IT5100B

Industry Readiness

Stream Processing

LECTURE 4

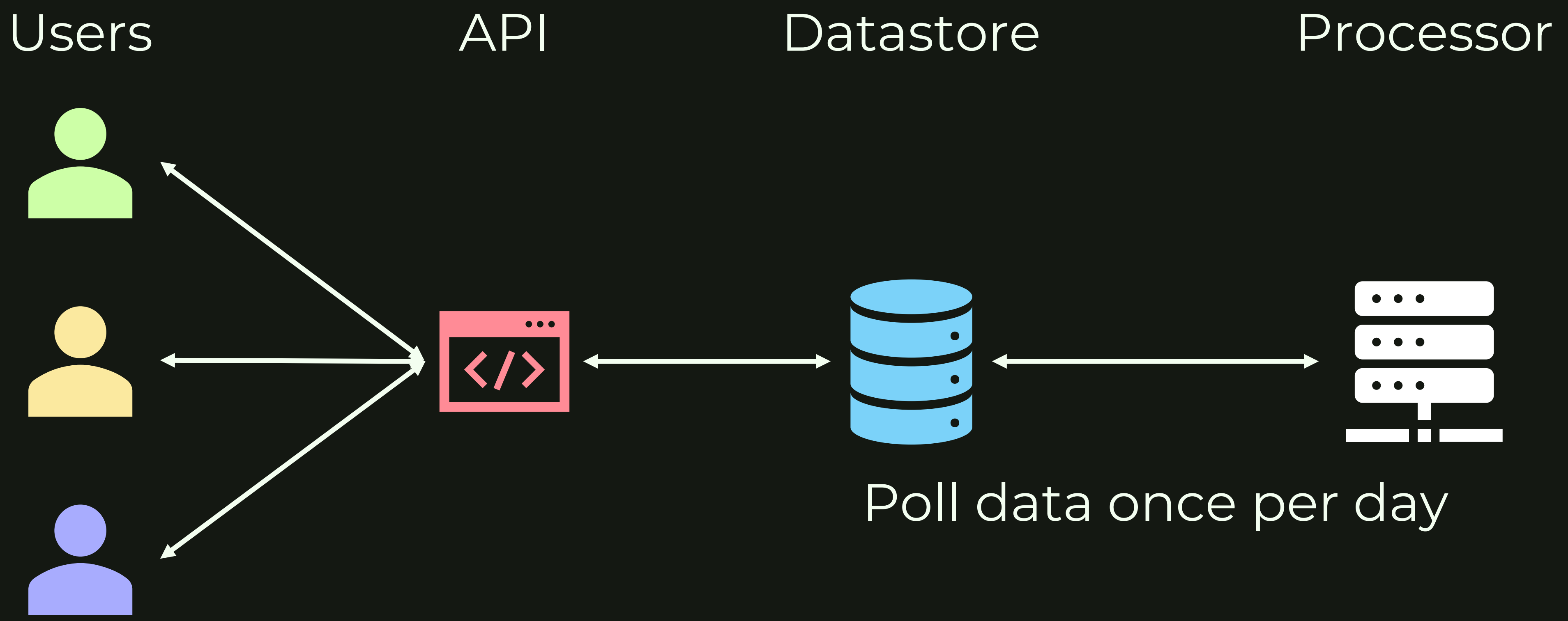
Event Streaming

FOO Yong Qi
yongqi@nus.edu.sg

CONTENTS

WHAT/WHY EVENT STREAMING

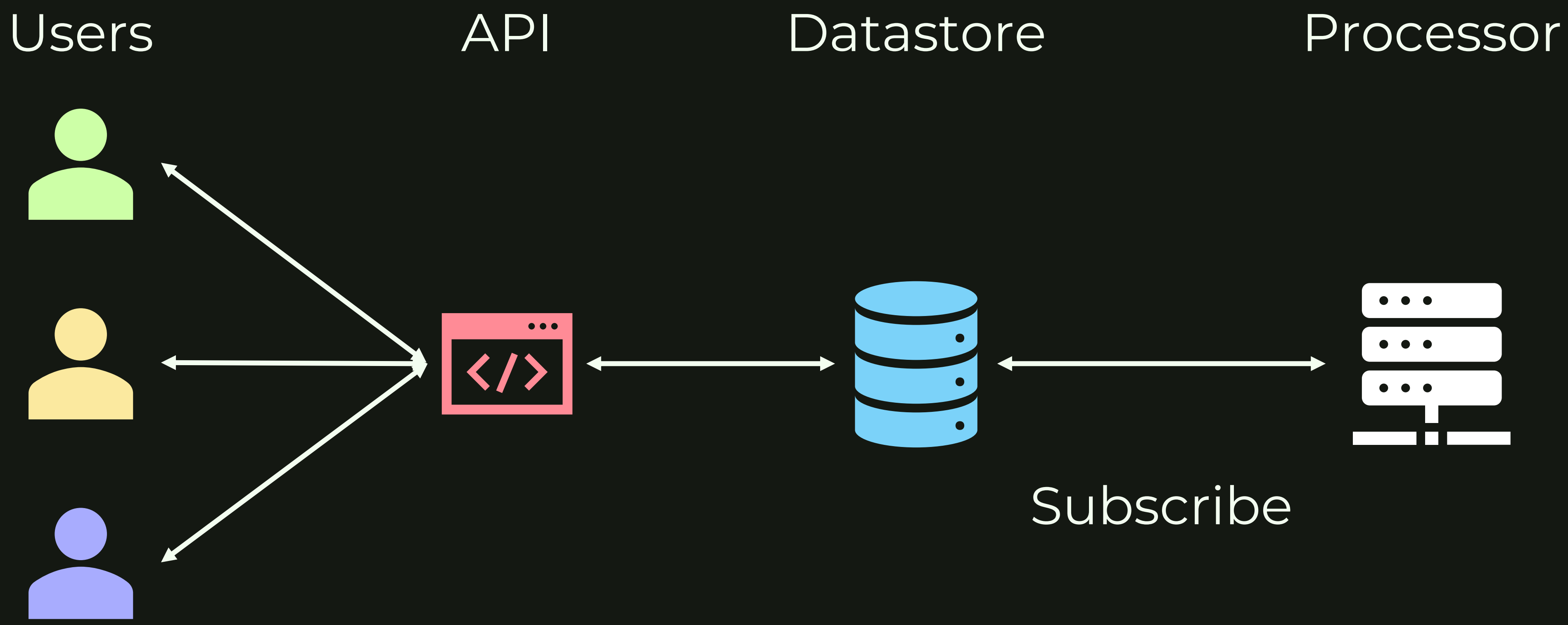
DATA IN MOTION



Batch processing: processor pulls data from data store as a batch, then handles processing

Increased latency

DATA IN MOTION



Processor **subscribes** to state changes in **real time**, avoiding need for batch processing

DATA IN MOTION

MODERN SYSTEM REQUIREMENTS

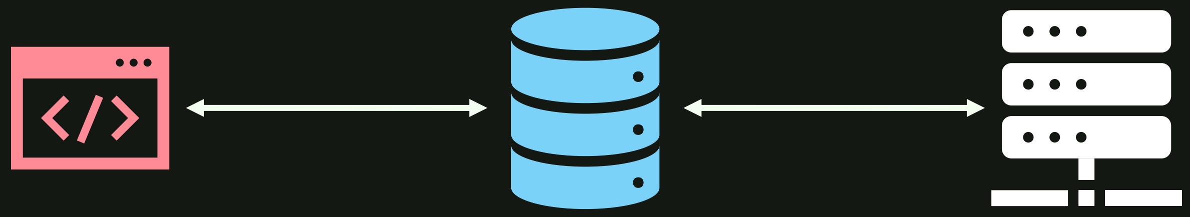
Increasingly, applications process **data in motion** (instead of data at rest)

- Newspaper (daily report) vs news feed
- Polling food orders by batch vs order notifications
- CRUD digital banking vs advanced financial systems

Is having a subscription-based architecture **enough**?

