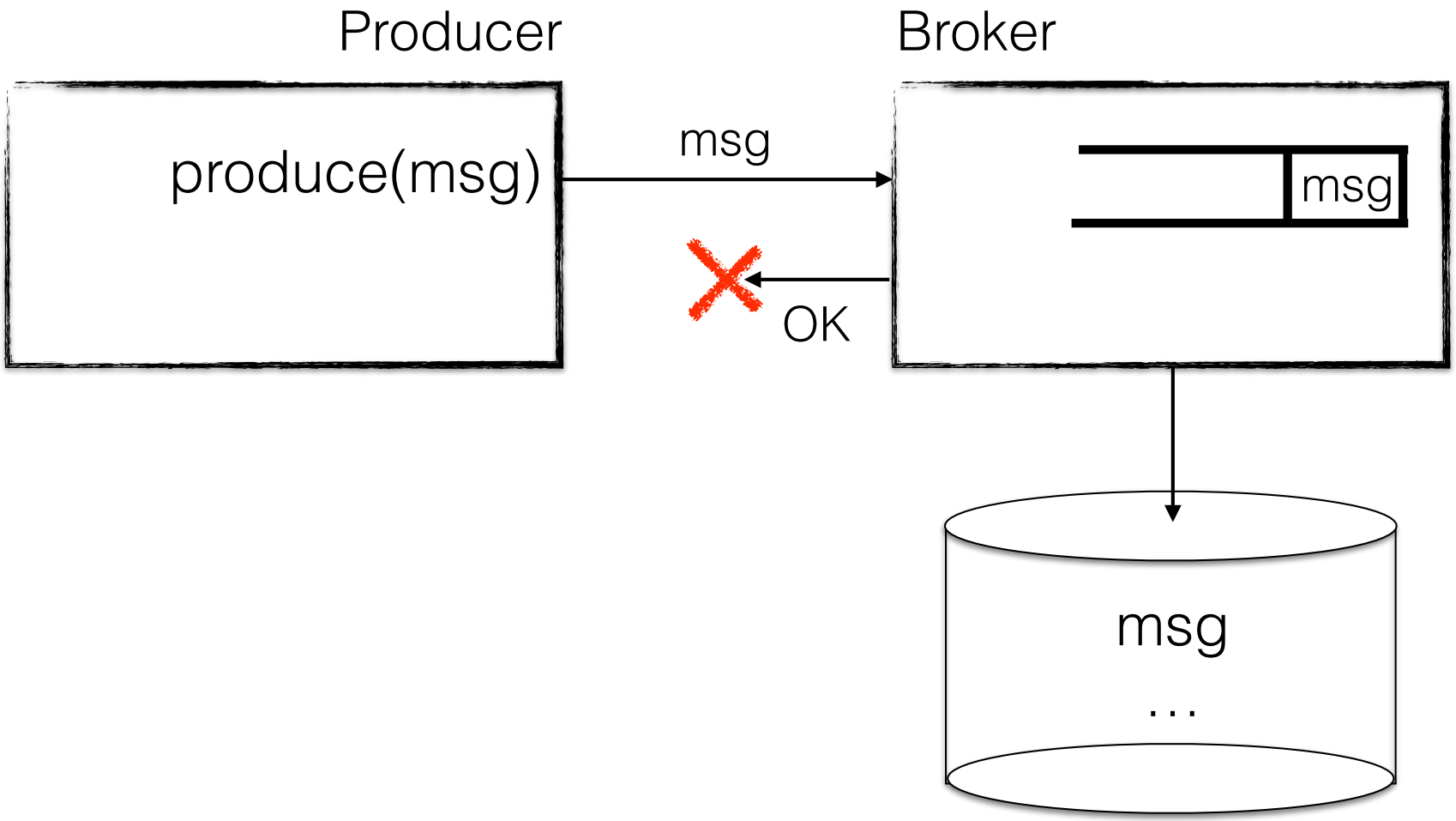


✓ Focus on making produce calls idempotent

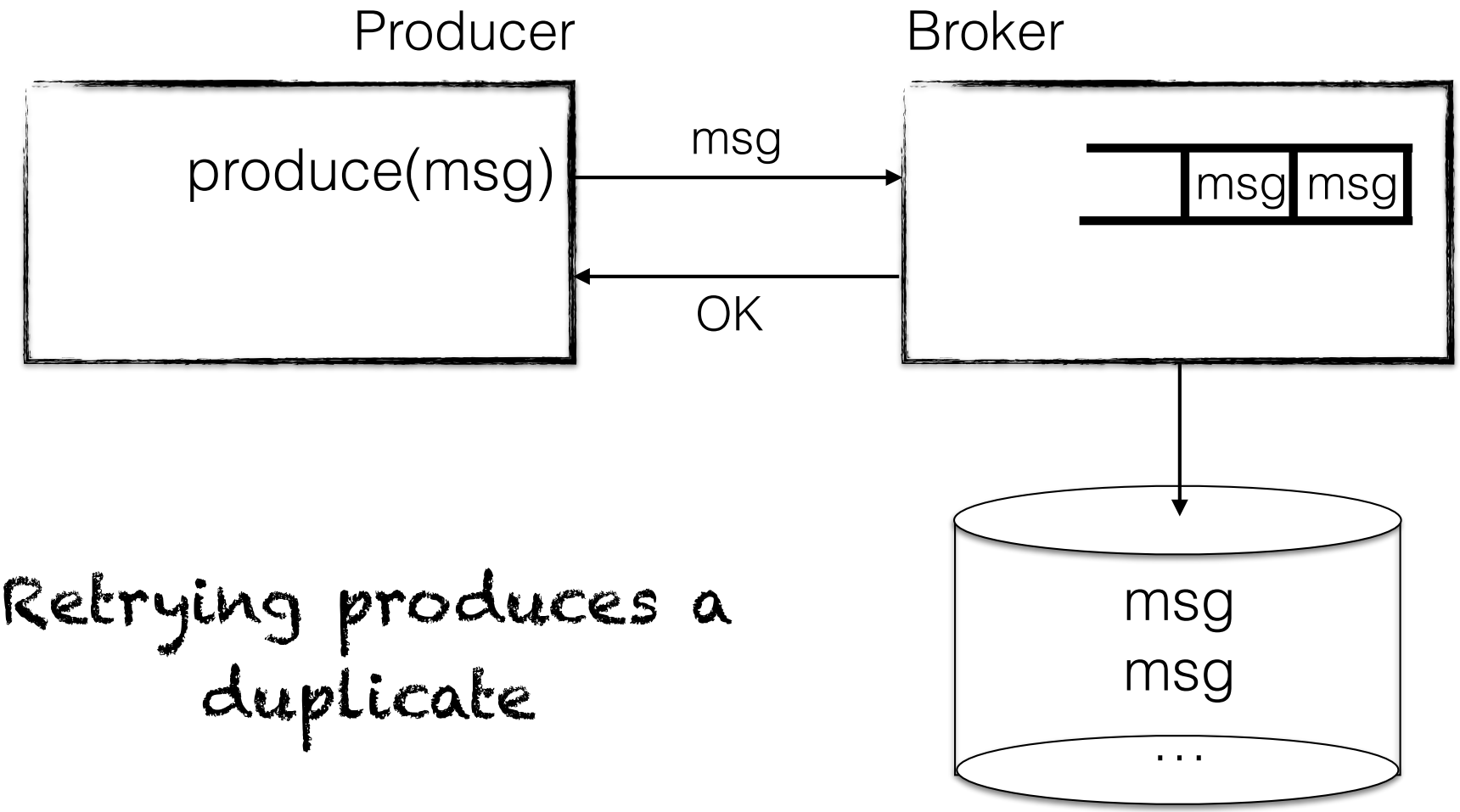
Messaging



Messaging

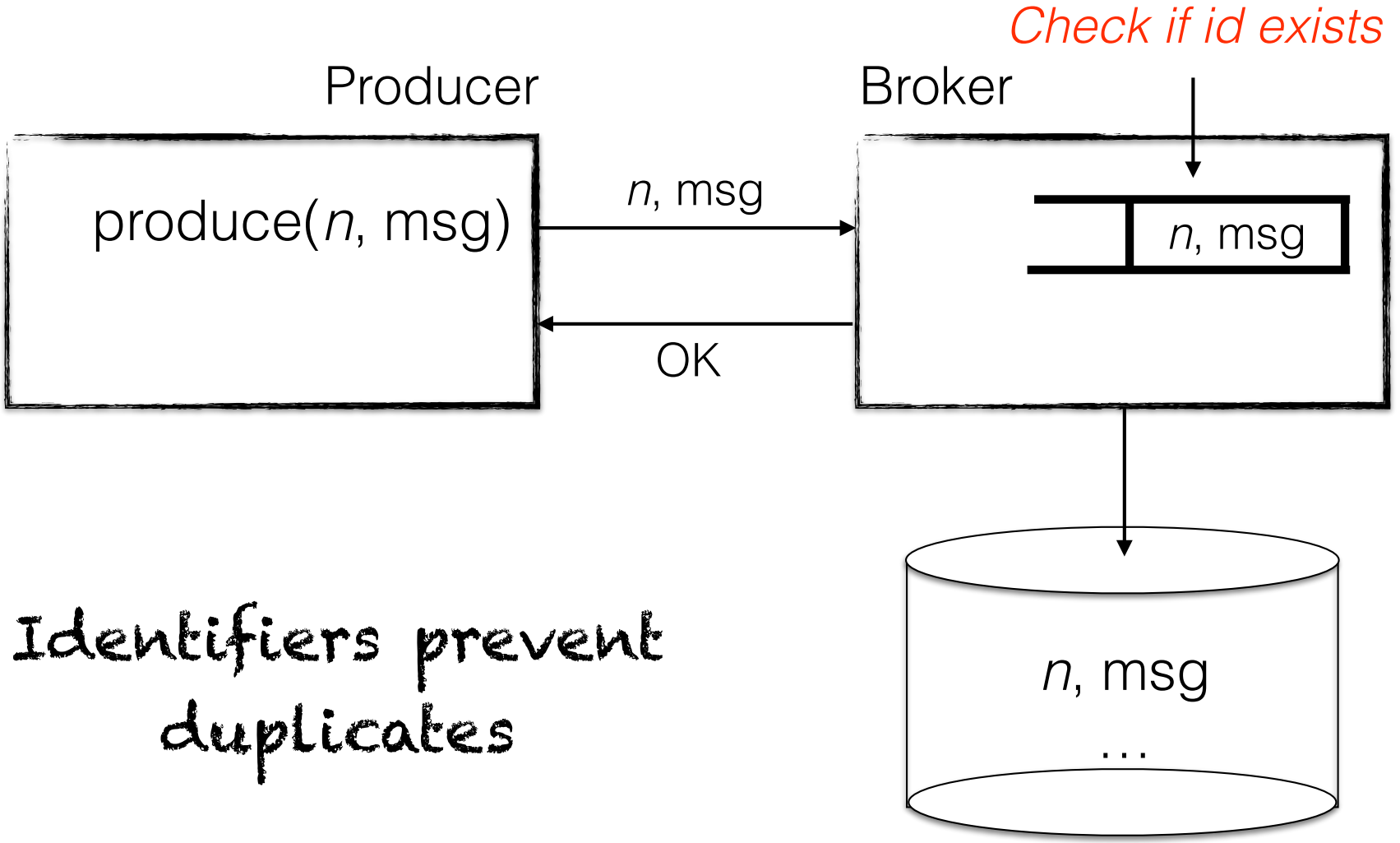


Messaging



Retrying produces a duplicate

Messaging - Avoiding the duplicate

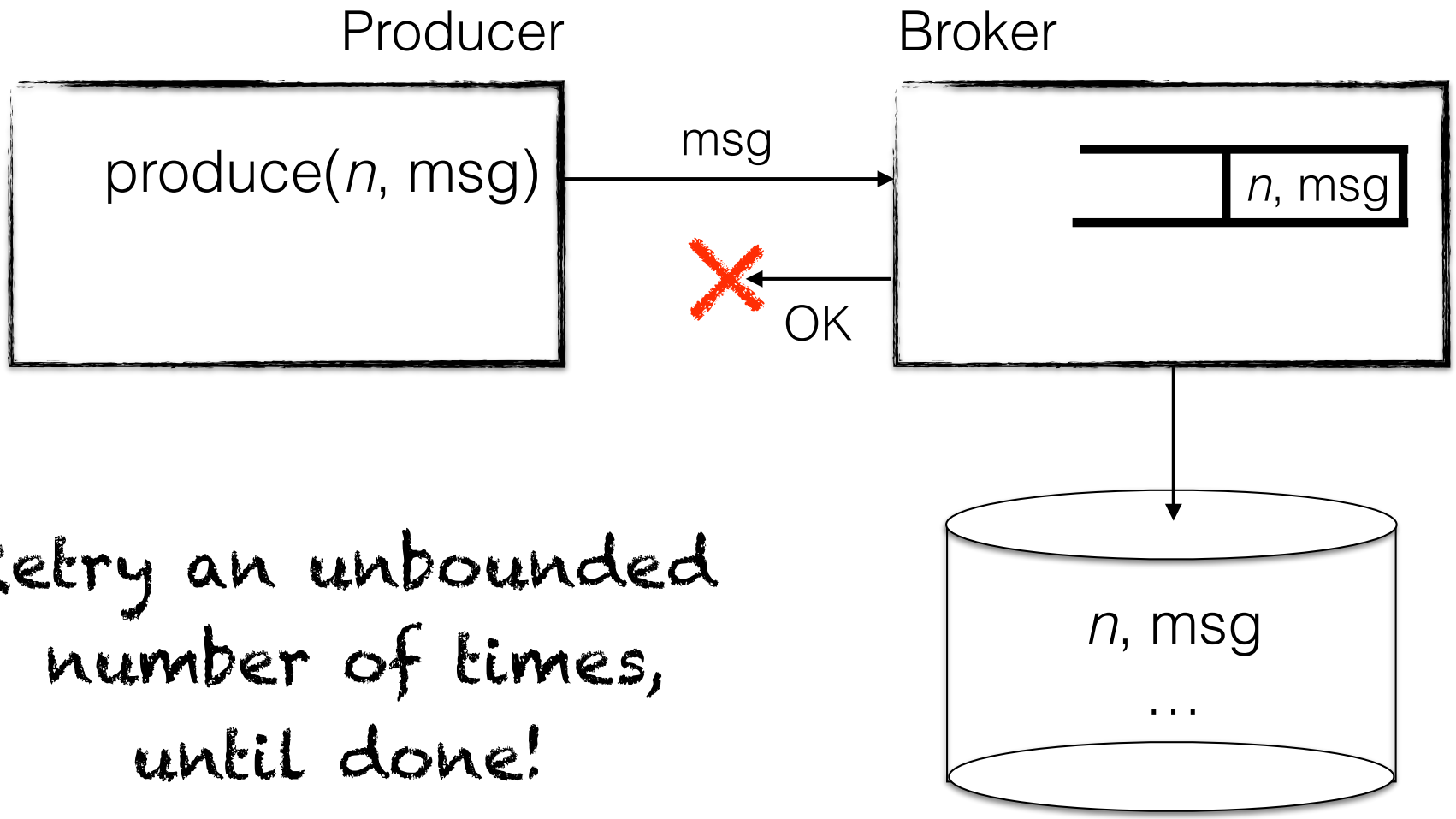


Identifiers prevent duplicates

Looking more closely

- Id guarantees no more than once
- **At least once**
  - ✓ Retry upon network error
  - ✓ Retry forever? How long is enough?
- **At most once - Id verification**
  - ✓ Use separate data structure?
  - ✓ Store all identifiers ever seen?

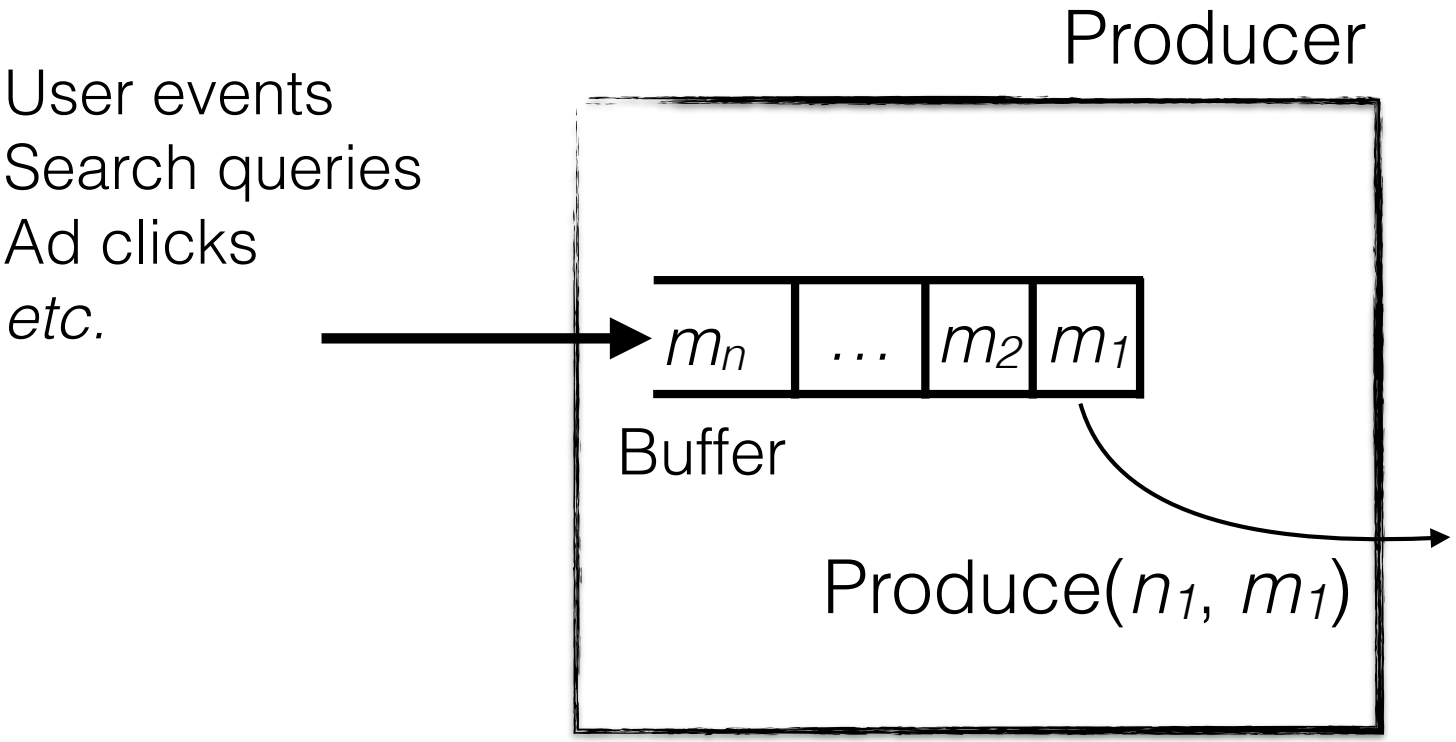
At least once



Retry an unbounded number of times, until done!

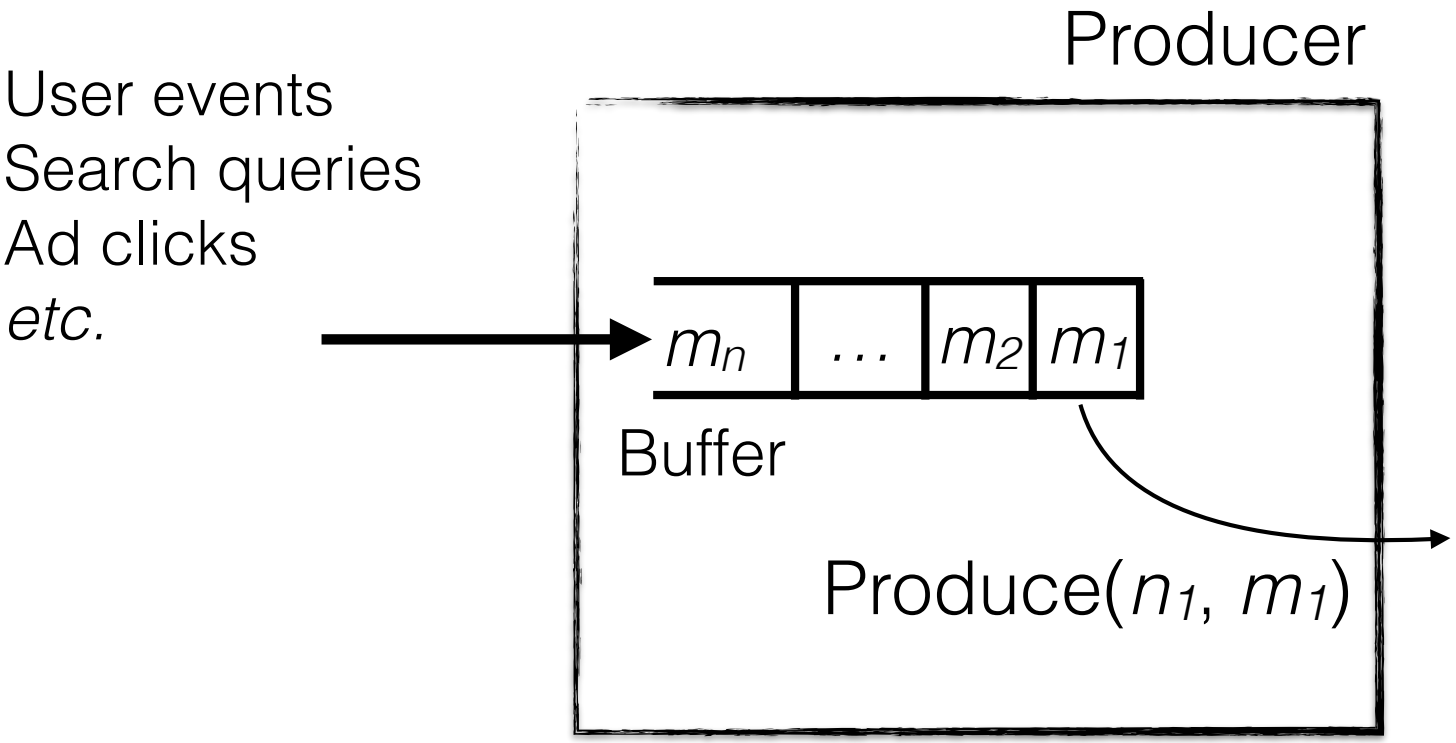
Safe, but not live…

At least once



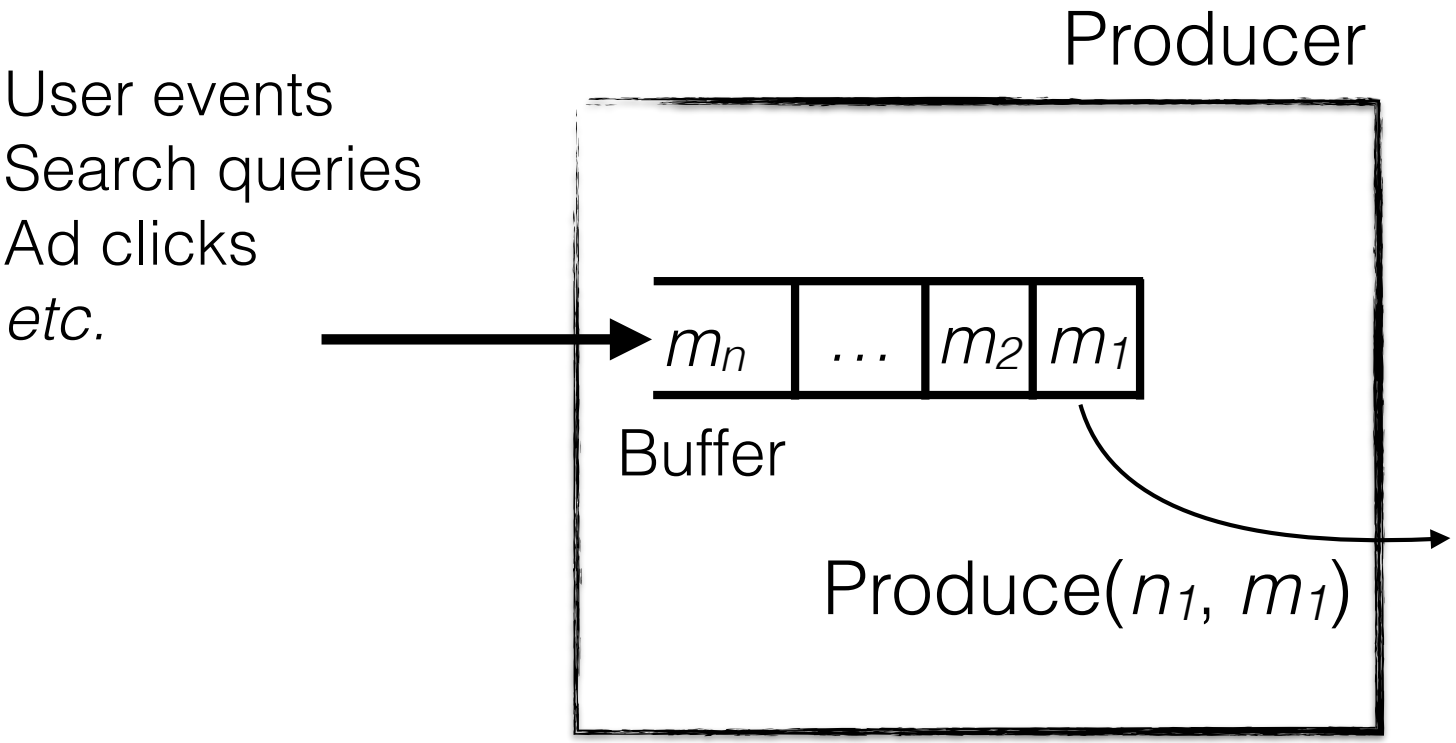
Retrying indefinitely causes the buffer to fill up

At least once



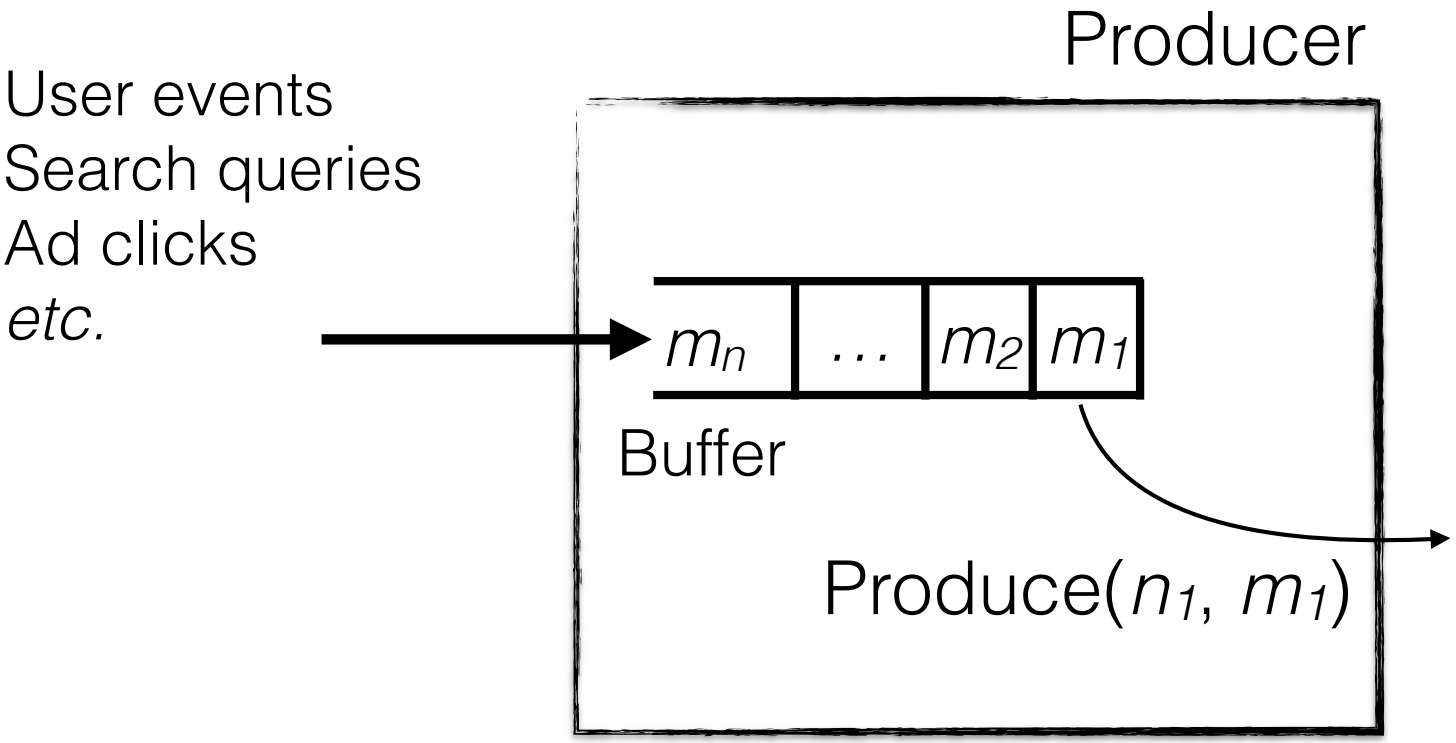
Eventually, we have to start dropping events

At least once



... and the at-least-once guarantee is violated

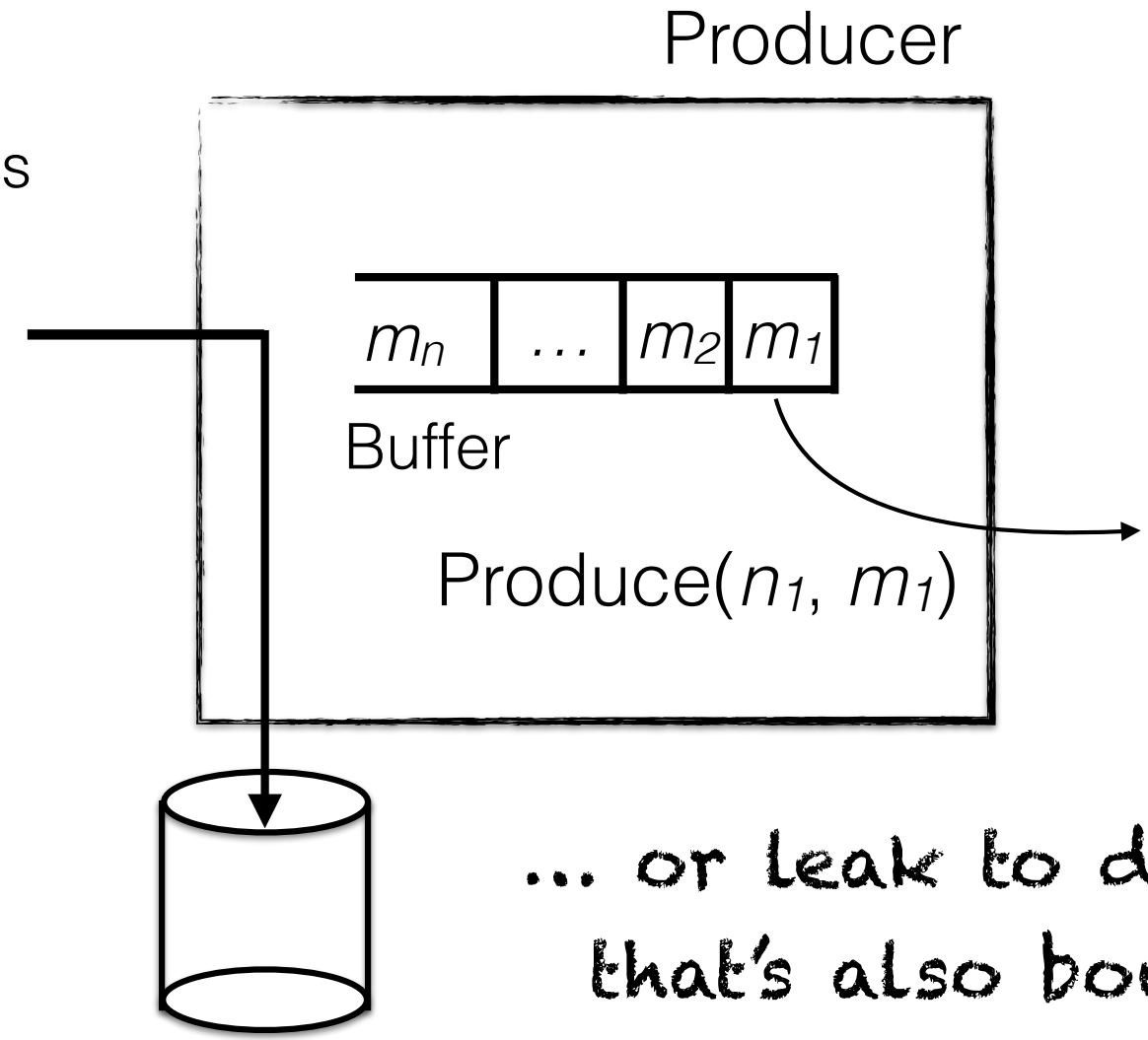
At least once



... unless you can apply back pressure

At least once

User events
Search queries
Ad clicks
etc.

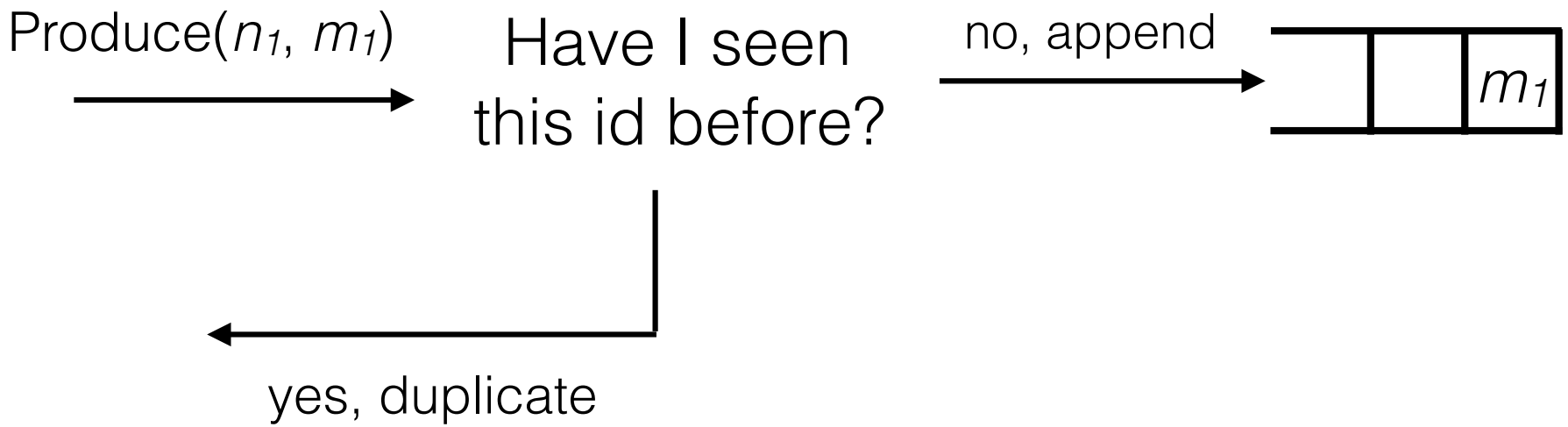


... or leak to disk, but that's also bounded

At least once

- To enforce it
  - ✓ Might need to stall
  - ✓ Not live for some unknown period
  - ✓ Pressure back to source
- When progress is strictly necessary, drop

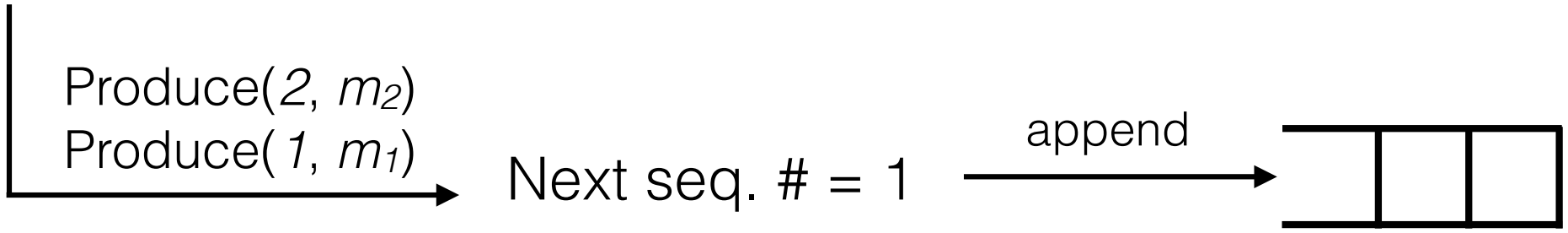
Id verification



Have I seen this Id before?