DATA IN MOTION

OTHER CONSIDERATIONS



Complex logic for interacting with data store

Complex implementation of services observing state changes

Application may require different views of the same data

EVENTS VS STATE

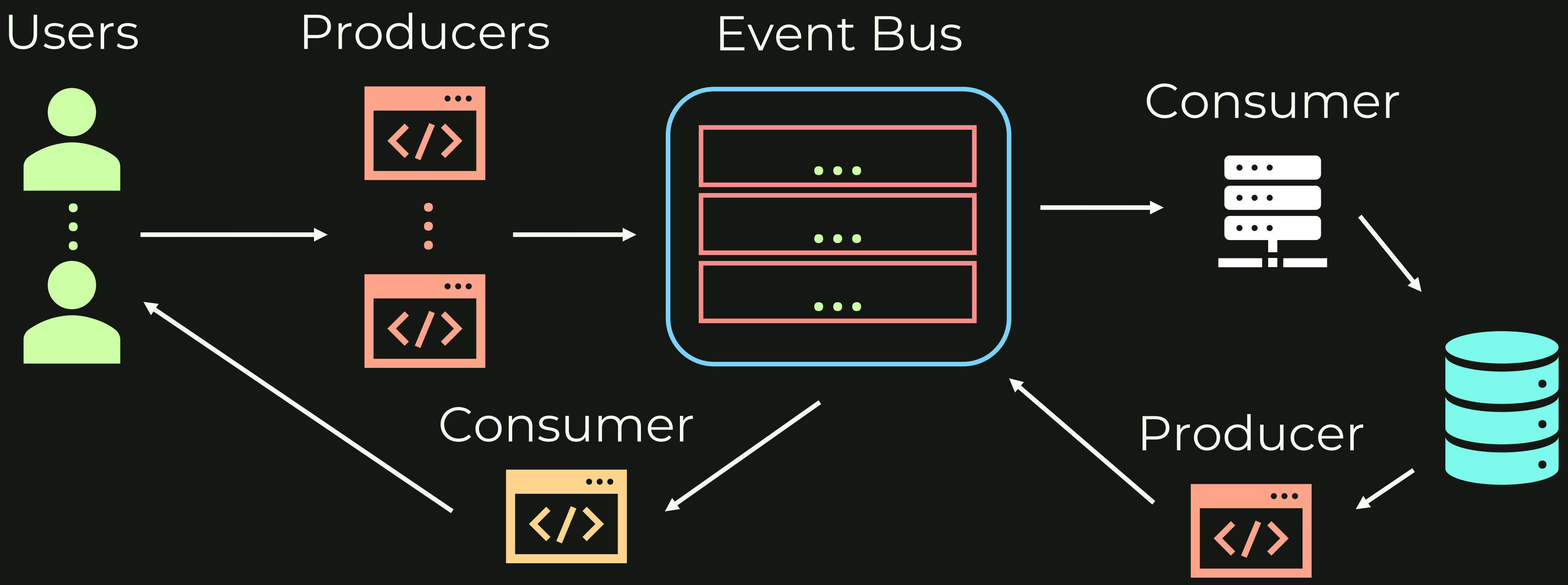
REDUCE LEVEL OF ABSTRACTION

If you think about it:

Data in a database is an **aggregation** of **events**

Keep things simple! Things that happen are events, store and deal with events!

EVENT BUS



Store events instead; allows (1) decoupling producers and consumers, (2) easier to support more complex business logic and analysis, (3) supports real-time processing

APACHE KAFKA®

APACHE KAFKA®

EVENT STREAMING PLATFORM

Open-source distributed **event streaming platform** for high-performance data pipelines, streaming analytics, data integration and mission-critical applications

APACHE KAFKA®

CORE CAPABILITIES

High Throughput

Scalable

Permanent Storage

High Availability

APACHE KAFKA®

USE CASES

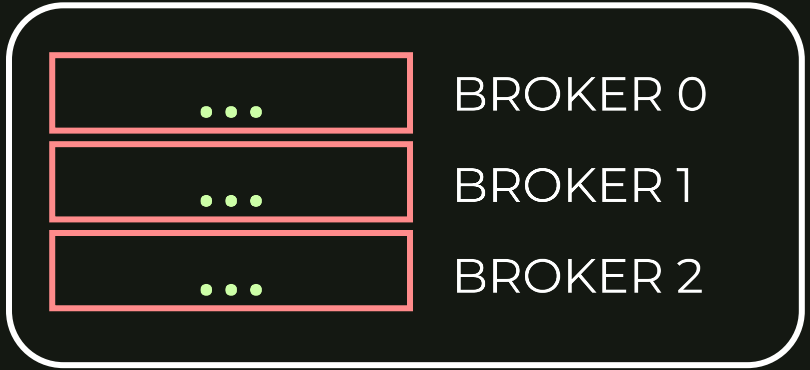
Over **80% of Fortune 500 companies** use Apache Kafka:

- Financial services: real-time fraud detection, customer experience
- E-commerce: real-time analysis, notifications, 360 view of customers
- Healthcare: Monitoring IoT medical devices
- Online gaming: better reliability and scalability
- Automotive: sensor data for real-time alerts and hazard detection

KAFKA CLUSTERS

HOW KAFKA APPLICATIONS WORK

BROKERS



Kafka Cluster

Kafka clusters comprise of several nodes; some of which are brokers; each broker can be a separate machine/process/container

HOW KAFKA APPLICATIONS WORK

CLUSTER CONSENSUS WITH KRAFT

NODE 0 (BROKER)
NODE 1 (CONTROLLER)
NODE 2 (CONTROLLER + BROKER)

Kafka Cluster

A Kafka cluster is split into the **data plane** (handles data), and the **control plane** (handles cluster metadata)

Brokers work in data plane, controllers work in control plane

Cluster metadata was previously managed by Apache Zookeeper®

KAFKA CLUSTER STARTUP

DOCKER

Containers—lightweight, standalone, executable package of software that includes everything needed to run an application

Run Kafka on any machine (can install locally)

Instead, we shall use docker to run Kafka brokers on docker containers

KAFKA CLUSTER STARTUP

MULTI-NODE CLUSTER

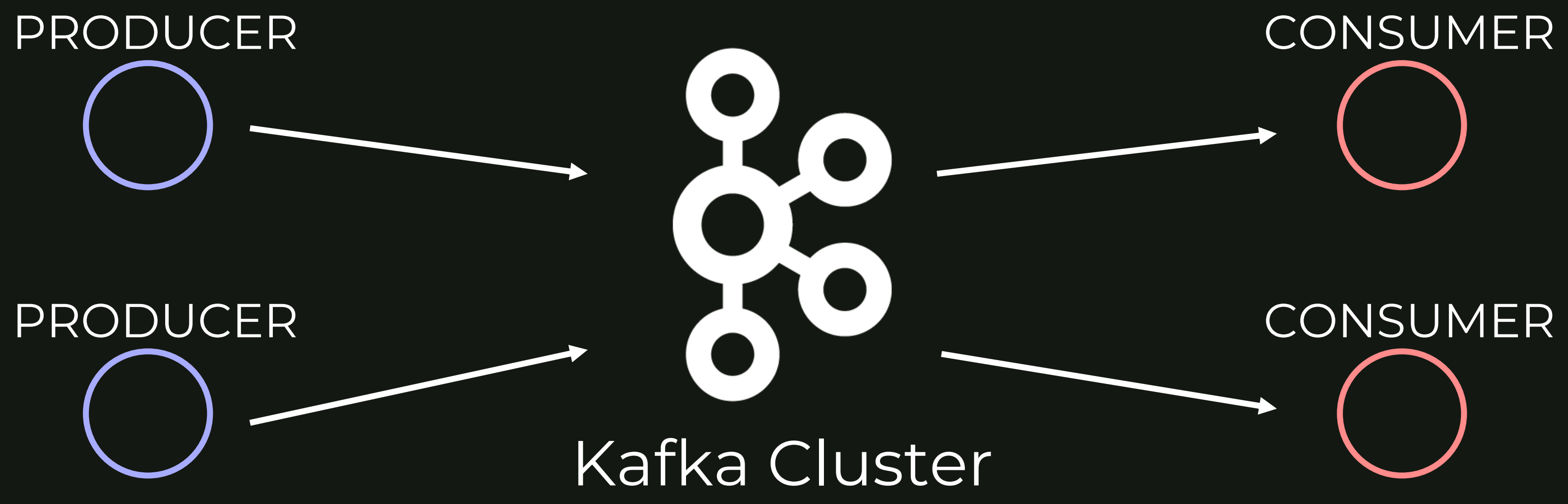
Can use `docker-compose` to start up a cluster consisting of multiple nodes (see Canvas :: Files for compose file):

1. Three node cluster running in shared mode
2. Exposed ports (`EXTERNAL`) at ports `9092`, `9093`, `9094` respectively
3. All have distinct node IDs, and shared KRaft cluster ID

Connect to cluster via any of the broker's external ports (`localhost:9092`, `localhost:9093` or `localhost:9094`)

HOW KAFKA APPLICATIONS WORK

PUBLISHER/SUBSCRIBER MODEL

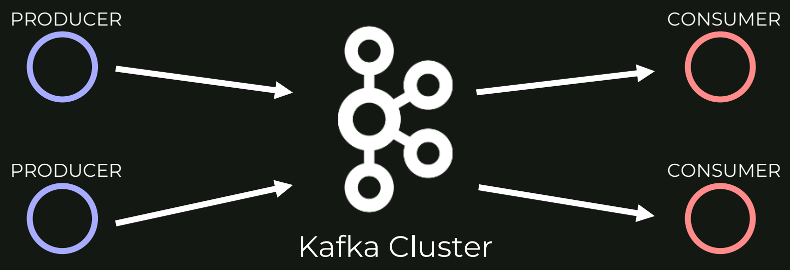


Publishers or **producers** write events into Kafka cluster

Subscribers or **consumers** read events from Kafka cluster

HOW KAFKA APPLICATIONS WORK

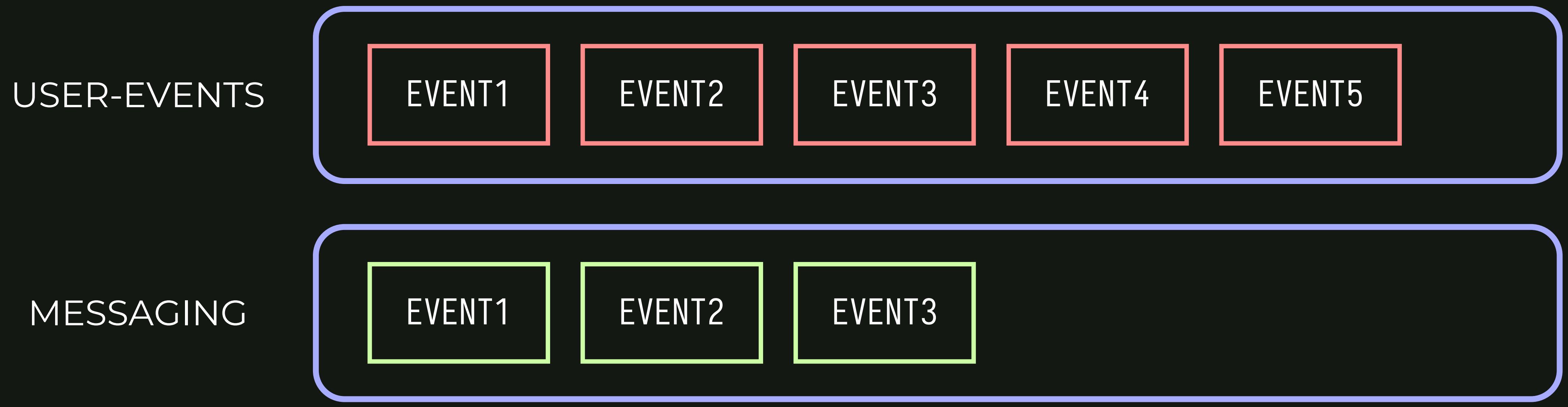
PUBLISHER/SUBSCRIBER MODEL



Producers and consumers are decoupled:

- Slow consumers do not affect producers
- Producers and consumers can scale independently
- Failure of producer/consumer does not affect system
- Consumer with new features can be added independently

TOPICS



Just like database tables, events are organized into **topics**:

- Logical category of related events
- Producers to topics to consumers are many-to-many
- Unlimited number of topics (don’t overuse topics though)

EVENTS

```java
public class User {
    UUID id;
    String name;
}
```

UserSerializer →

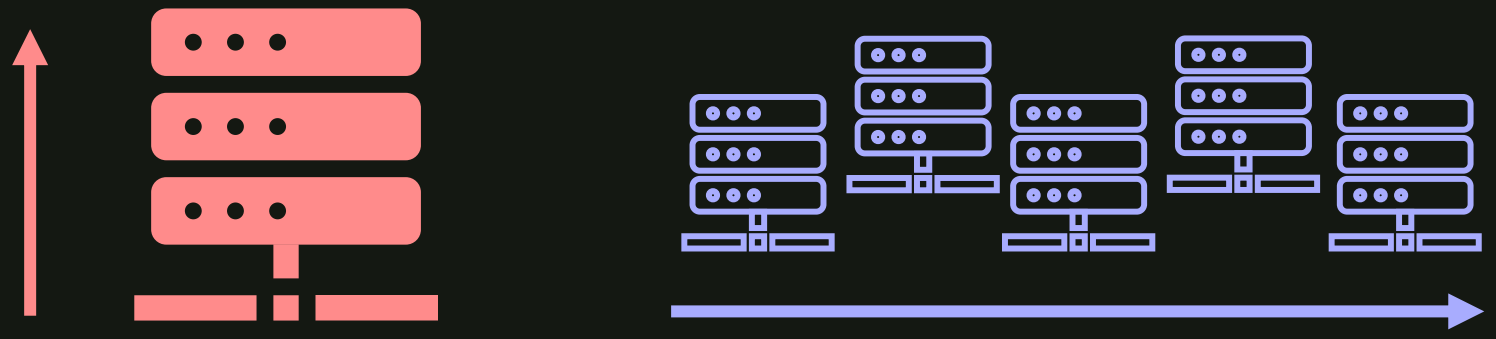
← UserDeserializer

Optional headers
Key
Value
Timestamp

An event is just a **key-value pair**

Conversion of Java objects to/from event key/value requires **serializer/deserializer** (**SerDes**)