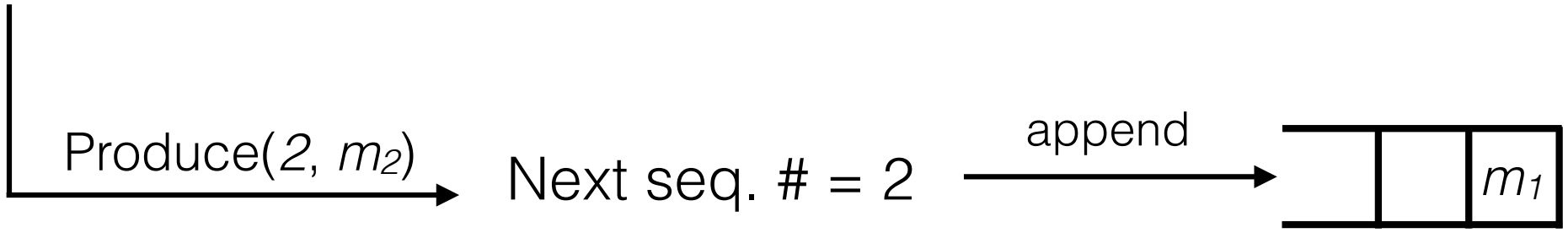
- Choices
  - Sequence number?
  - Arbitrary byte vectors or strings?
- Critical for efficient processing

Sequence numbers



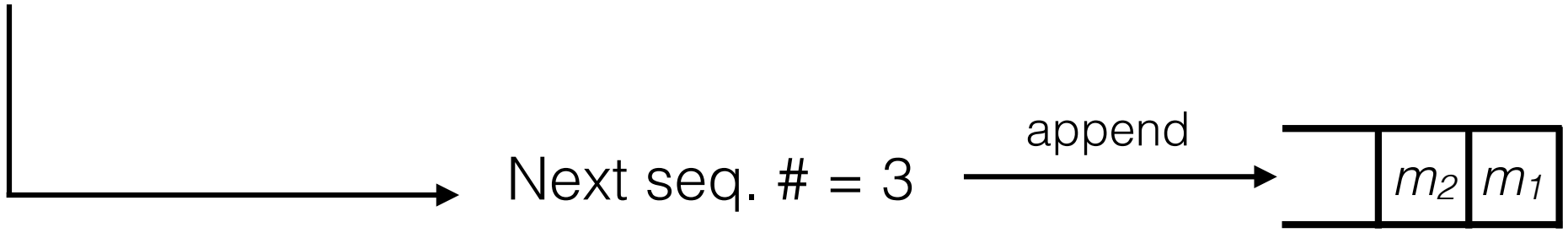
✓ Expected sequence number is 1

Sequence numbers



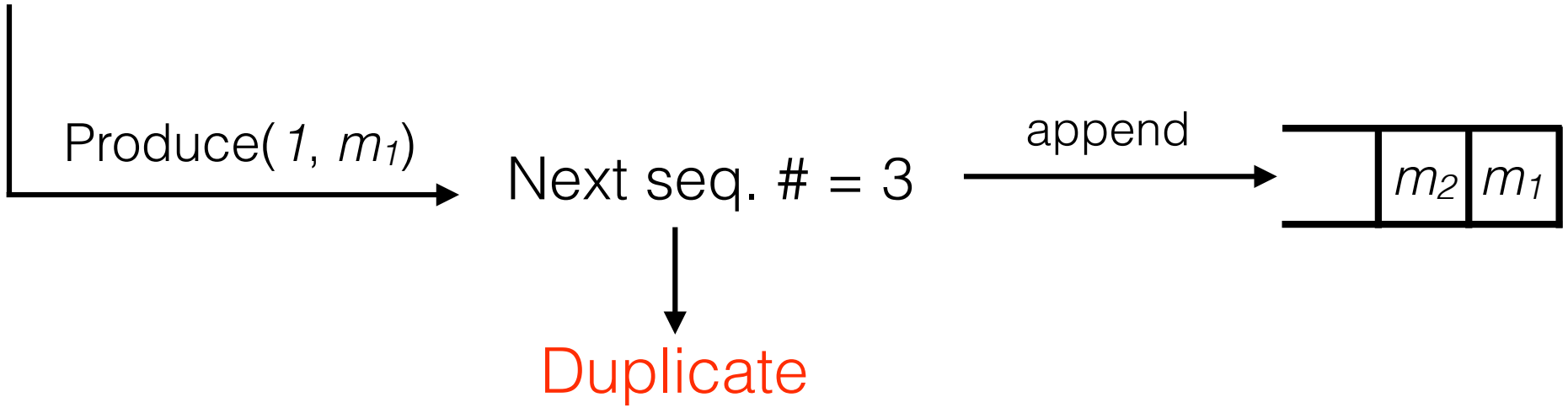
- ✓ Message is successfully produced
- ✓ Advance expected sequence number to 2

Sequence numbers



✓ Advance expected sequence number to 3

Sequence numbers

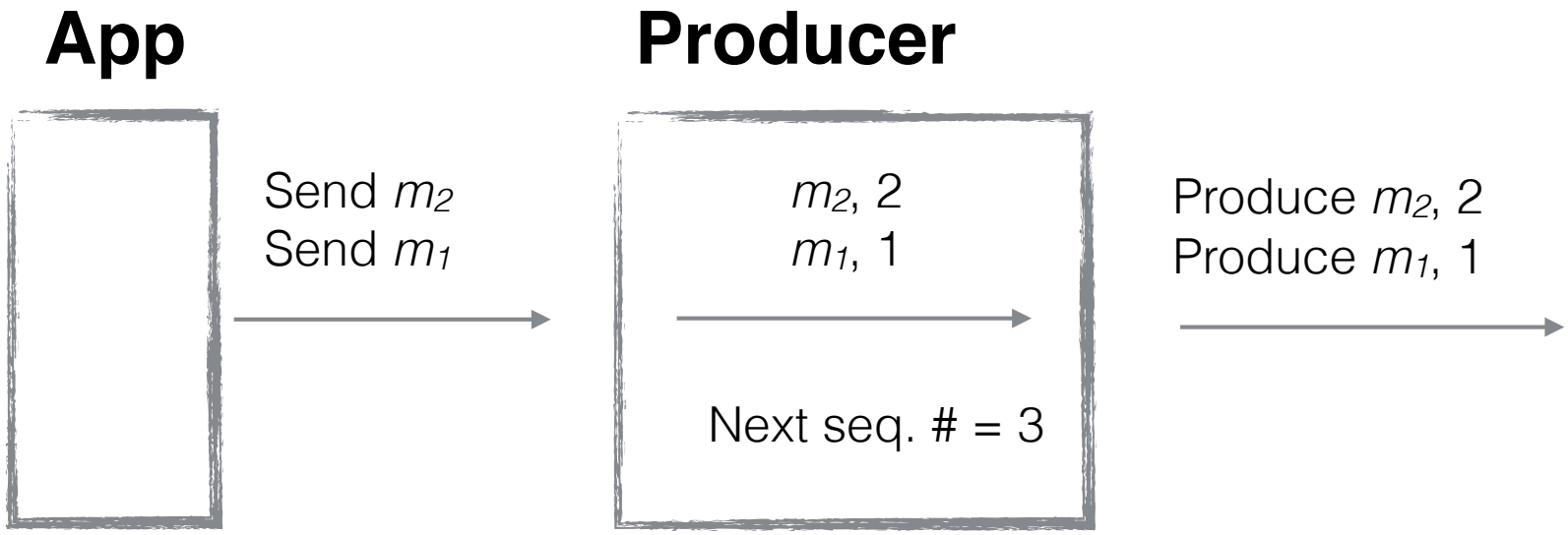


- ✓ Produce request fails
- ✓ Expected is 3
- ✓ Produce request is 1

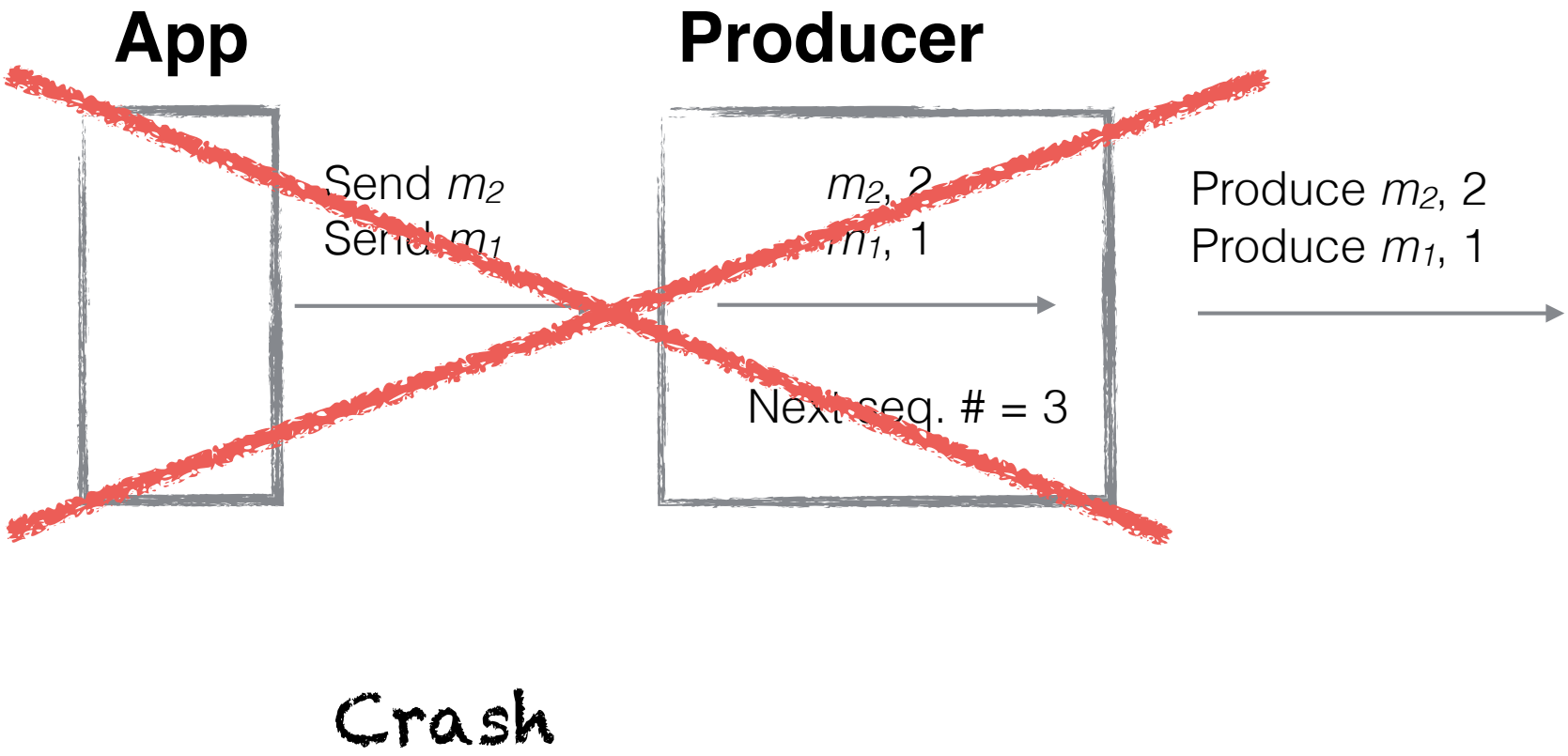
Where?

- **Remote Reference/Remote Operation Model**
- **Kafka - Idempotent producer (proposal)**
  - Unique identifiers for produced messages
  - Id = Producer Id + Epoch + Seq. #
- **RIFL - SOSP'15**
  - Linearizability for RPCs

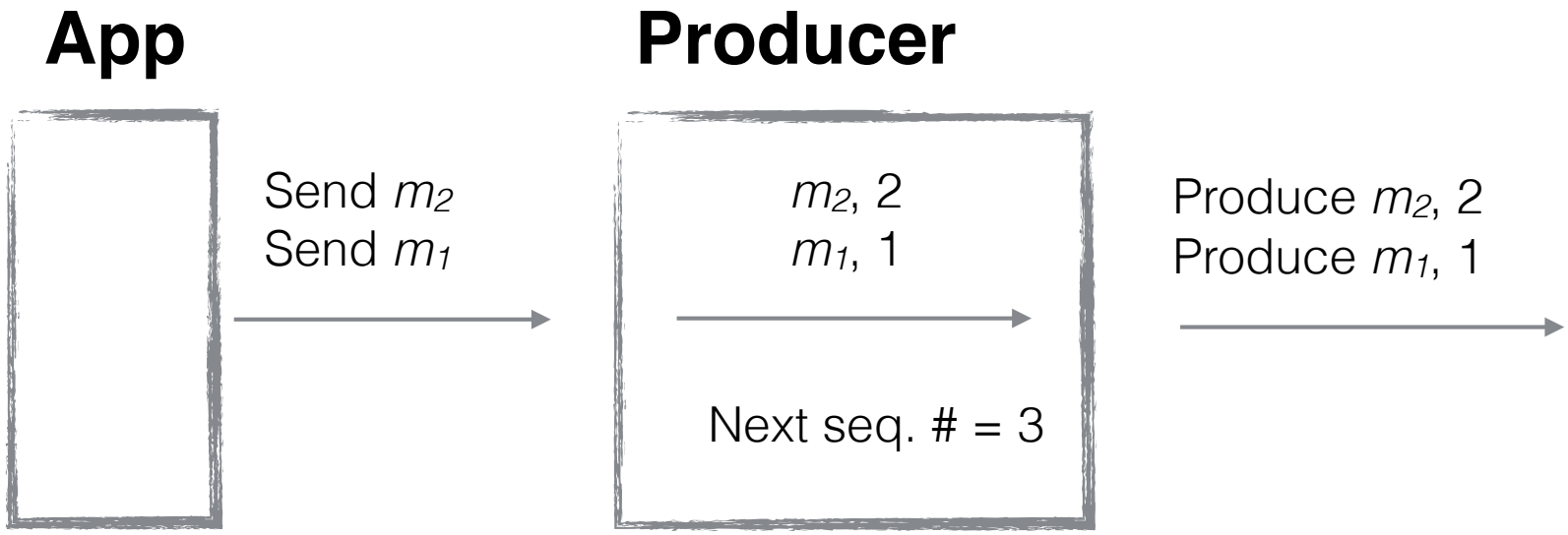
Caveat - Sequence number assignment



Caveat - Sequence number assignment

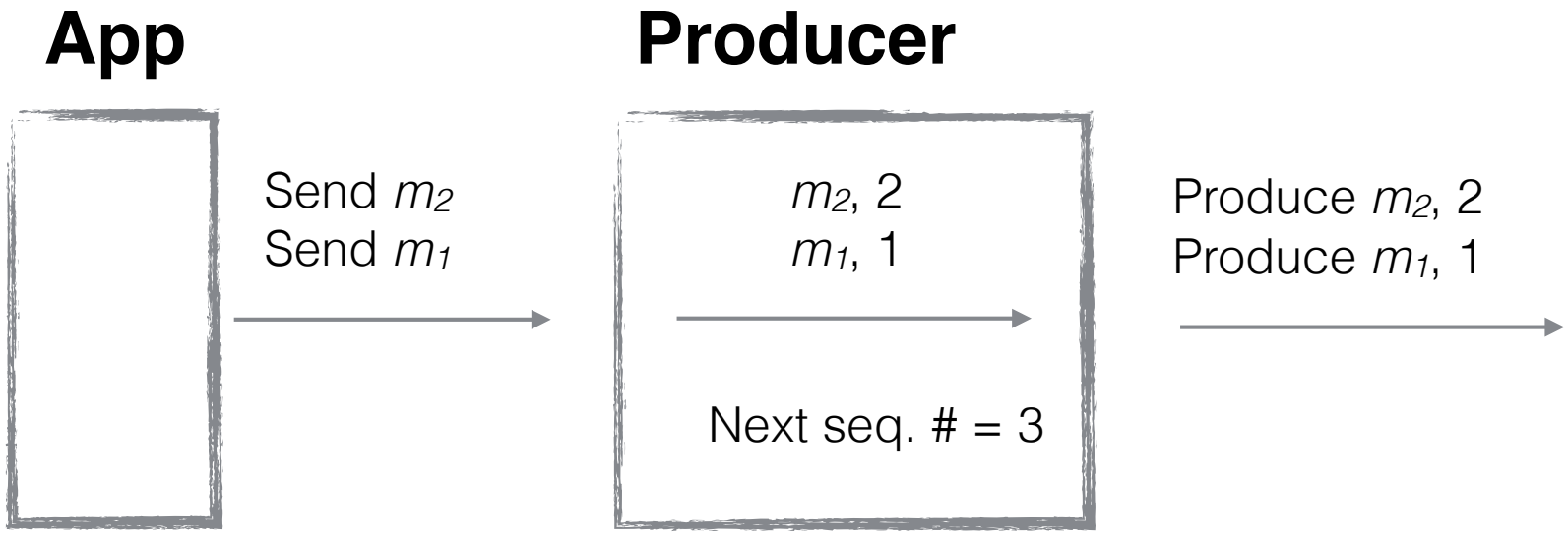


Caveat - Sequence number assignment



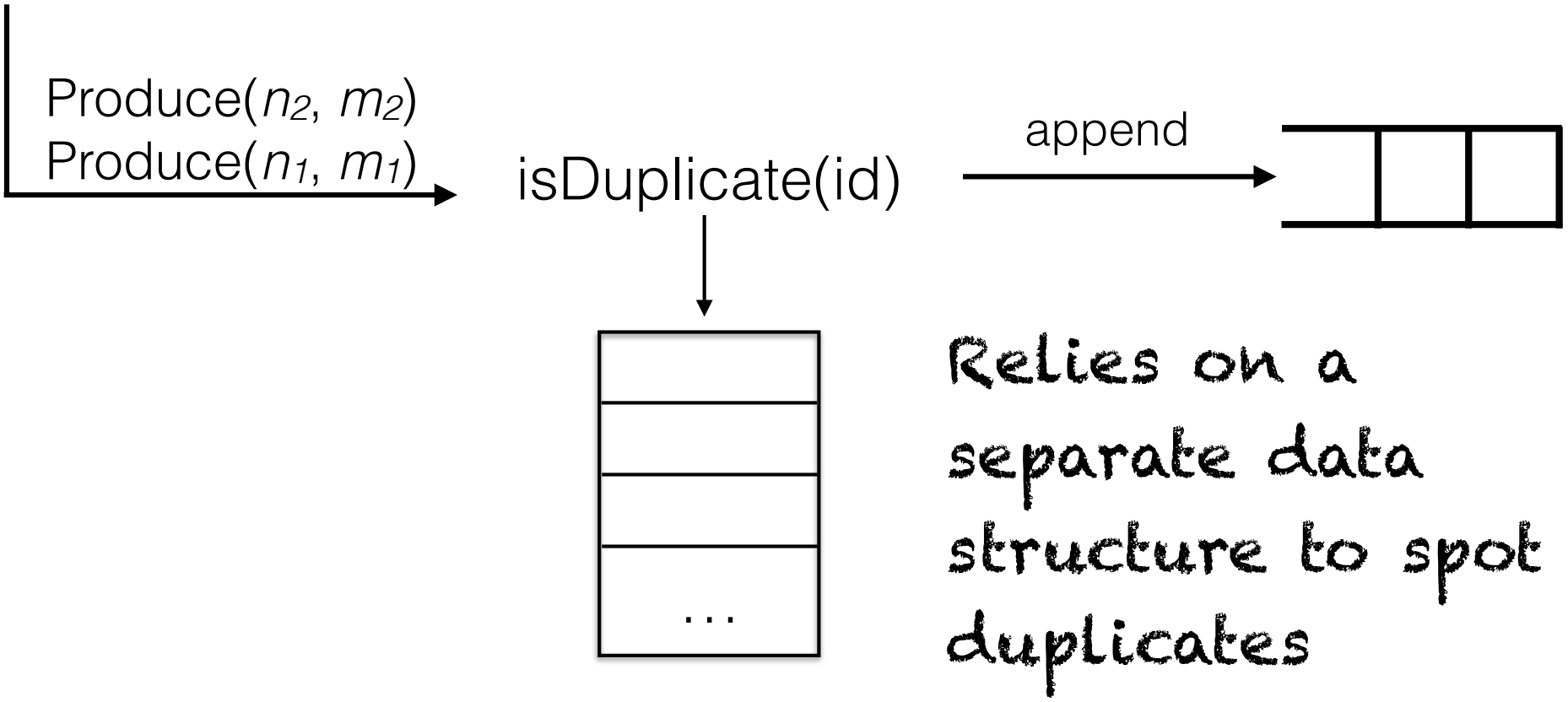
Crash, and upon recovery, must be able to assign the same sequence numbers