SCALING



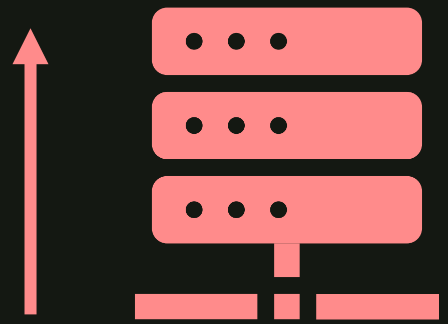
How does a system handle increasing load?

Vertical scaling—getting a stronger/bigger computer

Horizontal scaling—get more computers

SCALING

VERTICAL SCALING



Pros:

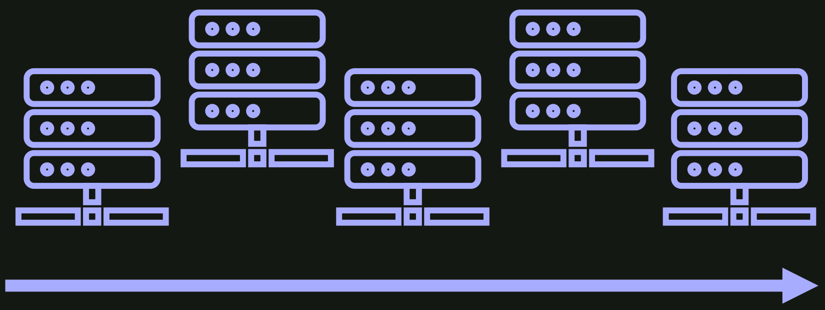
- Easy to implement
- Easy to maintain
- Cost-effective

Cons:

- Cannot scale indefinitely
- Single point of failure

SCALING

HORIZONTAL SCALING



Pros:

- Flexible scaling
- Much higher limits
- Redundancy and reduced downtime
- Better proximity to end-users

Cons:

- Distributed computing is more complex
- Higher initial cost of adding more computers
- Harder to maintain

DISTRIBUTED COMPUTING

RANDOM ADVERTISEMENT

Several courses in NUS dive into distributed computing, worth checking out if interested:

CS5223 Distributed Systems

CS5424 Distributed Databases

PARTITIONS

HORIZONTAL SCALING

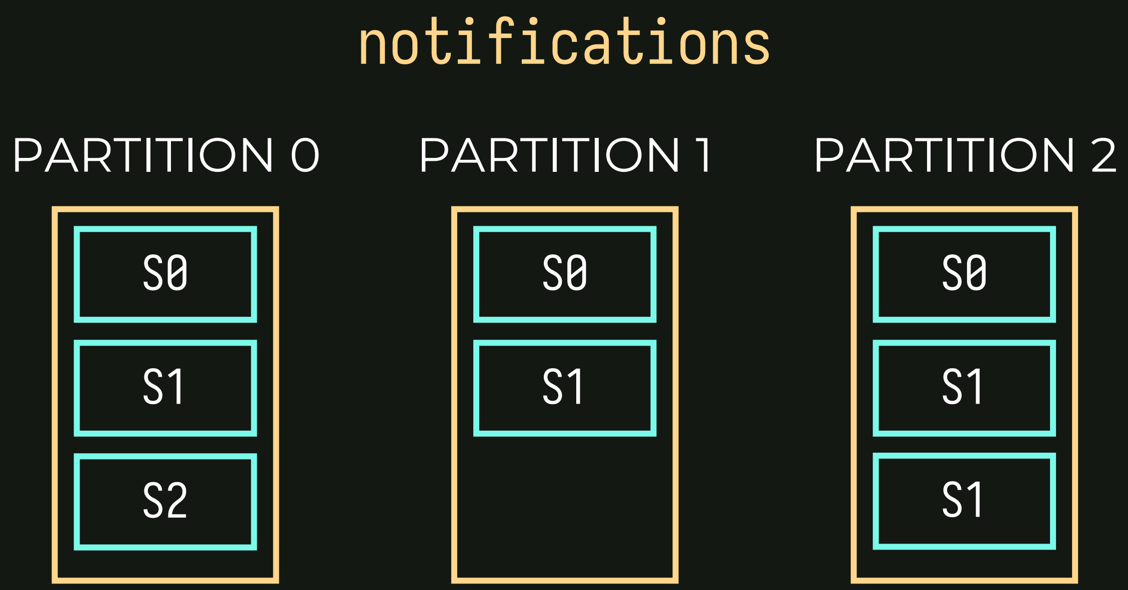
Topics are split into partitions; each partition lives on one broker, one broker can have many partitions

SEGMENTS

Partitions are split into **segments** containing the actual events, like a rolling log file

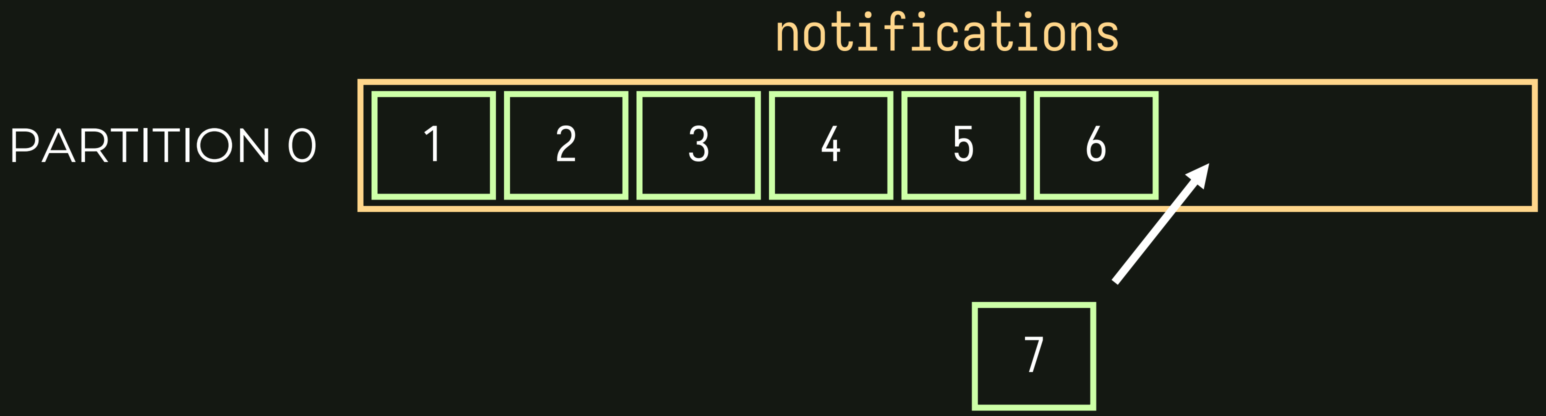
Segments are stored in broker’s local storage (persistent!)