Caveat - Sequence number assignment

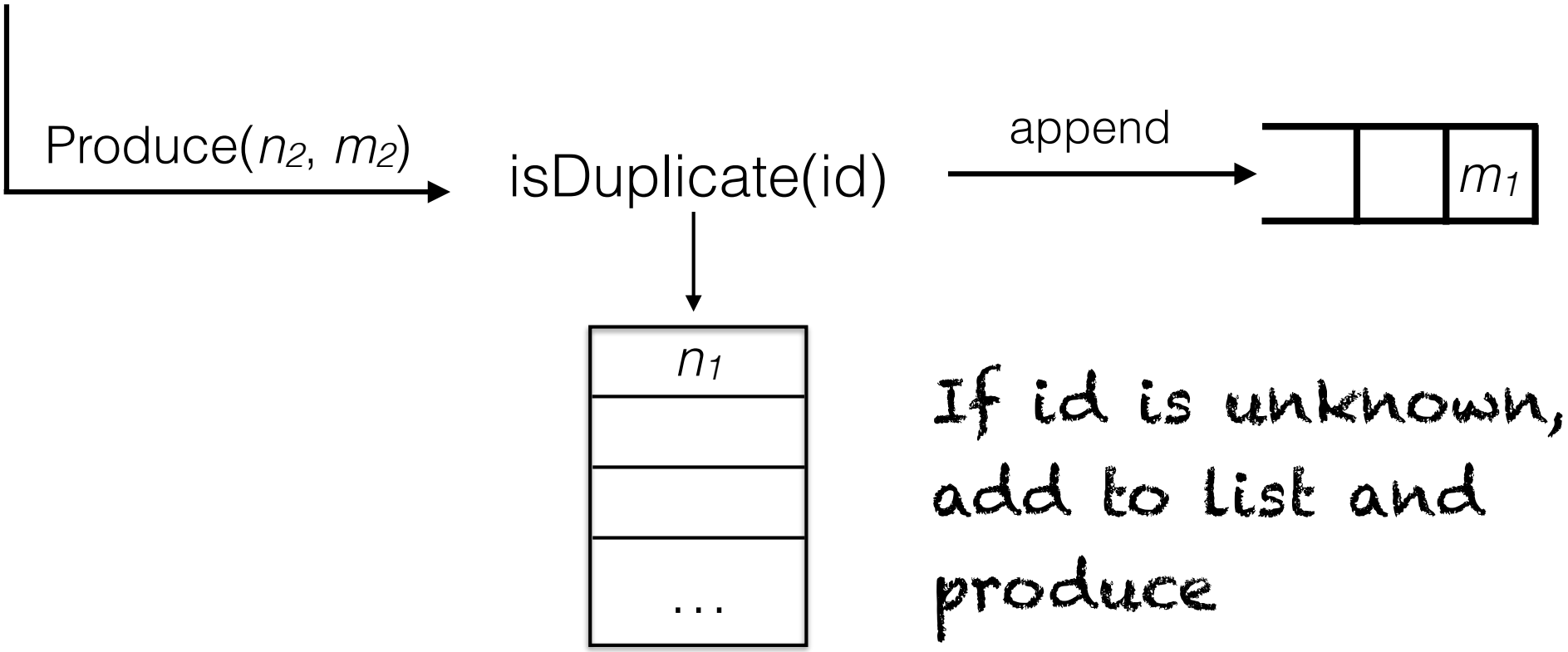


- Two options
  - ✓ Start from scratch (first message)
  - ✓ Persist ⟨source offset, seq. #⟩ pairs

Arbitrary Identifiers

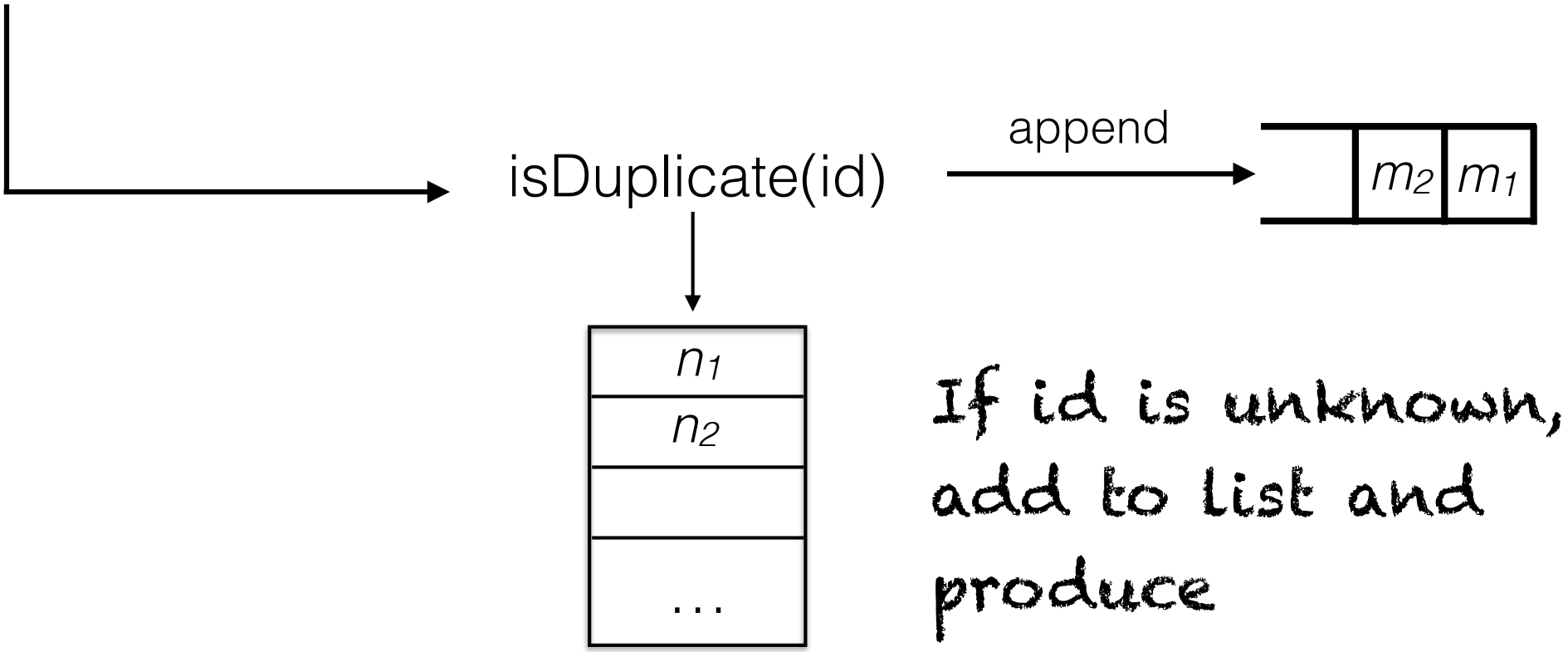


Arbitrary Identifiers



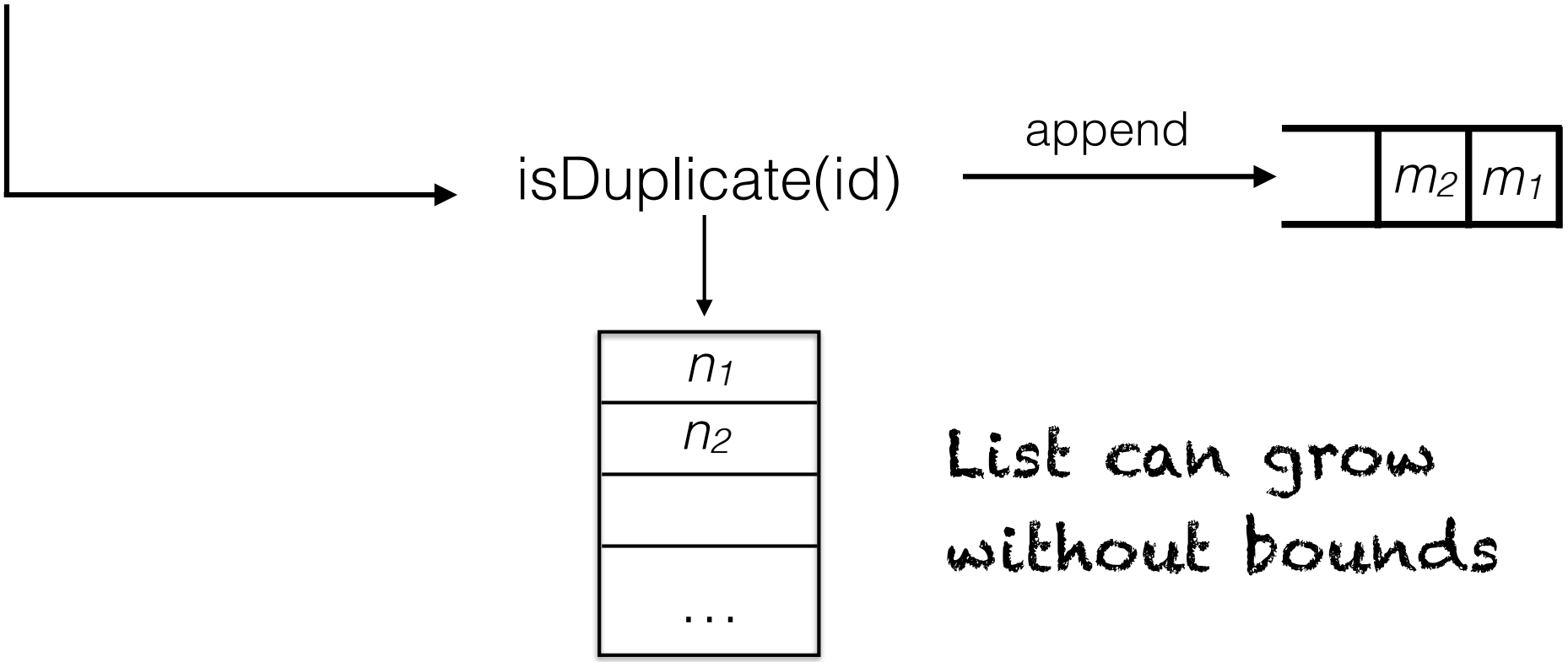
If id is unknown, add to list and produce

Arbitrary Identifiers

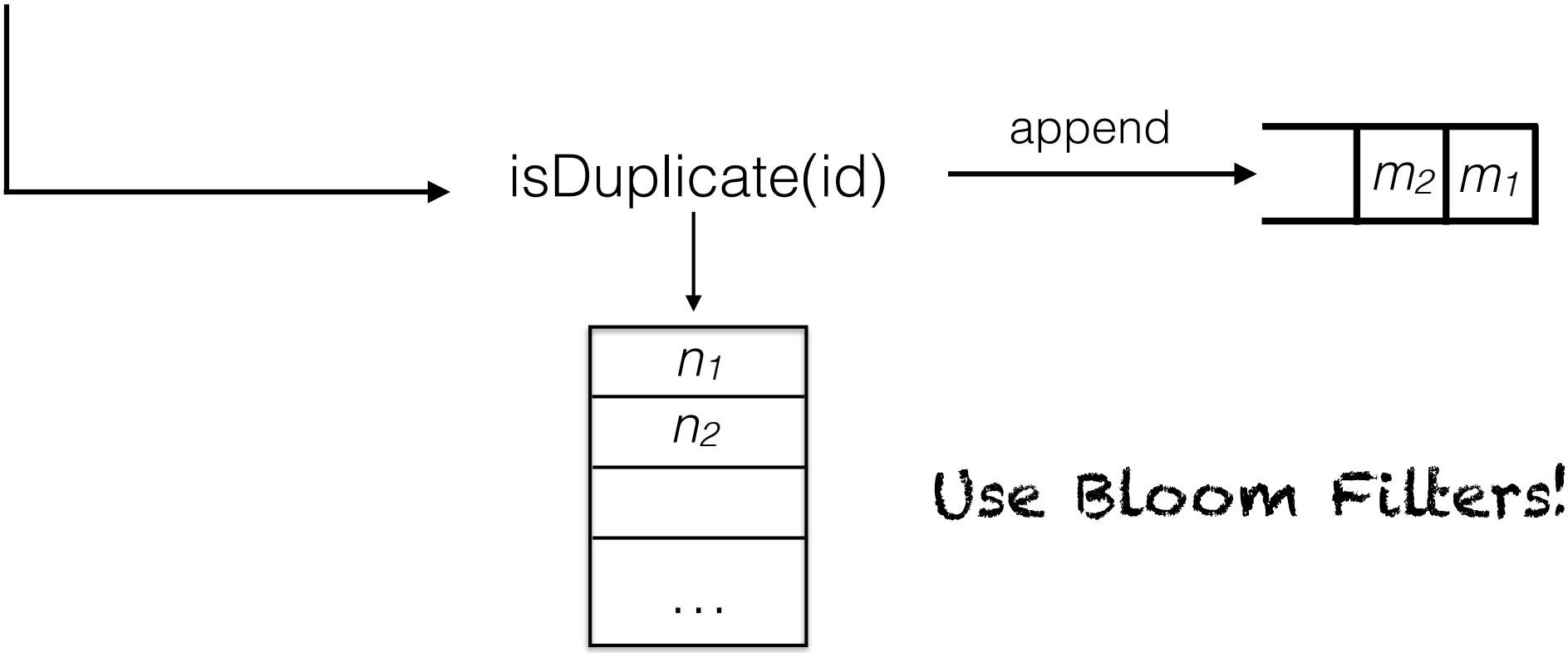


If id is unknown, add to list and produce

Arbitrary Identifiers



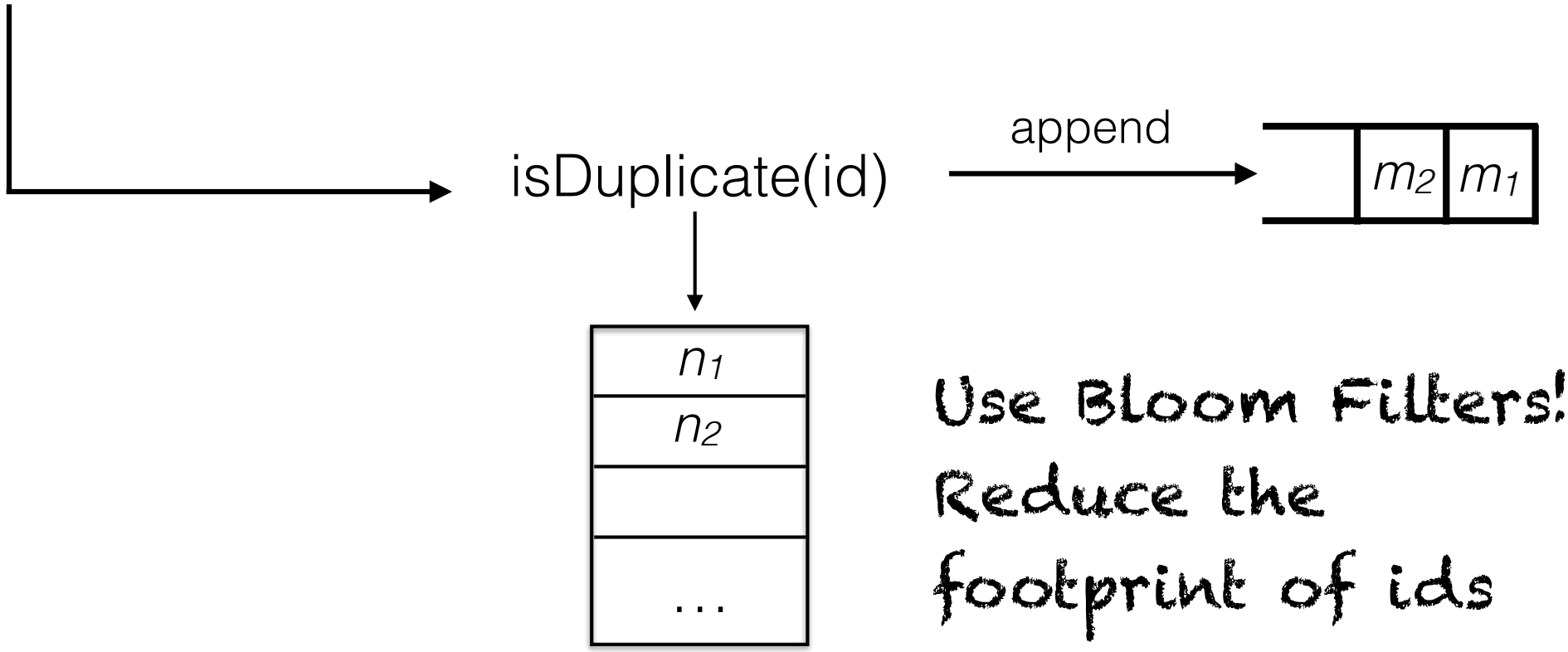
Arbitrary Identifiers



Use Bloom Filters!

Bloom filters!

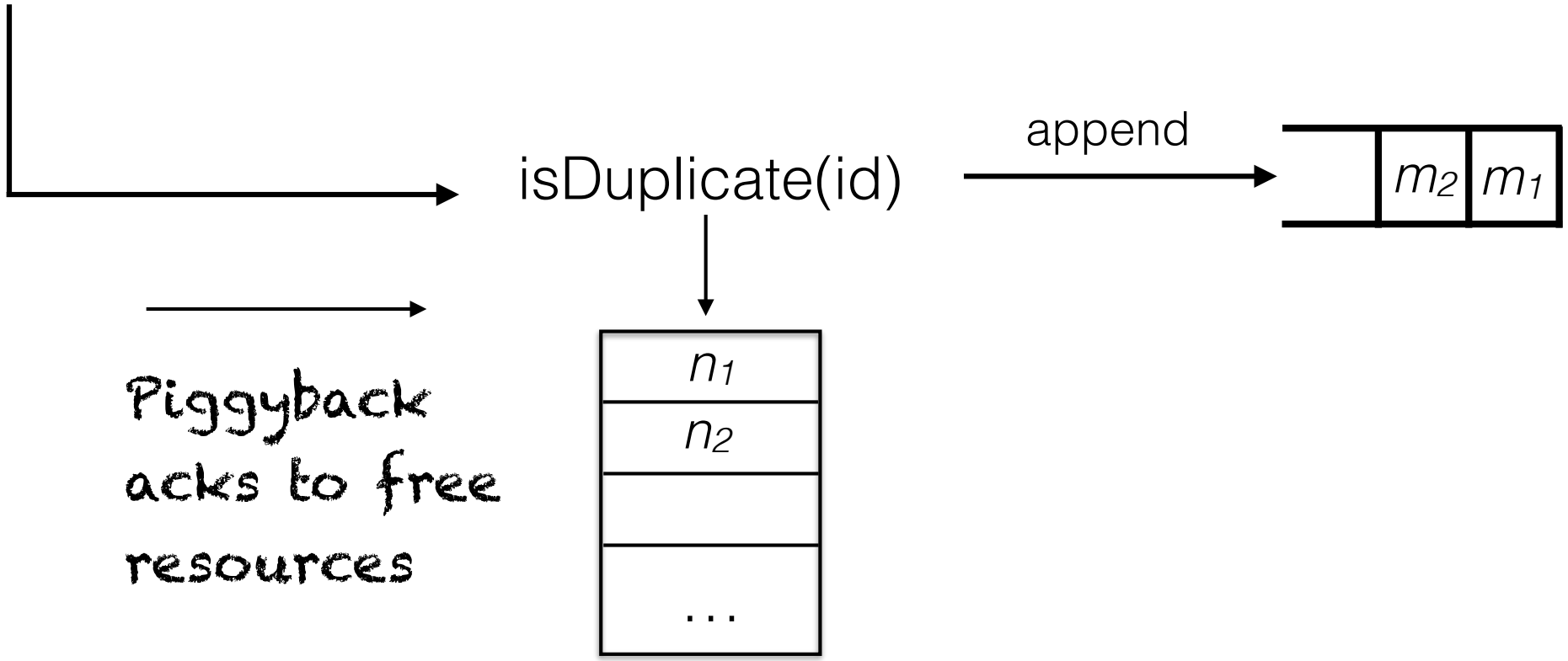
Arbitrary Identifiers



Use Bloom Filters!
Reduce the
footprint of ids

Bloom filters!

Arbitrary Identifiers



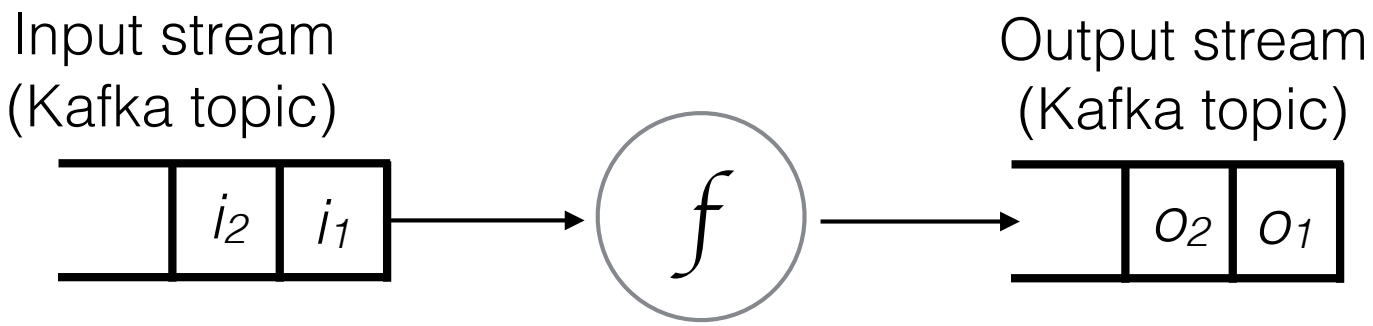
Where?

- Google Millwheel [Tyler Akidau et al., VLDB 2013]
  - ✓ Low-latency data processing
  - ✓ Unique ids and Bloom filters of record fingerprints
  - ✓ Bigtable as a backend
- Google Photon [R. Ananthanarayanan et al., SIGMOD 2013]
  - ✓ Event id: server IP + process id + timestamp
  - ✓ Multi-DC Paxos DB for deduplication

Let’s assume sequence numbers for the next part

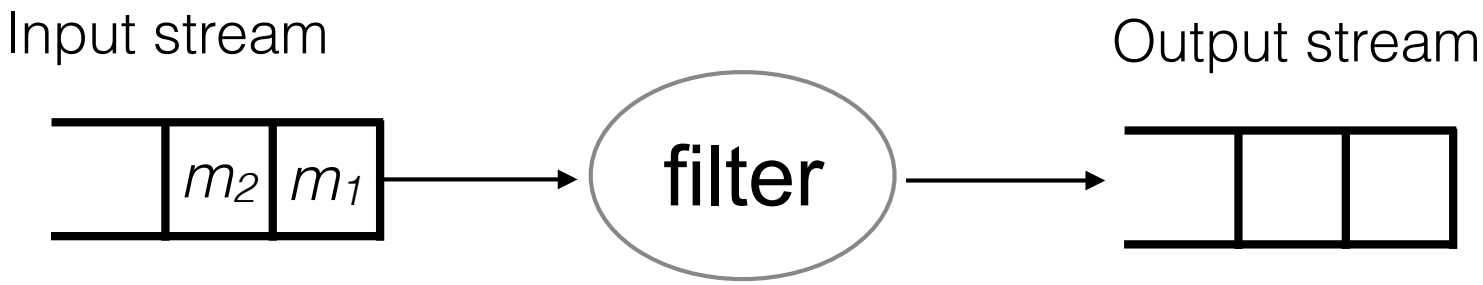
Exactly-once for Streams

Abstracting streams



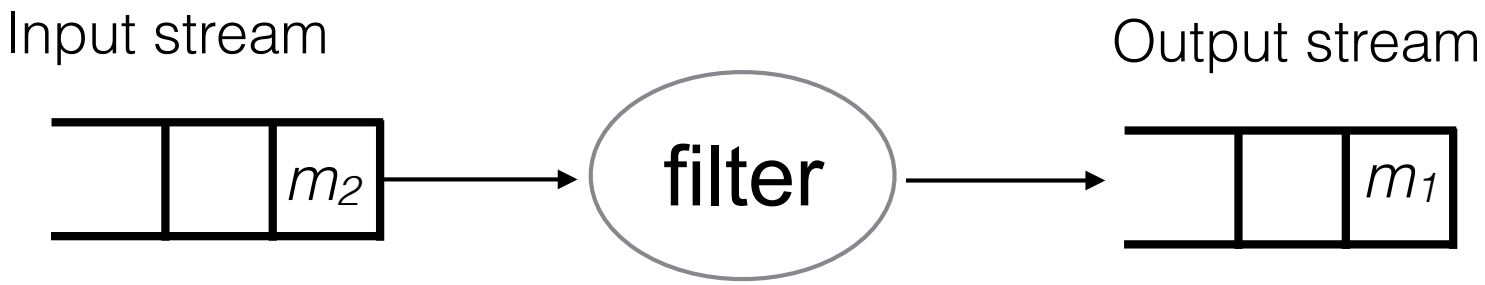
Function f transforms the messages of the input stream (e.g., filter)

Simple streams example



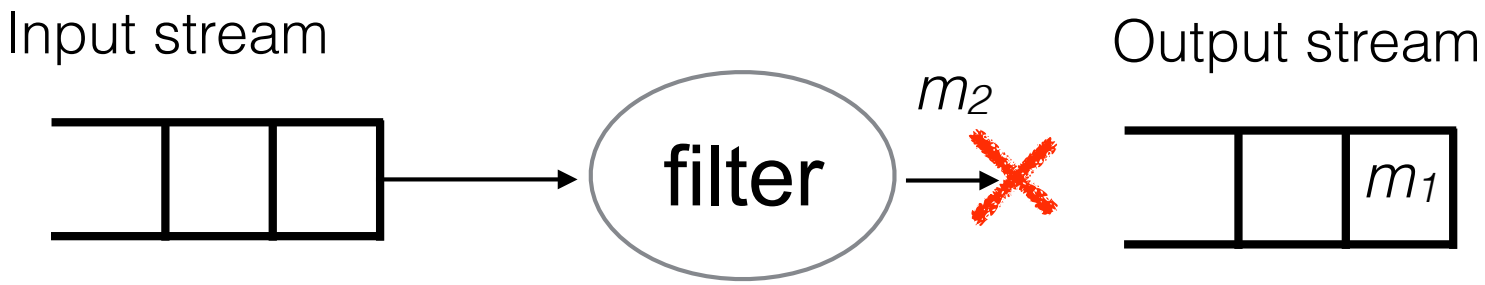
Say we are simply filtering

Simple streams example



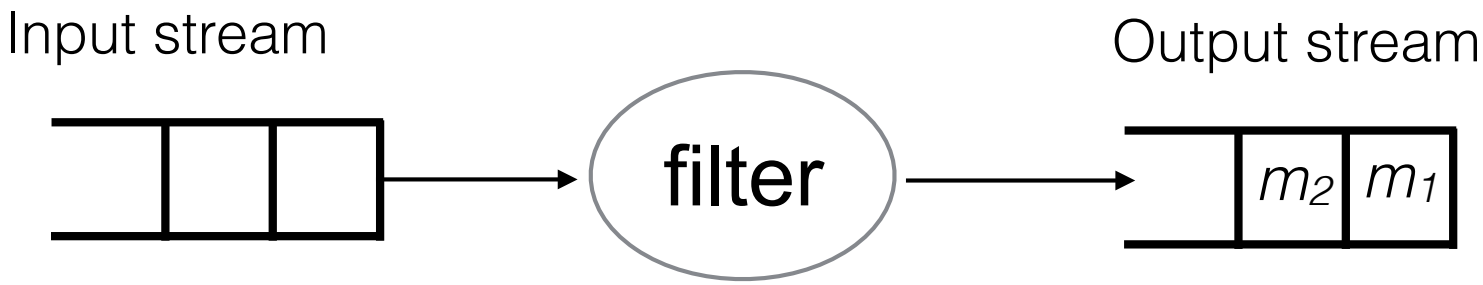
- Message m_1 passes the filter
- Produced to the output stream

Simple streams example



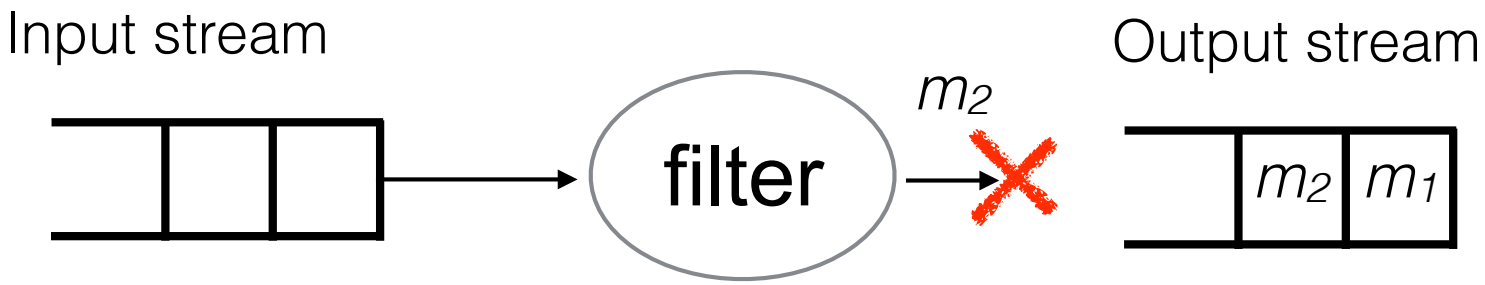
- Message m_2 passes the filter
- Request fails, retry

Simple streams example



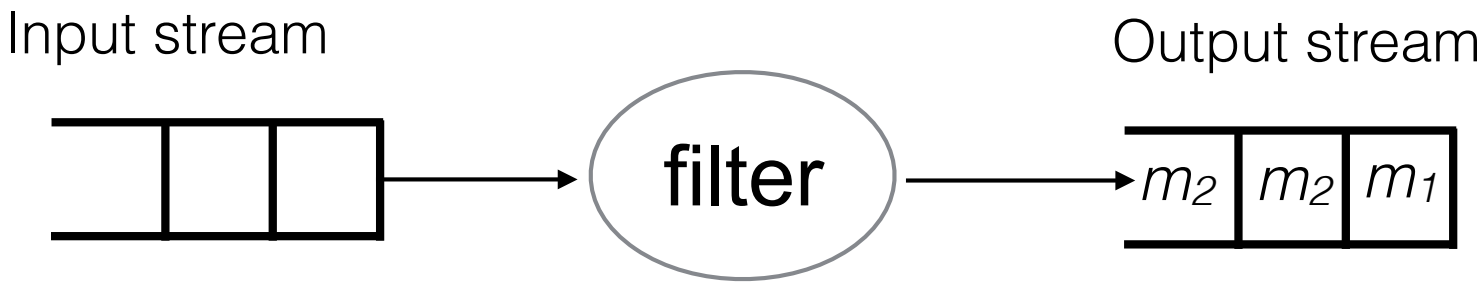
- Message m_2 passes the filter
- Request fails, retry

Simple streams example



- Message m_2 passes the filter
- Request fails, retry

Simple streams example

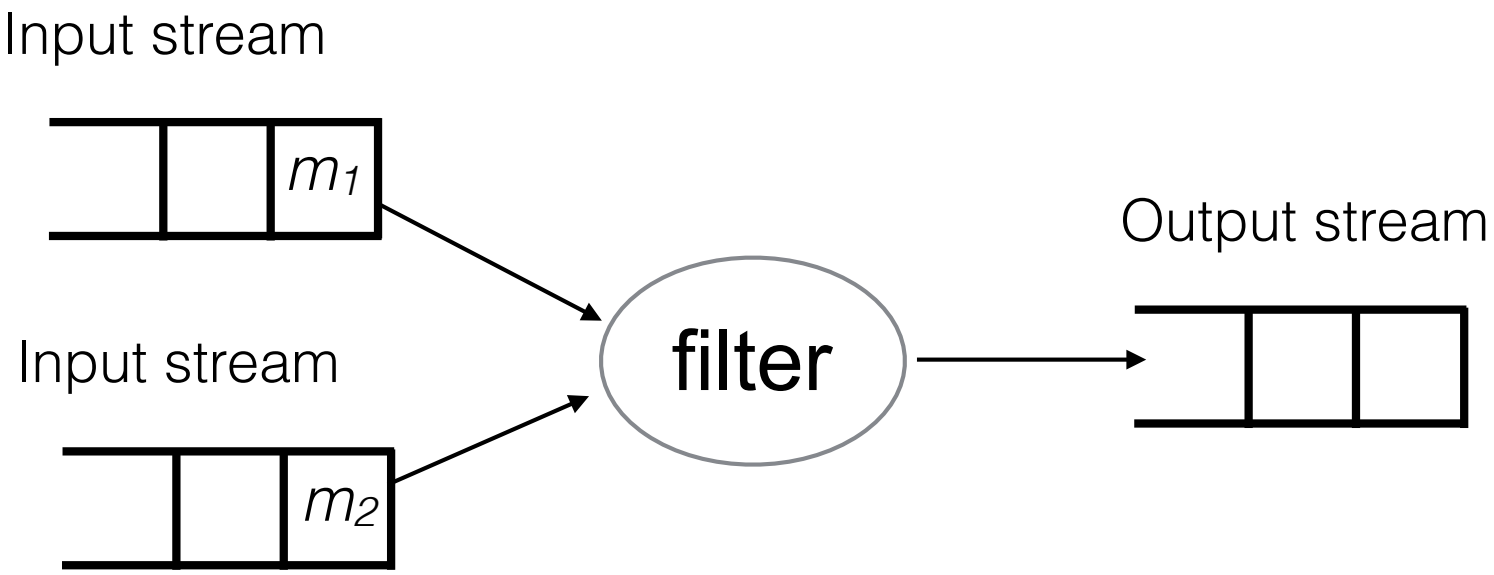


- Message m_2 passes the filter
- Requires deduplication!

Deduplication isn’t always sufficient

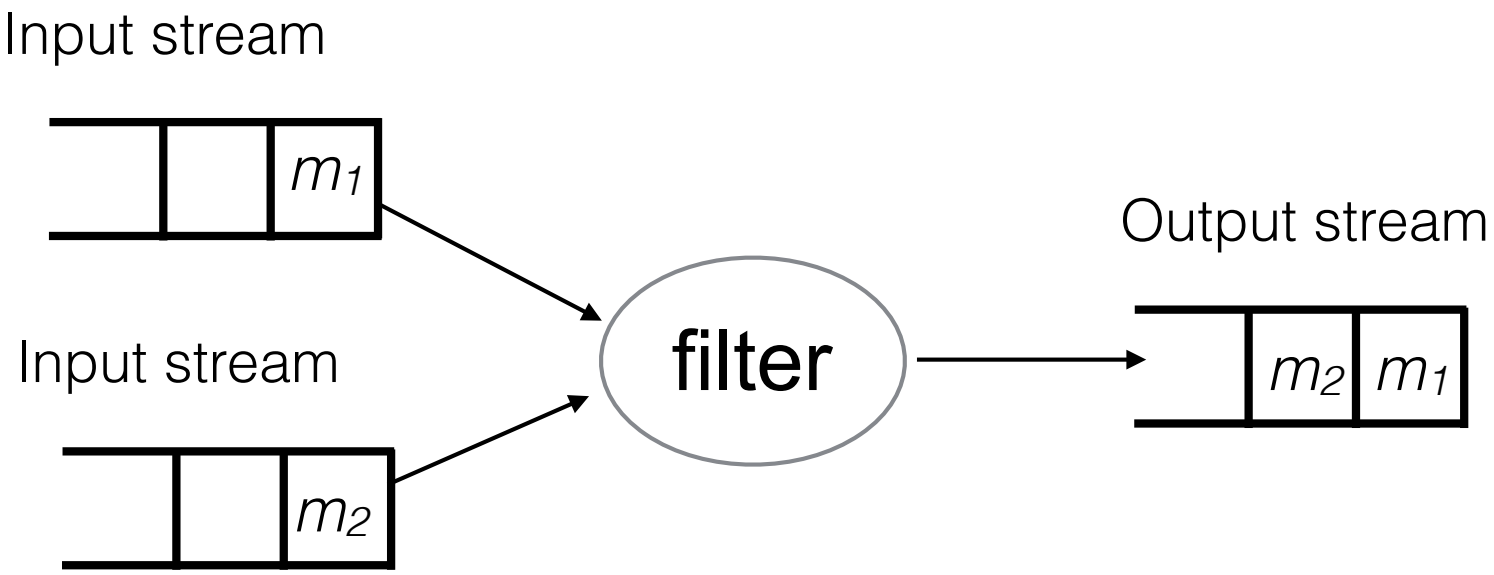
…because of non-determinism

Two input sources



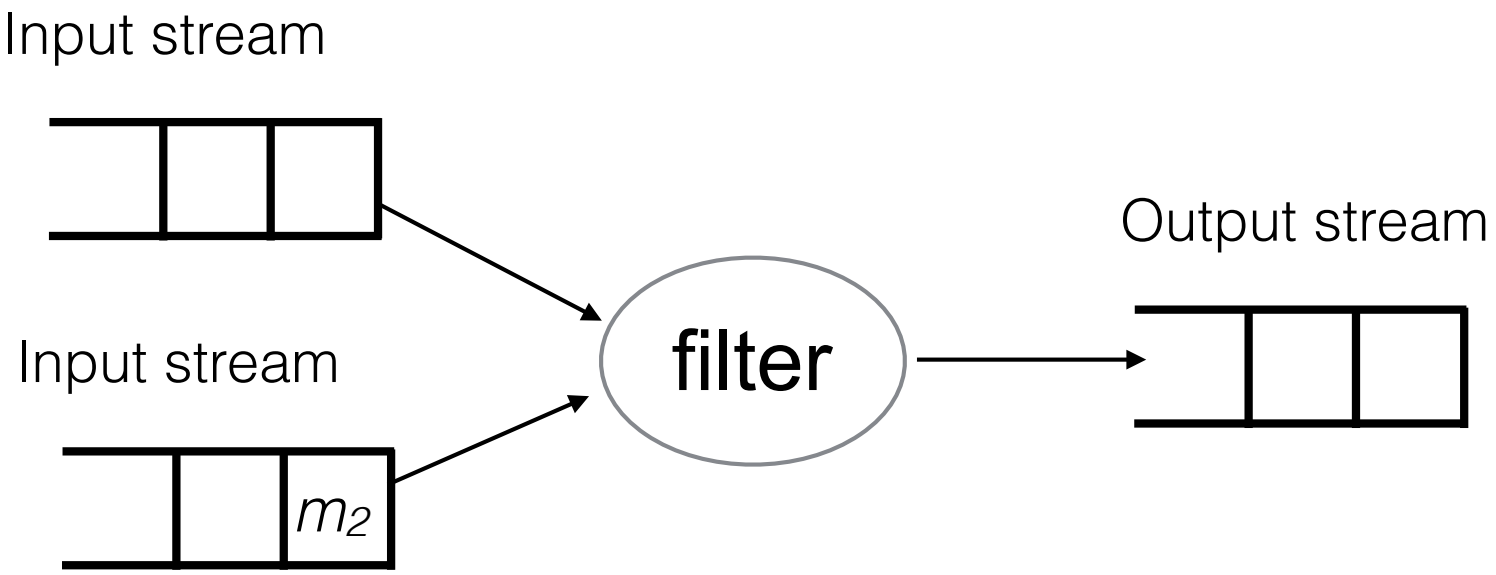
- Two input streams
- Two input Kafka partitions