TOPICS/PARTITIONS/SEGMENTS



Overall, a topic is split into different partitions (that can live across different brokers); the three partitions above form the `notifications` stream; partitions are the units of parallelism of a topic

THE EVENT LOG



A partition of a topic is an append-only immutable log of events; consuming does not pop/destroy events

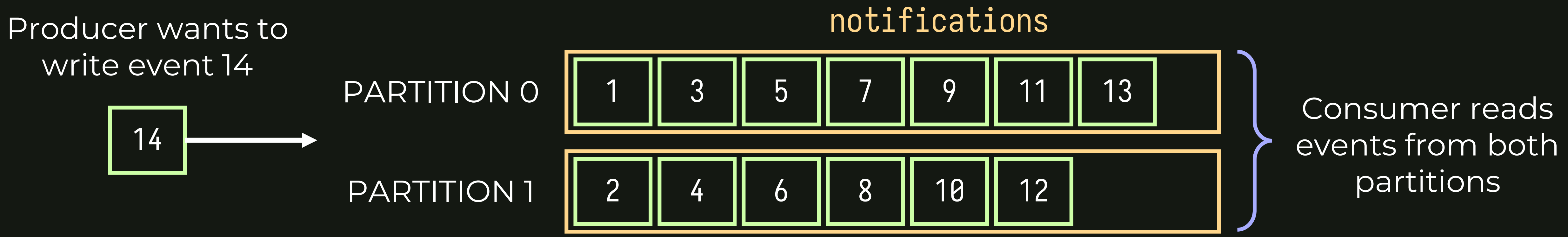
Segments are retained for (by default) 7 days (can be set globally or per-topic)

RETENTION POLICY

How long should/can data be retained?

- How long do you need the data?
- How much data is there? Can you afford to keep it all?
- Regulatory considerations: PDPA, GDPR etc.

PRODUCER WRITES



Which partition does a producer write to?

- To balance the load, can write in round-robin fashion
- Consumer will read topic from both partitions in interleaved fashion
- Ordering therefore generally not guaranteed

EVENT ORDERING

Ordering events is important sometimes:

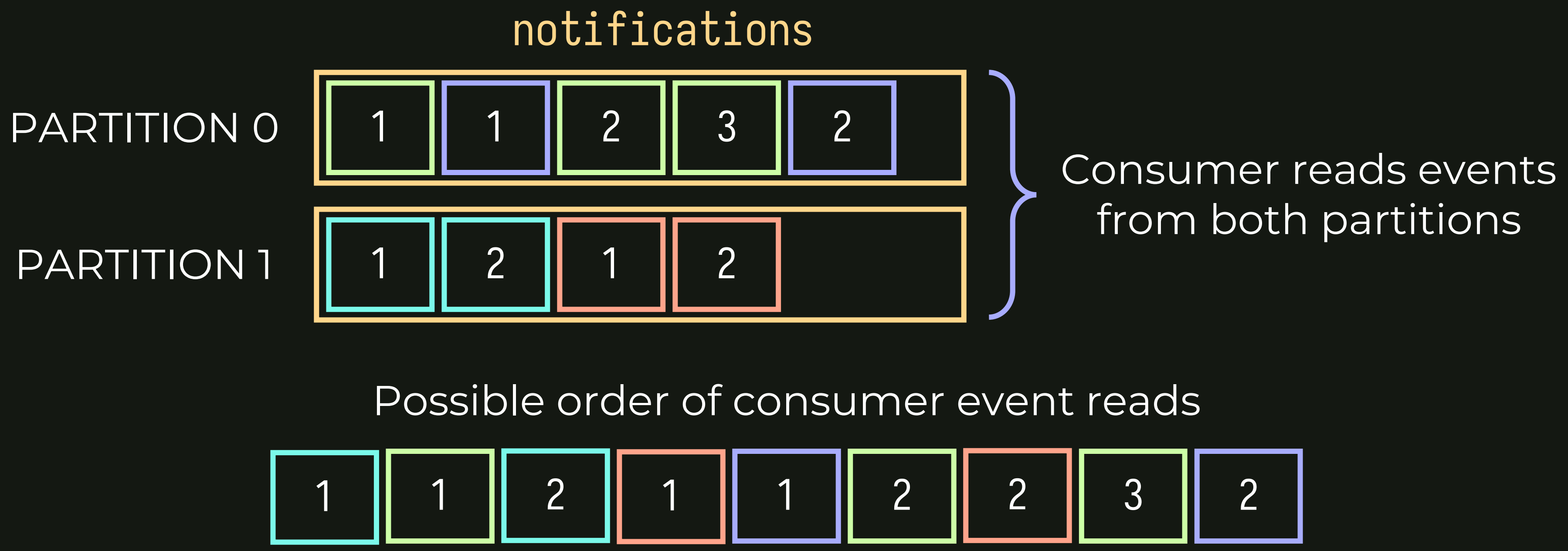
- User state change events
- Order of messages received in a chatroom
- Outgoing payments from a user

Let event key decide partition to write to!

```
p = hash(key) % num_partitions
```

EVENT ORDERING

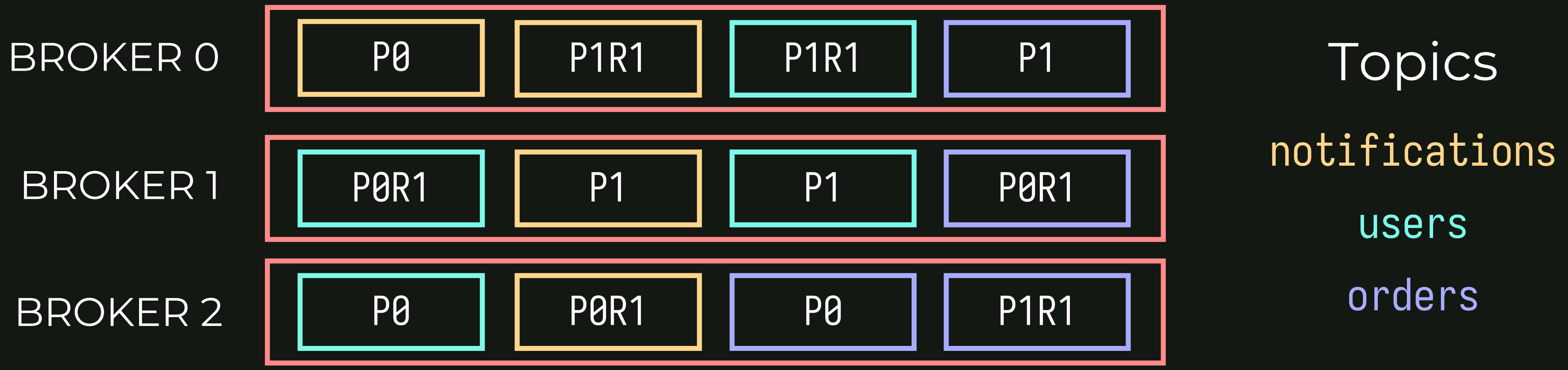
EVENT KEY DECIDES PARTITION



When producing events to a topic, events that should be ordered must share the same key