Two input sources



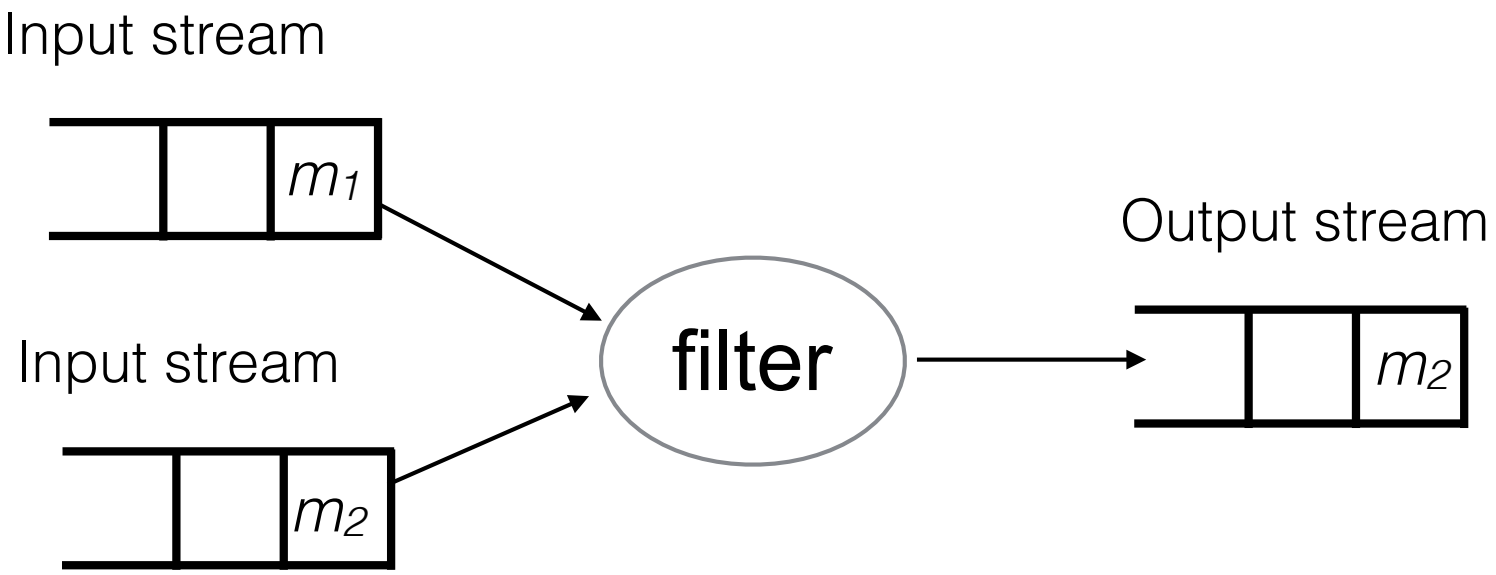
Output order depends on how the input is scanned

Two input sources



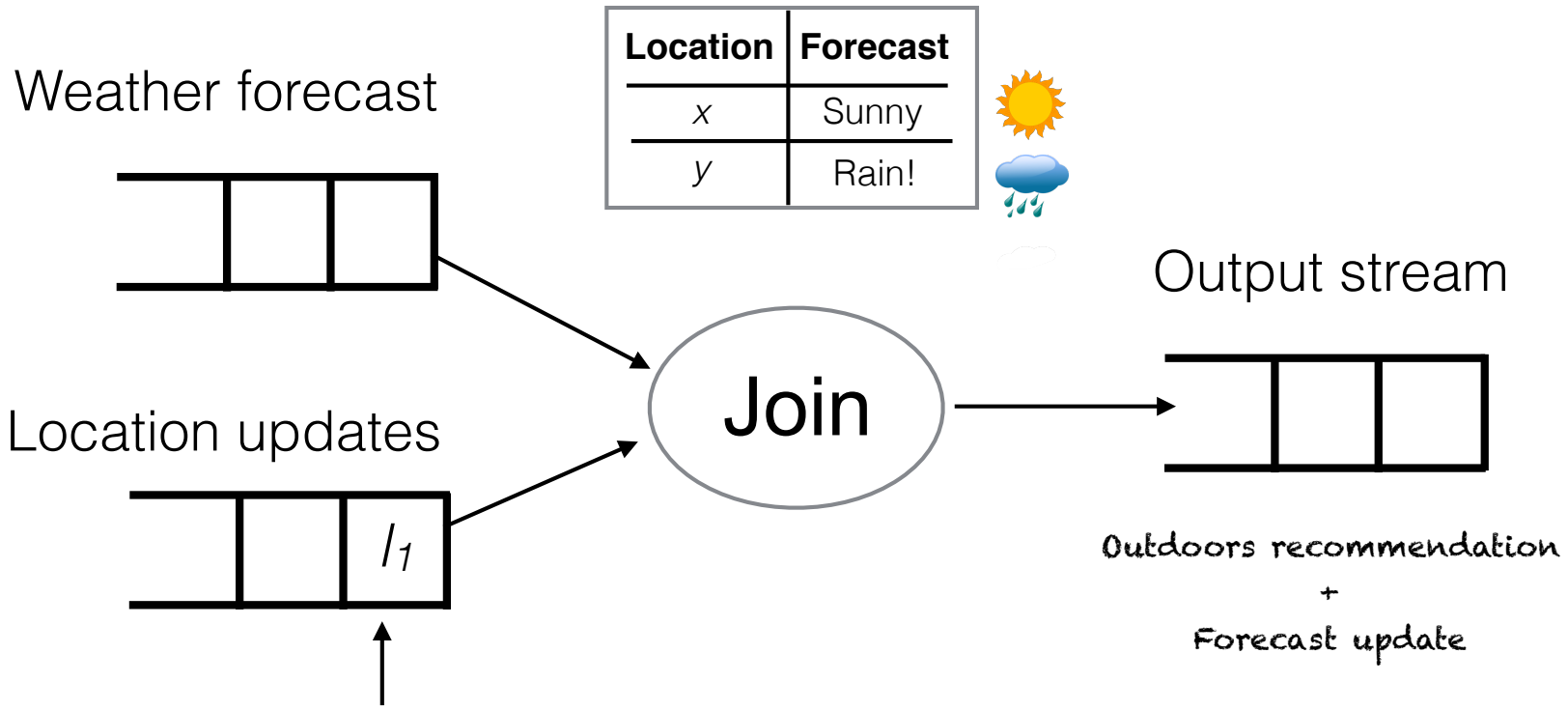
Even if scan order is deterministic, timing is important

Two input sources

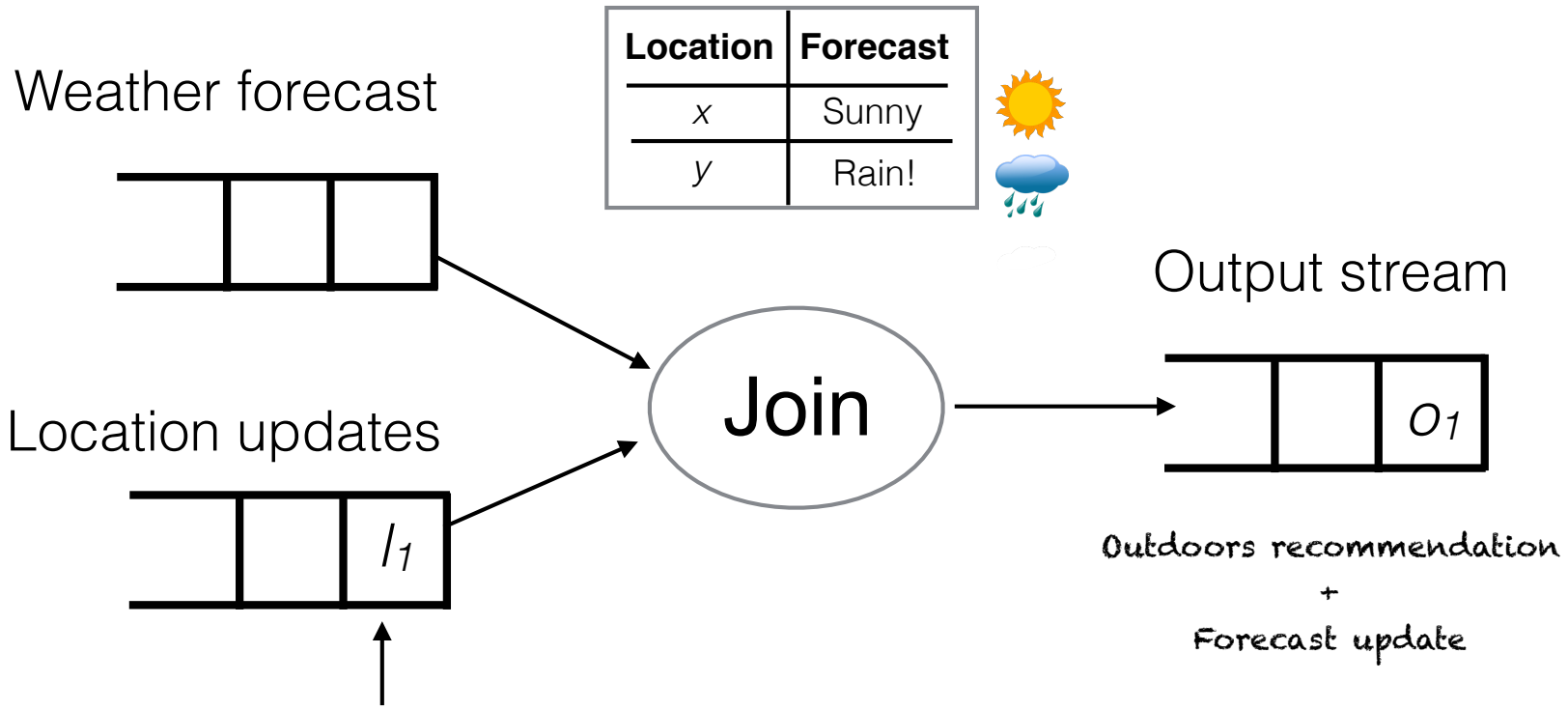


Even if scan order is deterministic, timing is important

Recommendations



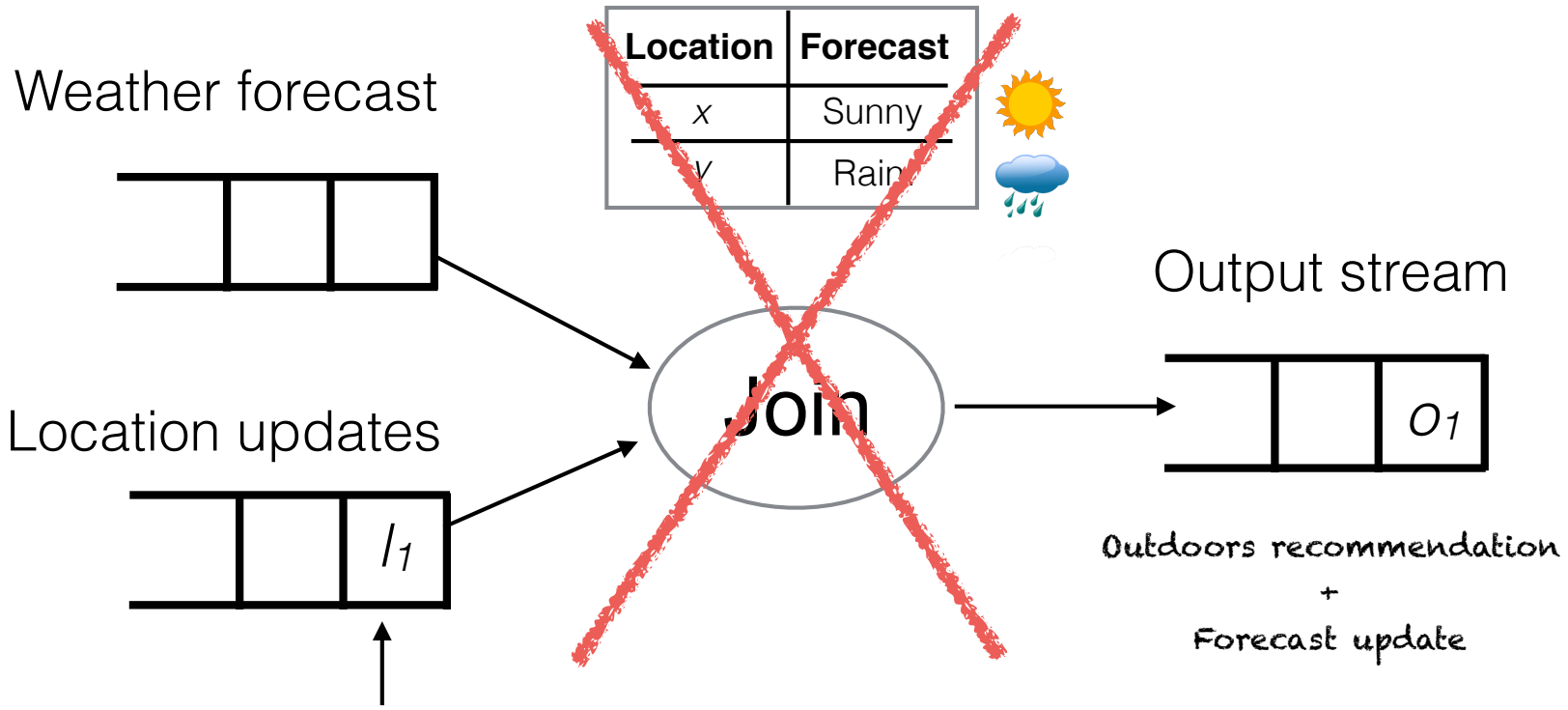
Recommendations



l_1 : Device d, Location x

o_1: great weather for outdoors

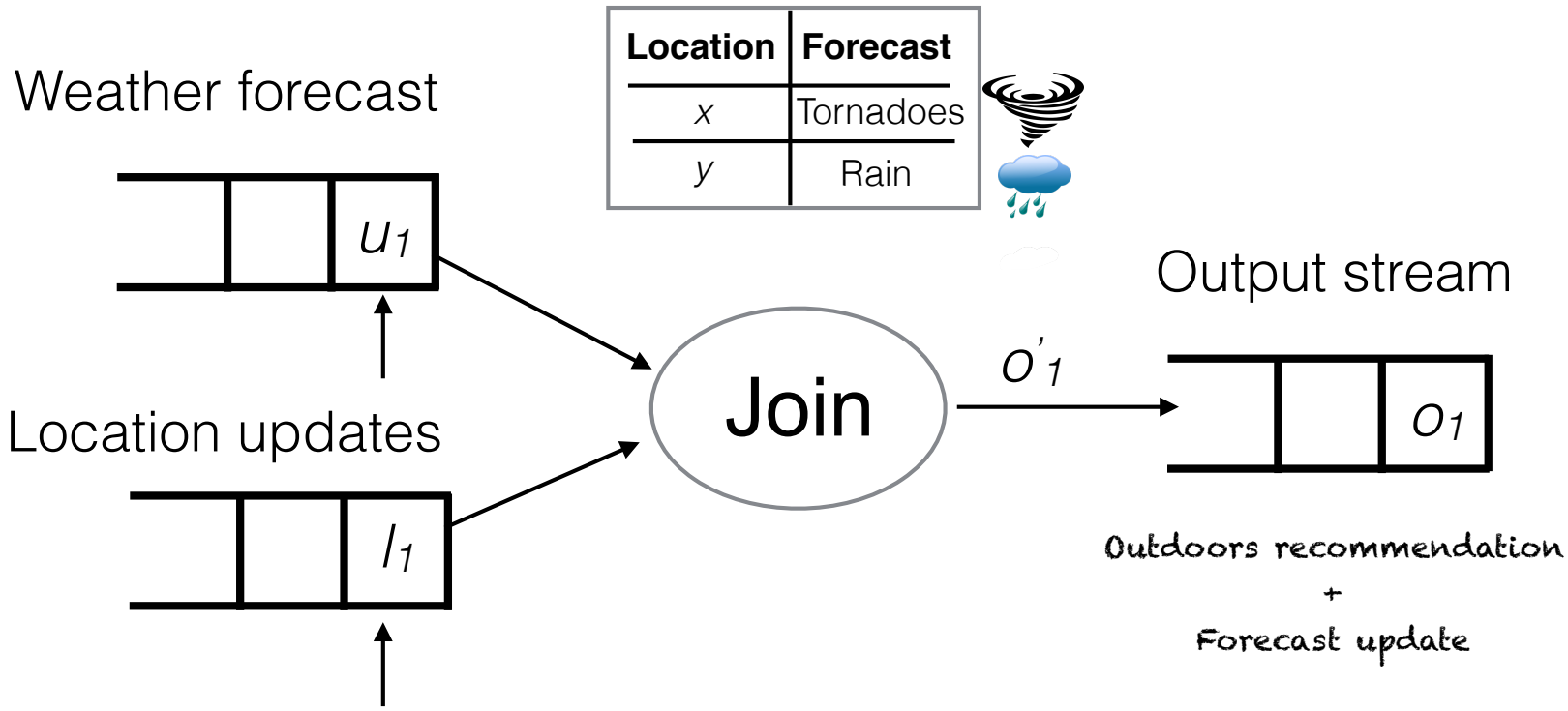
Recommendations



l_1 : Device d, Location x

o_1: great weather for outdoors

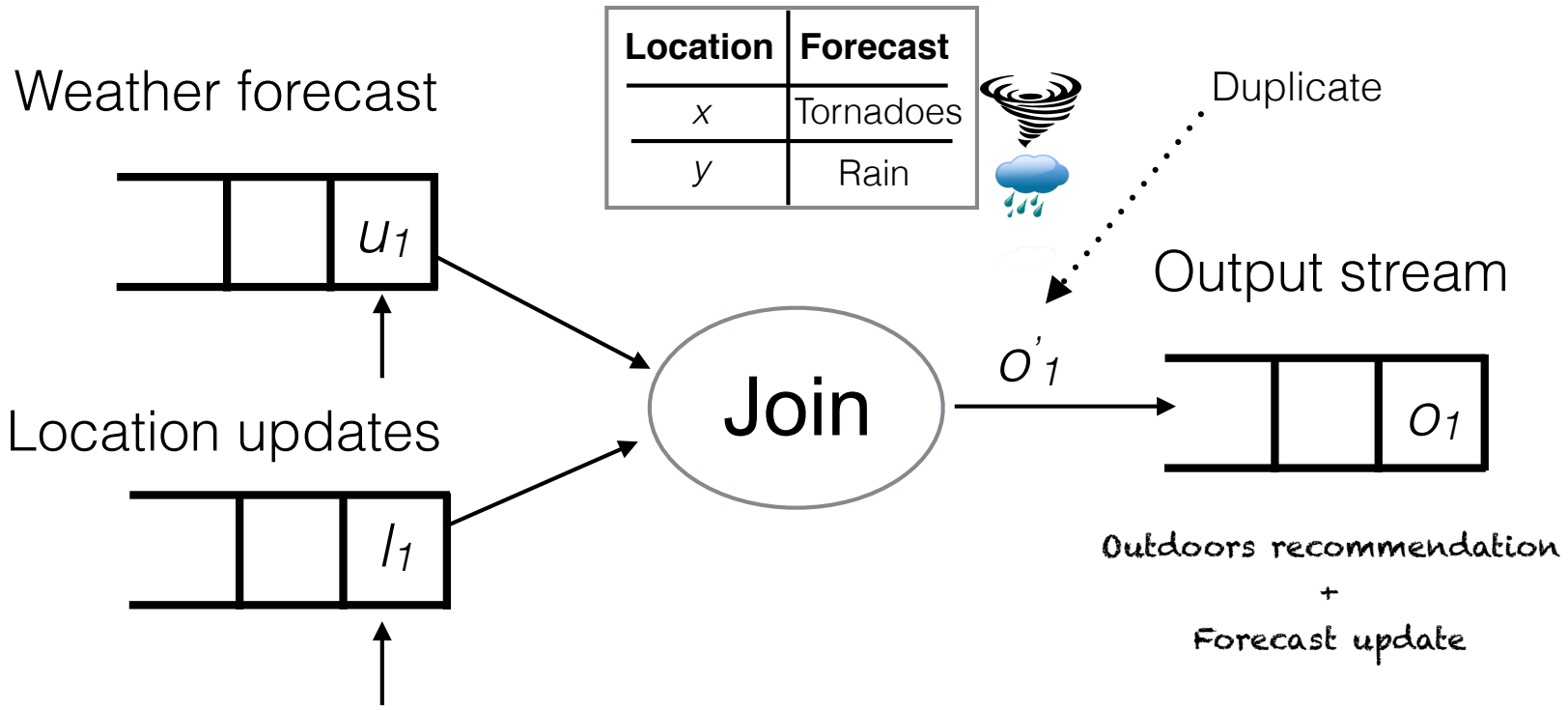
Recommendations



l_1 : Device d, Location x
u_1 : Location x, Tornadoes

o_1: great weather for outdoors
o'_1: warning!

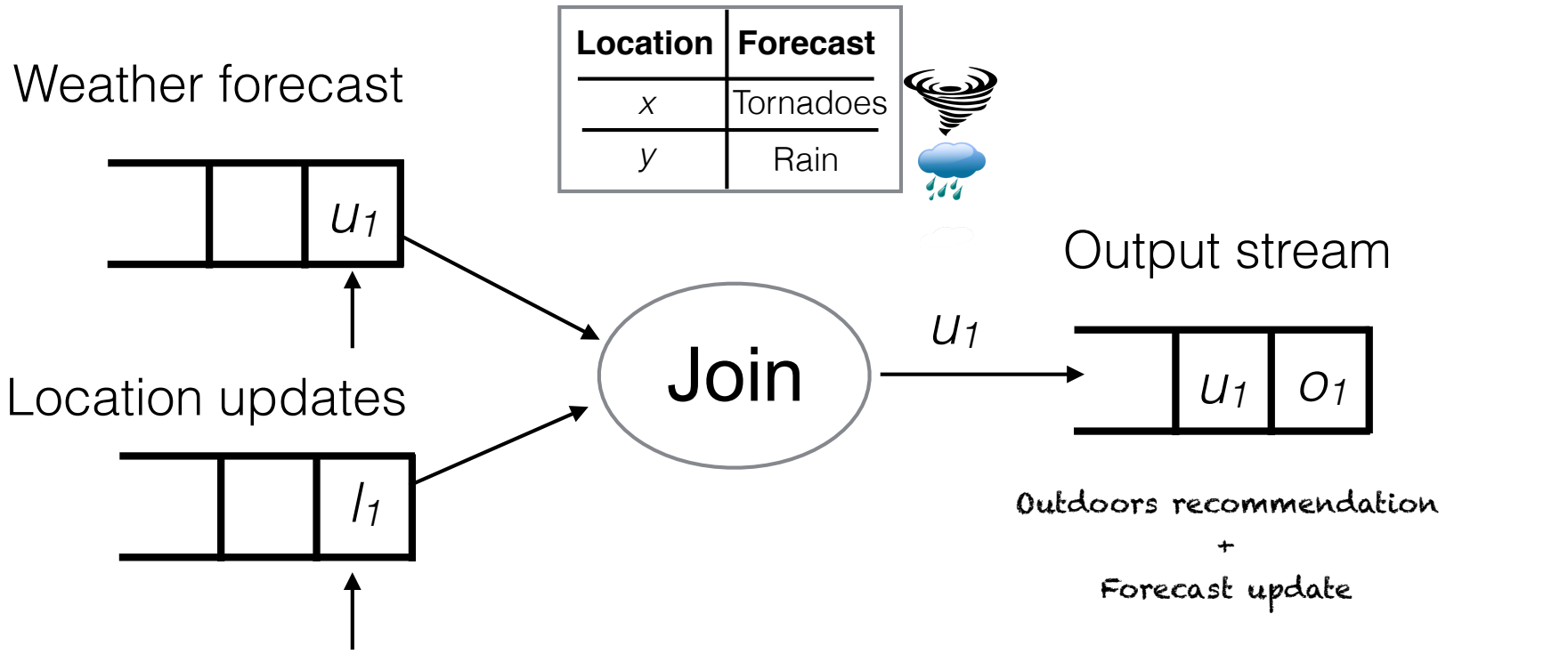
Recommendations



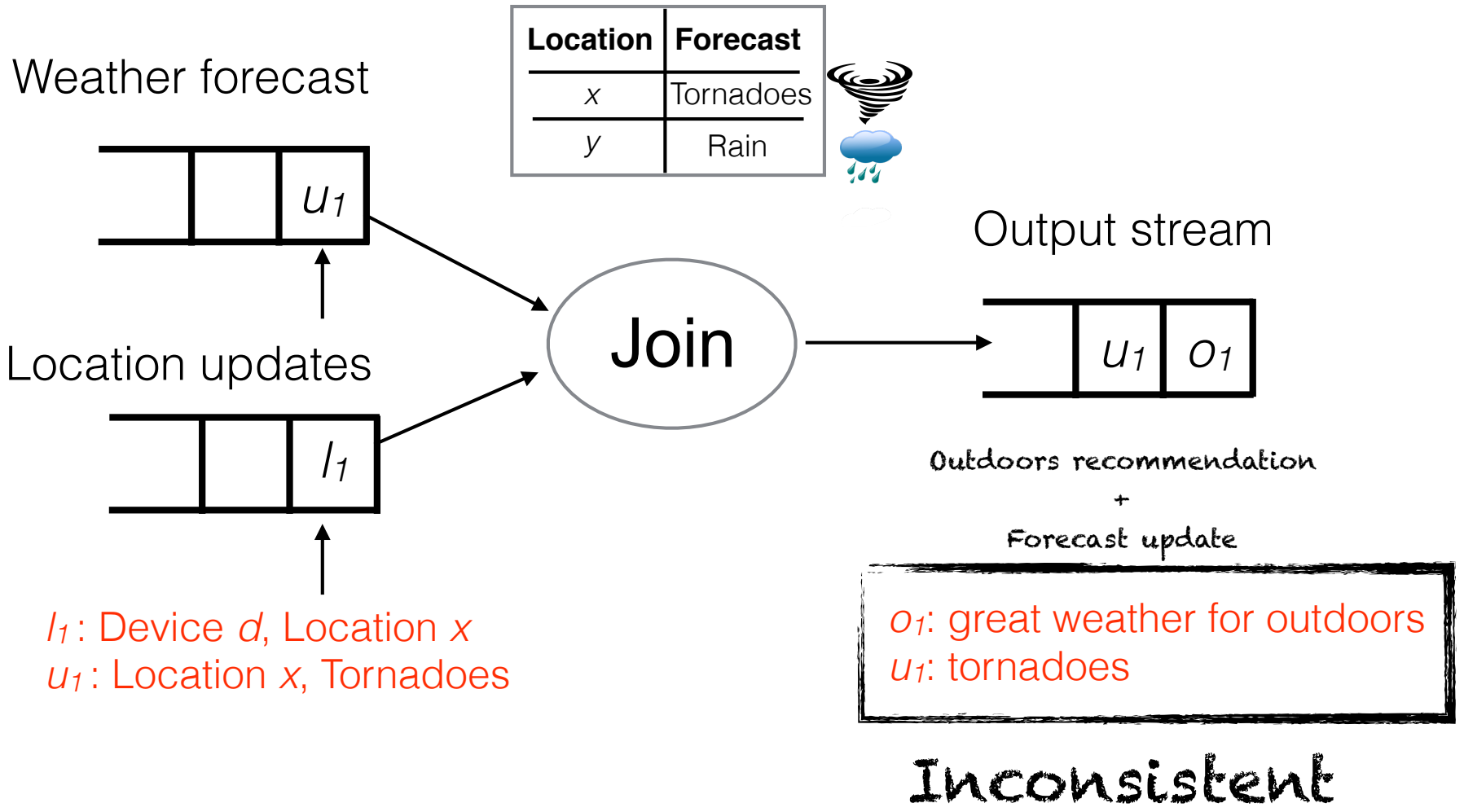
l_1 : Device d, Location x
u_1 : Location x, Tornadoes

o_1: great weather for outdoors
o'_1: warning!