REPLICATION

FAULT TOLERANCE AND HIGH AVAILABILITY



`replication-factor=2`

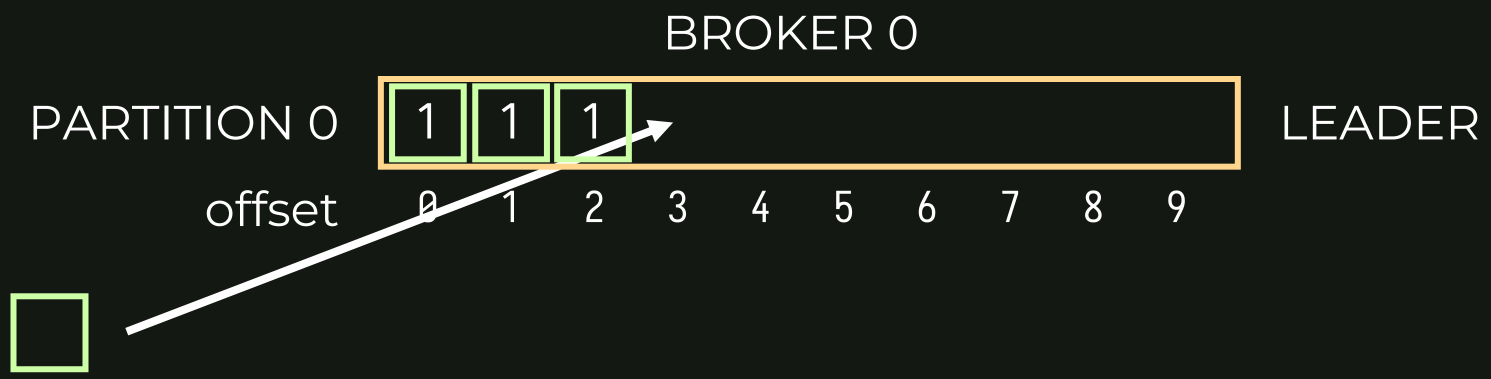
Partitions can be replicated for fault tolerance

One designated leader, the rest are followers

Produce/consume to/from leader, optionally consume from follower

In-sync replica (ISR) list is list of replicas that caught up with leader

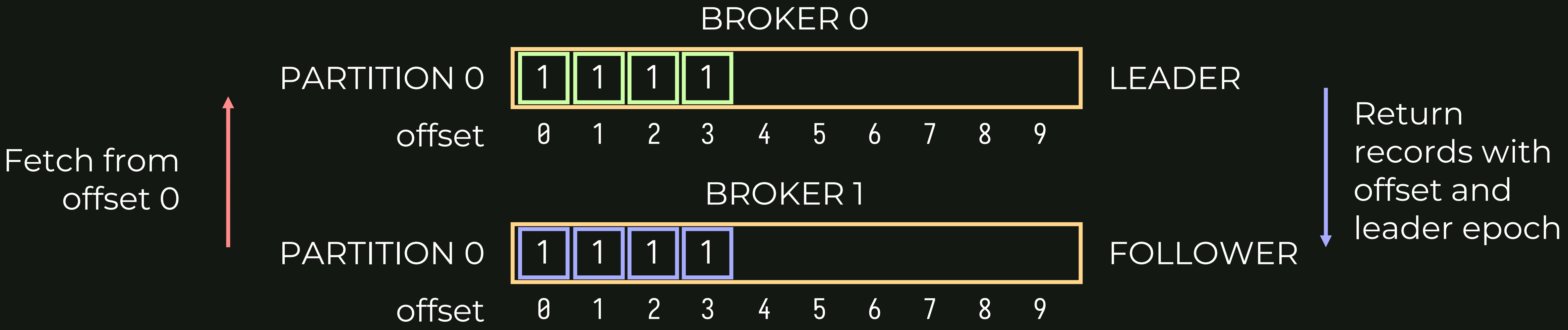
COORDINATION IN THE DATA PLANE



When a producer writes to a topic partition, it writes to the leader

Partition maintains offset (index) of events, each event also has leader epoch

COORDINATION IN THE DATA PLANE

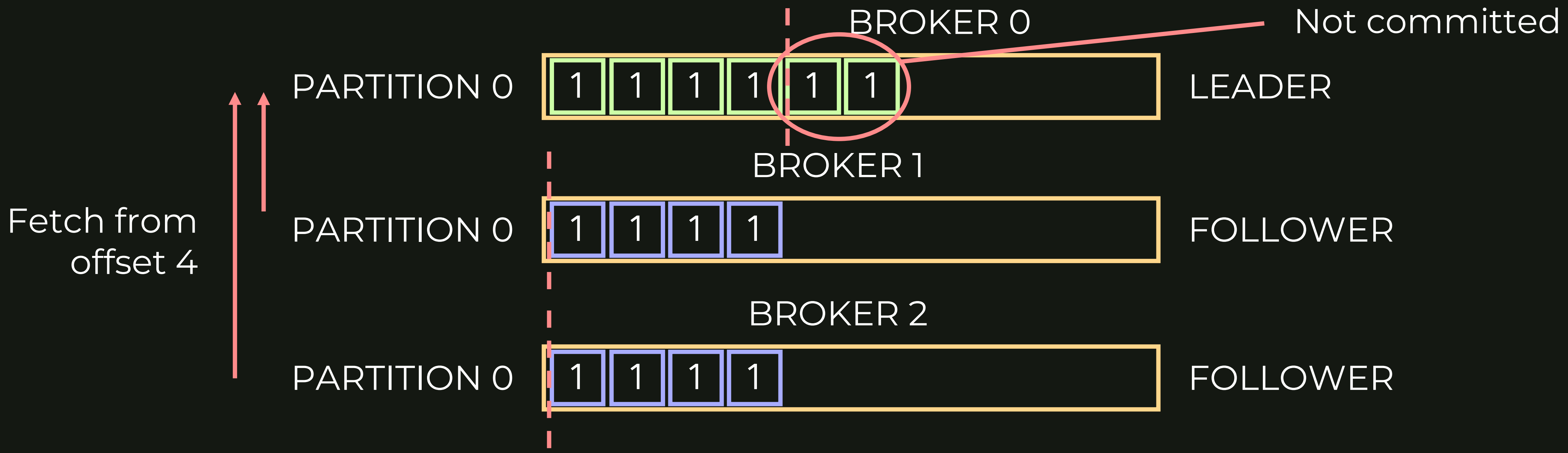


Follower fetches data from leader with offset

Leader sends records starting from requested offset

Follower appends events to log (with leader epoch)