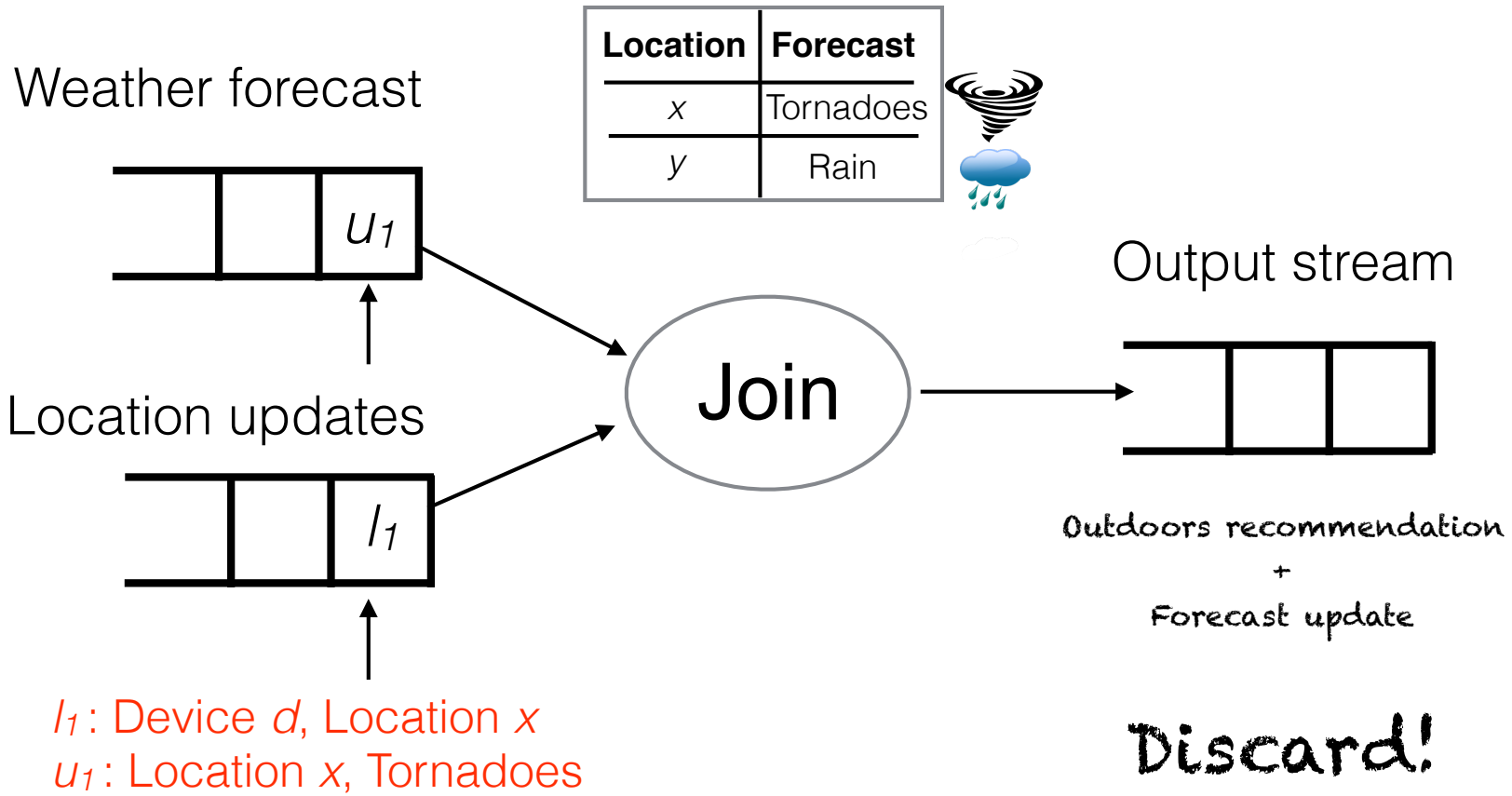
Recommendations



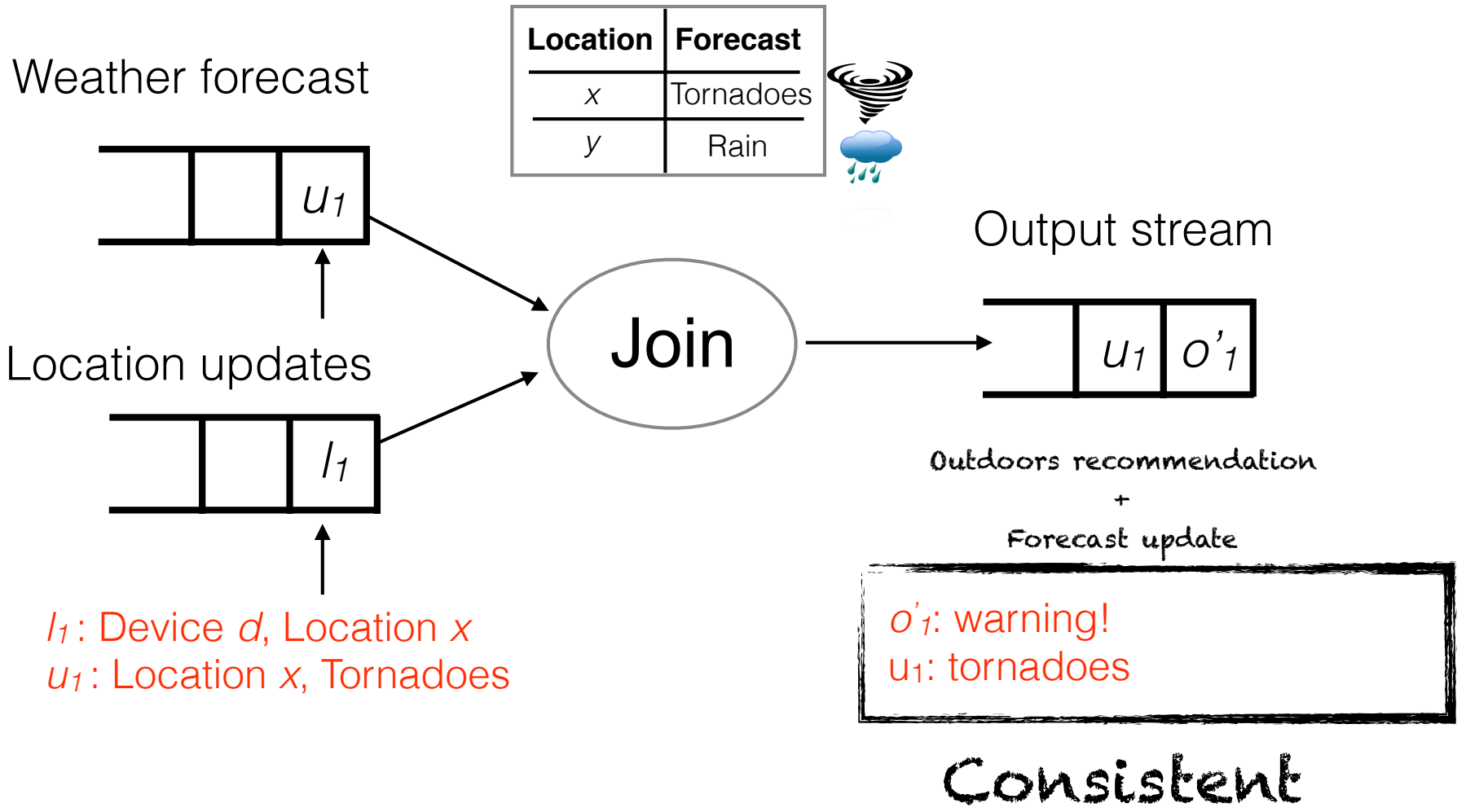
Recommendations



Recommendations



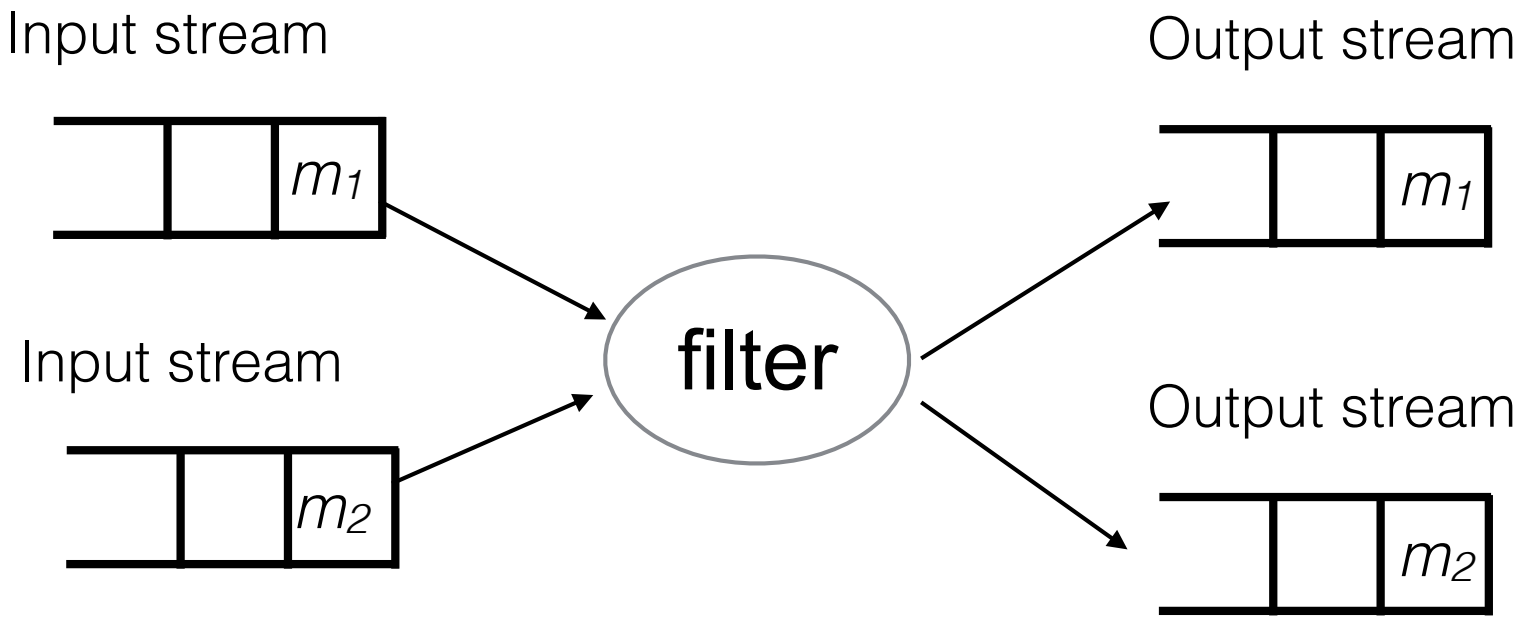
Recommendations



Discarding messages

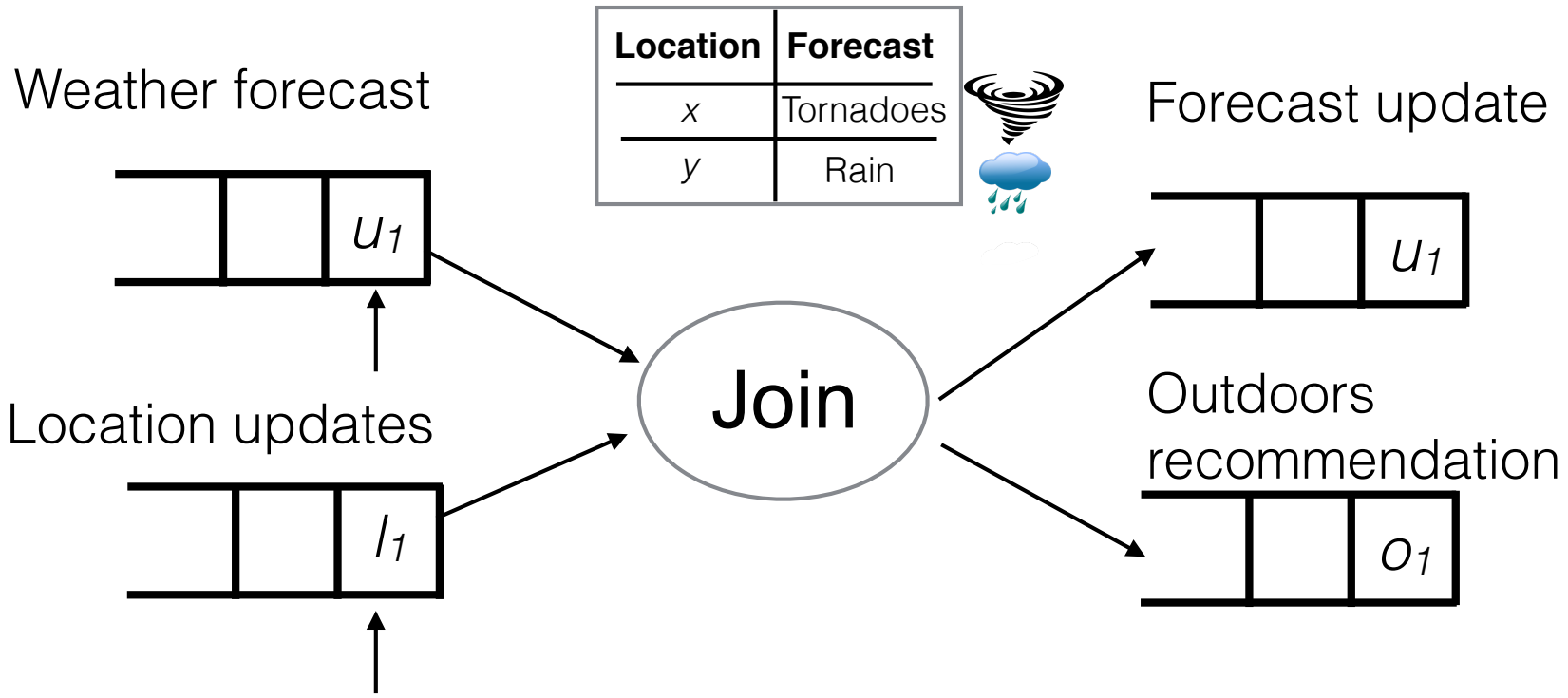
- Downstream
  - Separate tentative from stable
  - Use special marker messages
- Markers
  - **Stable**: messages are ready to be consumed
  - **Discard**: discard the last batch of messages

Two input sources
Two output partitions



- Multiple output partitions
- Output is sharded by key

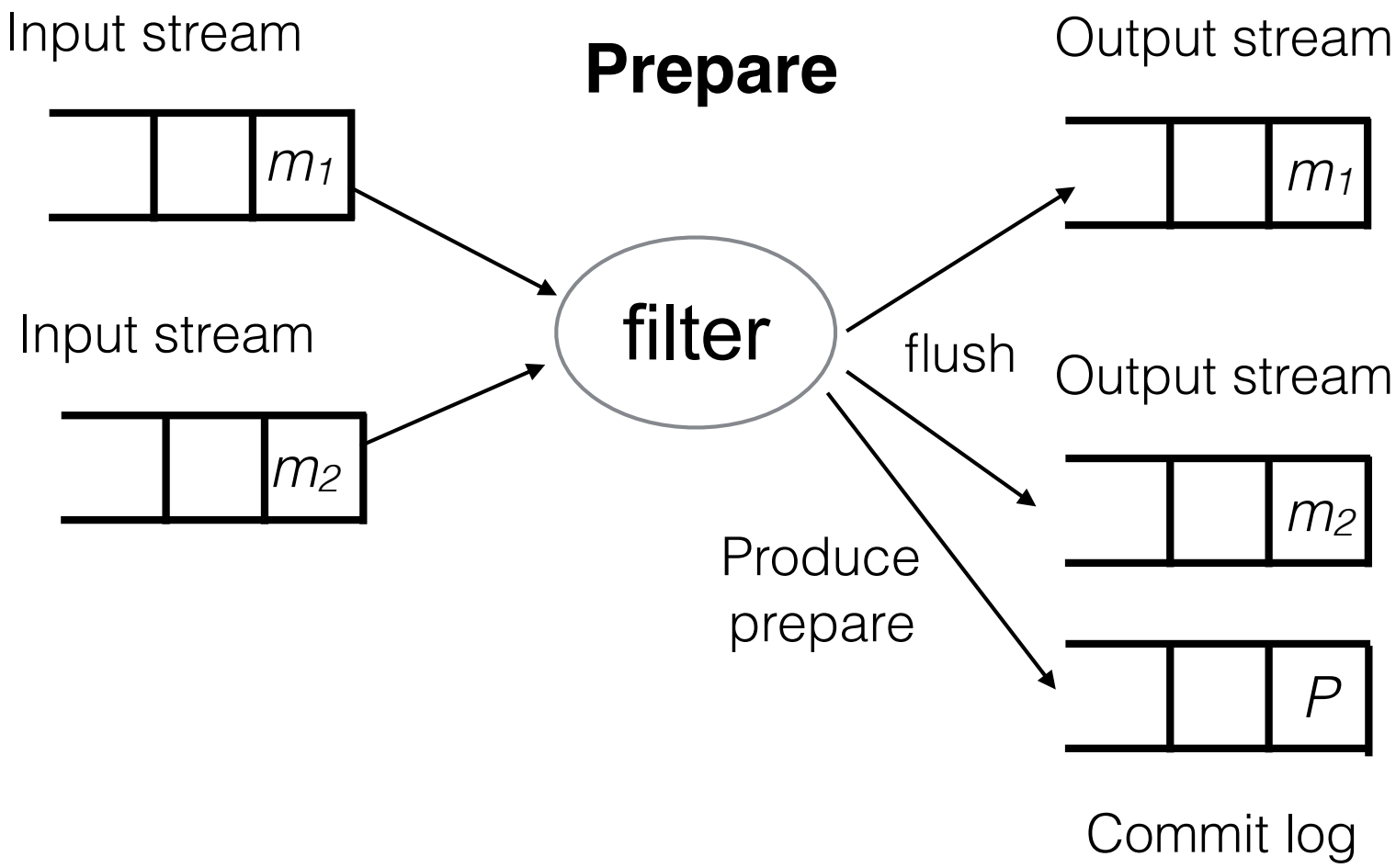
Recommendations



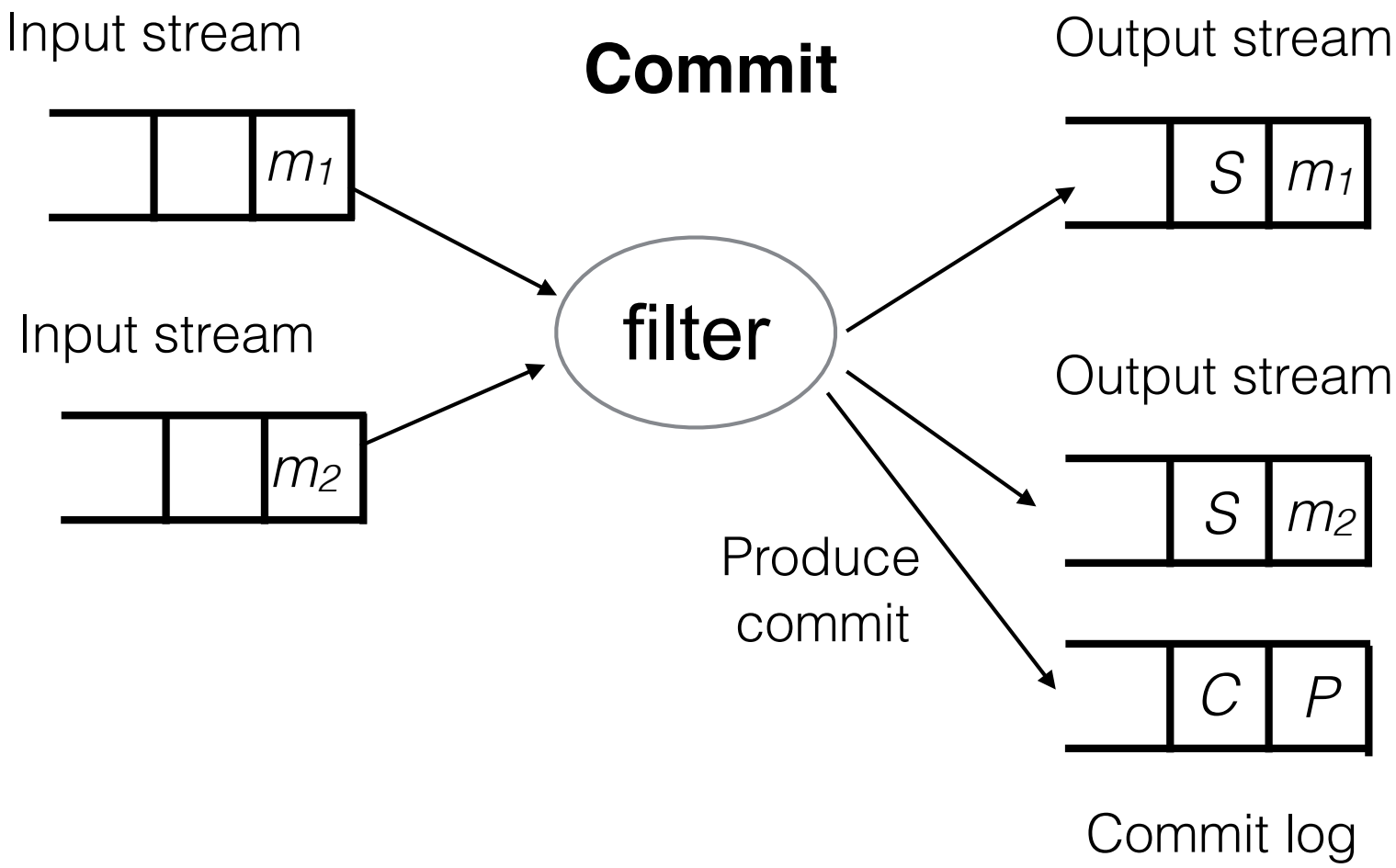
Transactional

- Batch of output messages
- All partitions are made stable or none
- Use a commit protocol to guarantee *atomicity*
- ***Atomicity****: Eventually all tentative messages are made either stable or discarded.*

Two input sources

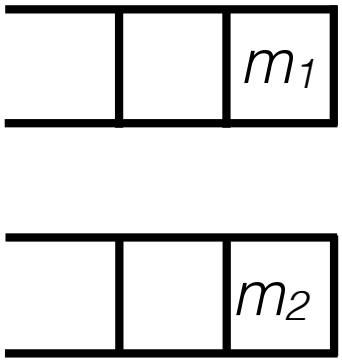


Two input sources



Wrap-up

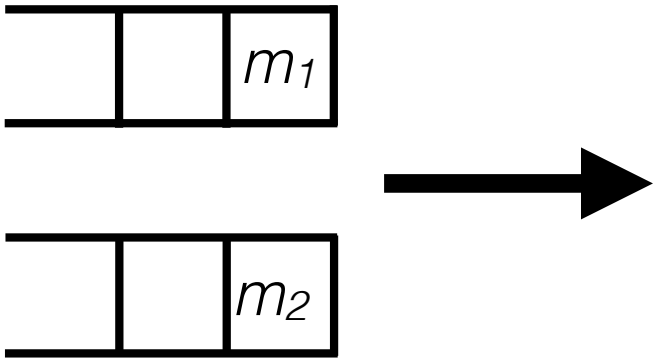
Consuming



Apache Flink
Apache Spark
Kafka Connectors

Exactly-onced
Kafka Topics

Consuming



Apache Flink
Apache Spark
Kafka Connectors

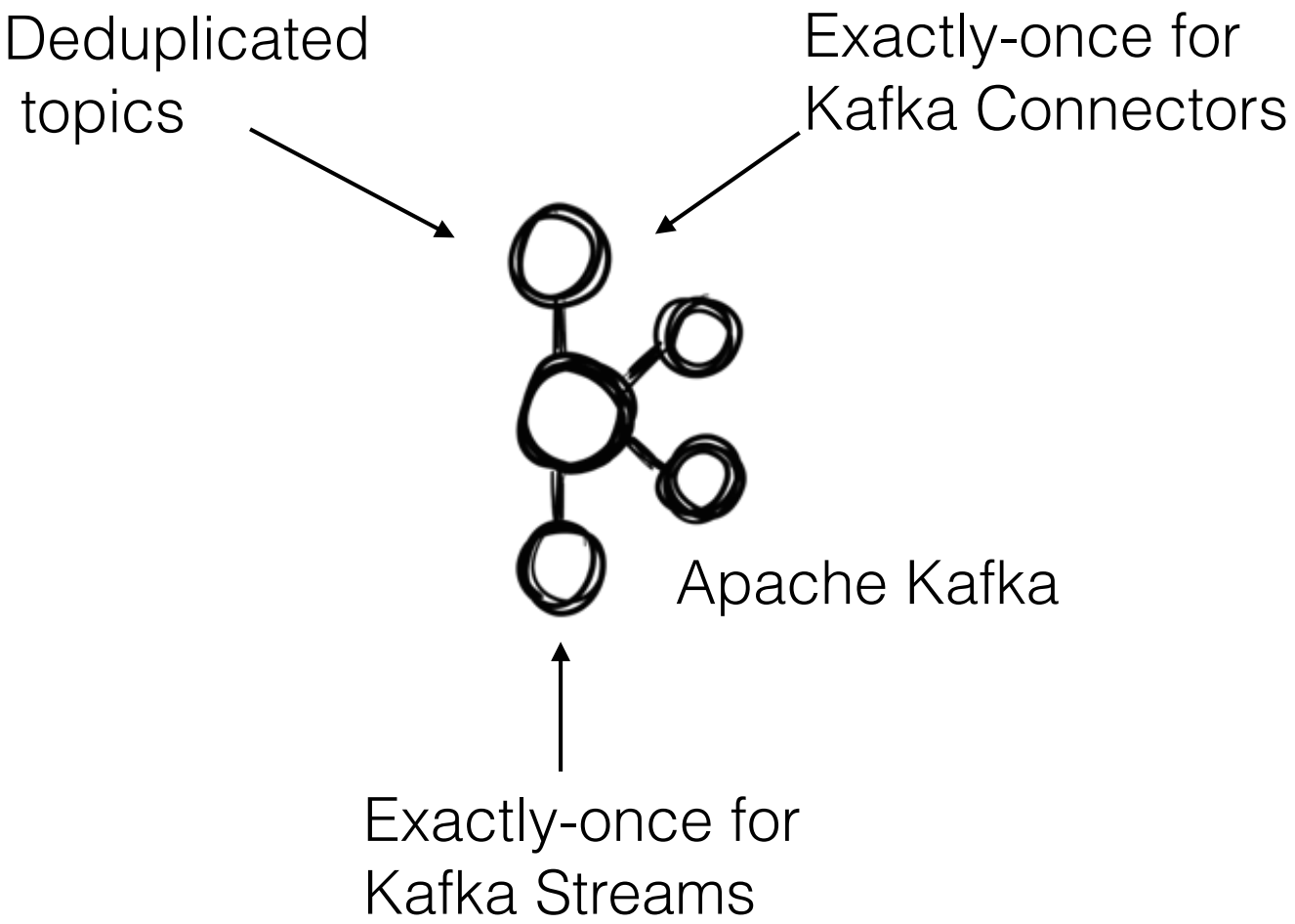
Exactly-onced Kafka Topics

Application involvement for end-to-end guarantee

Summing up

- Exactly-once semantics for RPCs
- State-Machine replication
- Exactly-once for messaging
  - ✓ Retries and deduplication
- Exactly-once for streams
  - ✓ Not sufficient to deduplicate messages
  - ✓ Require the ability of discarding partial state

Future


Deduplicated
topics
Exactly-once for
Kafka Connectors
Apache Kafka
Exactly-once for
Kafka Streams

Acknowledgments

Thanks to my Confluent colleagues:
✓ Eno Thereska
✓ Neha Narkhede
✓ Ismael Juma
✓ Ben Stopford
✓ Jun Rao
✓ Jay Kreps
✓ Guozhang Wang
✓ Ewen Cheslack-Postava
✓ Sriram Subramanian

Questions?

e-mail: fpj@apache.org
twitter: @fpjunqueira
web site: http://fpj.me

Visit us in the Confluent Booth (#217)

- Kafka: The Definitive Guide = Book Giveaway and Signing
- Making Sense of Stream Processing = Book Giveaway

Kafka Training with Confluent University

- Kafka Developer and Operations Courses
- Visit www.confluent.io/training

Want more Kafka?

- Download Confluent Platform Enterprise at http://www.confluent.io/product
- Apache Kafka 0.10 upgrade documentation at http://docs.confluent.io/3.0.0/upgrade.html
- Kafka Summit recordings now available at http://kafka-summit.org/schedule/