COORDINATION IN THE DATA PLANE



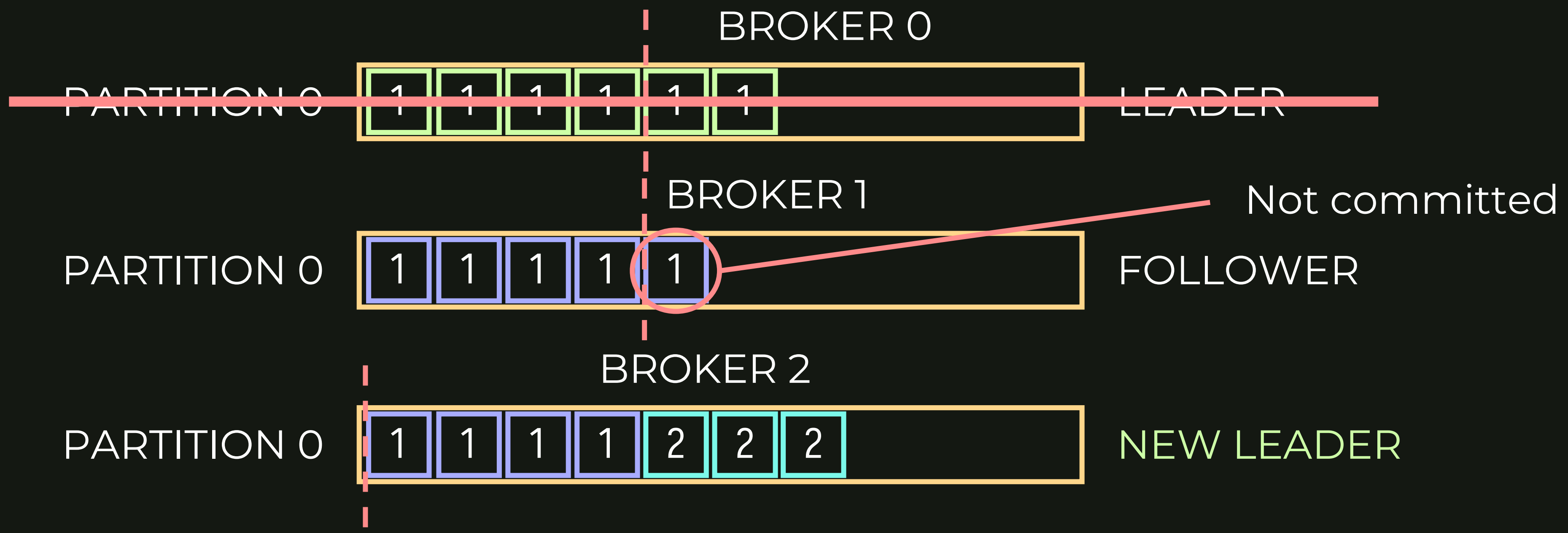
If all followers in ISR make fetch requests for offset at least n, followers all received events before offset n

Leader advances high watermark to commit events; committed events then consumed by consumers

Future fetch response sends new high watermark to followers

NUS National University of Singapore | School of Computing

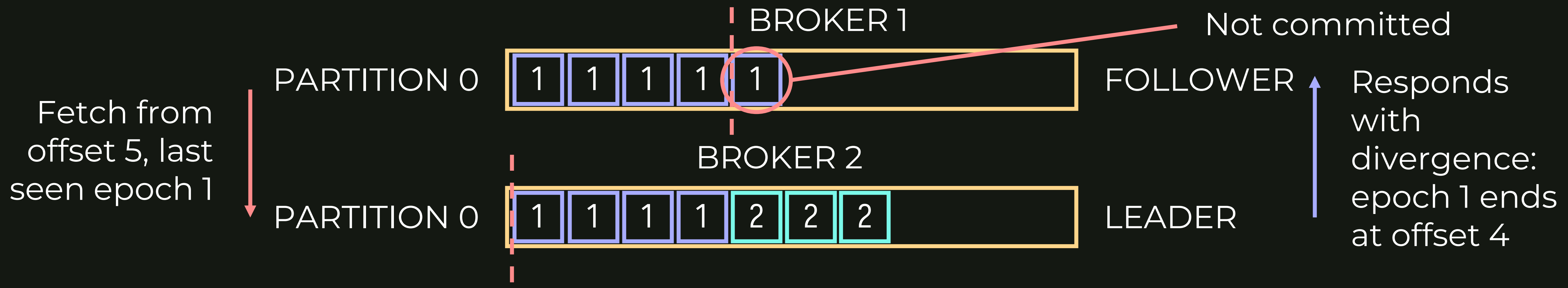
COORDINATION IN THE DATA PLANE



If leader dies, control plane (consisting of controllers) elects new leader from ISR list

New leader begins receiving new events from producer

COORDINATION IN THE DATA PLANE

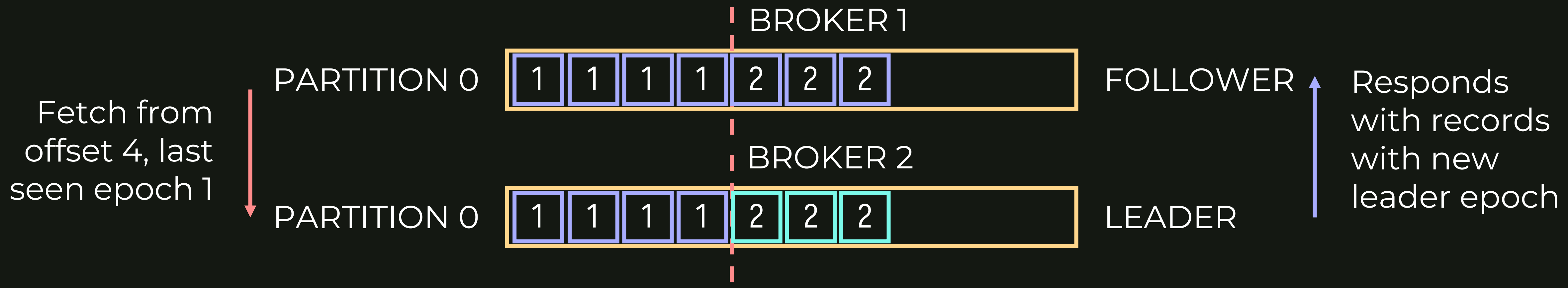


Fetch request contains last seen epoch

New leader identifies diverging epoch and sends response to follower

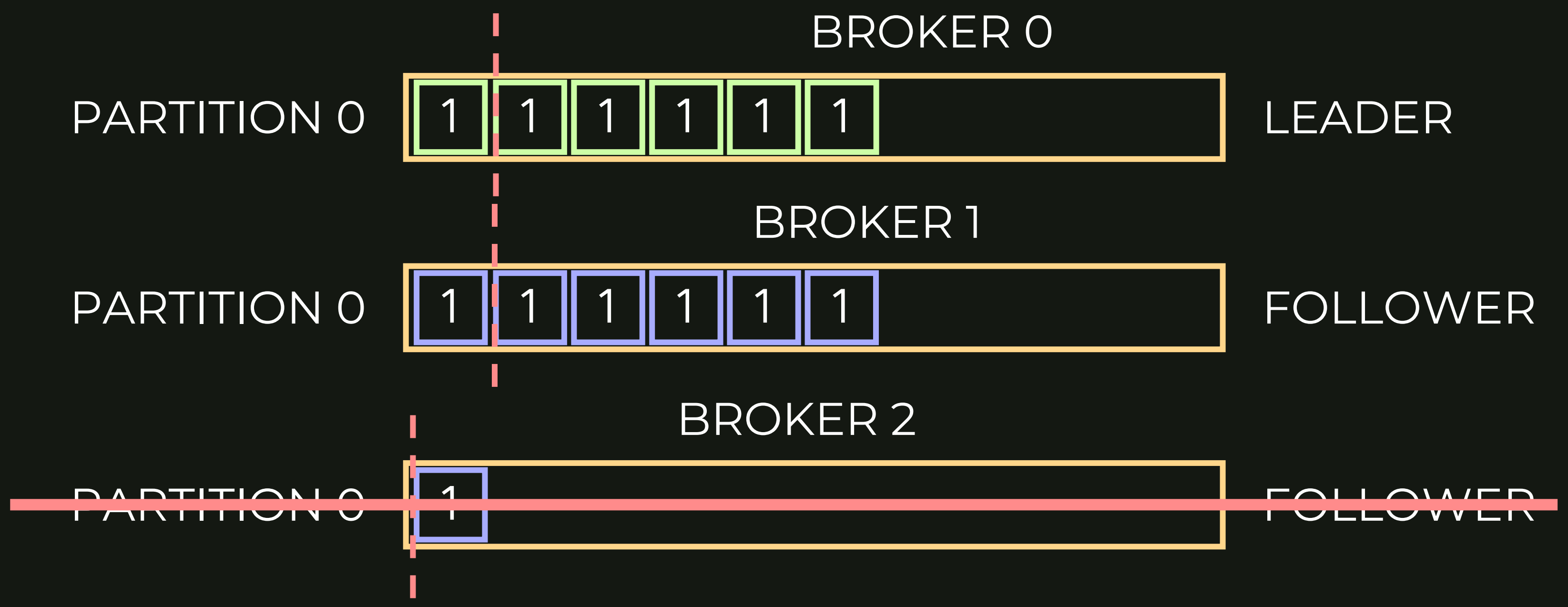
Follower truncates events of epoch 1 from offset 4 onwards

COORDINATION IN THE DATA PLANE



Now follower log has been reconciled and is consistent, fetch request proceeds as per normal

COORDINATION IN THE DATA PLANE



If follower in ISR is slow, (no fetch request after some time), removed from ISR so high watermark can continue to advance

`replica.lag.time.max.ms`

KAFKA TOPIC CREATION

```
kafka-topics.sh --create --topic hello --replication-factor 2
--partitions 3 --bootstrap-server kafka0:9094
```

Create a topic called `hello` with replication factor of 2 and 3 partitions (bootstrap server can be any advertised listener of any kafka node)*

```
kafka-topics.sh --describe --topic hello --bootstrap-server kafka0:9094
```

Describe topic `hello`

*Add in `--config retention.ms=1000` onto this command to set the retention period to 1s

NUS National University of Singapore | School of Computing

PRODUCERS AND CONSUMERS

PRODUCERS

EVENT → SERIALIZER → PARTITIONER → BUFFER → Kafka Cluster

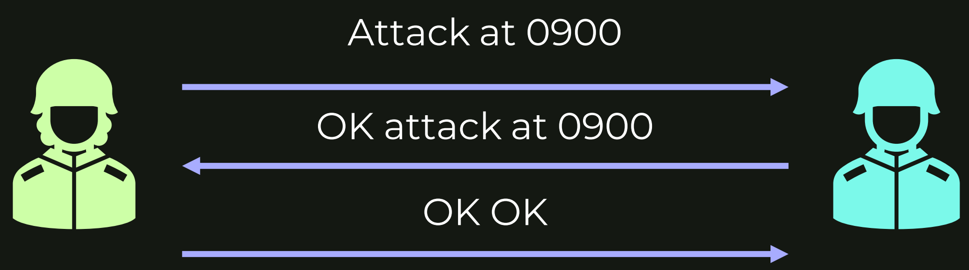
Producers send events to cluster; events contain topic, optional partition and key, and event value

Keys and values go through respective serializers

Events are buffered before sending across network to cluster

DELIVERY GUARANTEES

TWO GENERALS PROBLEM



It is impossible for two generals to agree on when to attack!

In general, **exactly once delivery** of events to Kafka is **impossible**

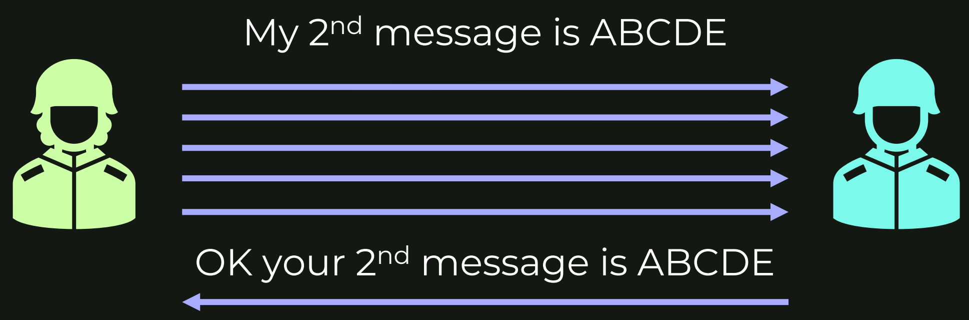
PRODUCER ACKNOWLEDGEMENTS

Kafka cluster can acknowledge produce request in three ways:

- No acknowledgement (`NONE`)—at most once delivery, some data loss, low latency
- Leader acknowledgement (`LEADER`)—at least once delivery, less data loss, some latency
- All acknowledgement (written to all ISRs) (`ALL`)—at least once delivery, no data loss, more latency

EXACTLY-ONCE SEMANTICS

IDEMPOTENT PRODUCERS



Idempotent producer with at least-once delivery gives **exactly once processing**

From Kafka version 3.0, by default, producers are idempotent with all acknowledgements

PRODUCING ATOMICALLY

TRANSACTIONS

Producing several events simultaneously to multiple topics/partitions can be done with Kafka **transactions**

REACTOR KAFKA

REACTOR + KAFKA CLIENTS

Reactor Kafka is a Project Reactor wrapper over the Kafka Clients API

You may wish to install the following dependencies:

1. Reactor Kafka
2. Some logging implementation (log4j-impl)

REACTOR-KAFKA PRODUCER

CREATING THE PRODUCER

```java
public static KafkaSender<Integer, String> createSender() {
    Properties props = new Properties();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "localhost:9092,localhost:9093,localhost:9094");
    props.put(ProducerConfig.CLIENT_ID_CONFIG, "my-hello-producer");
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class);
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    SenderOptions<Integer, String> senderOptions = SenderOptions.create(props);
    return KafkaSender.create(senderOptions);
}
```

Configure the producer (KafkaSender)

REACTOR-KAFKA PRODUCER

PRODUCING EVENTS

```java
public static void main(String[] args) {
    KafkaSender<Integer, String> producer = createSender();
    Scanner sc = new Scanner(System.in);
    String message;
    while (!(message = sc.nextLine()).equals("exit")) {
        ProducerRecord<Integer, String> event = new ProducerRecord<>("hello", 0, message);
        SenderRecord<Integer, String, Long> r = SenderRecord.create(event,
                System.currentTimeMillis());
        producer.send(Mono.just(r)).subscribe(System.out::println);
    }
    producer.close();
}
```

Send the record using the **send** method!

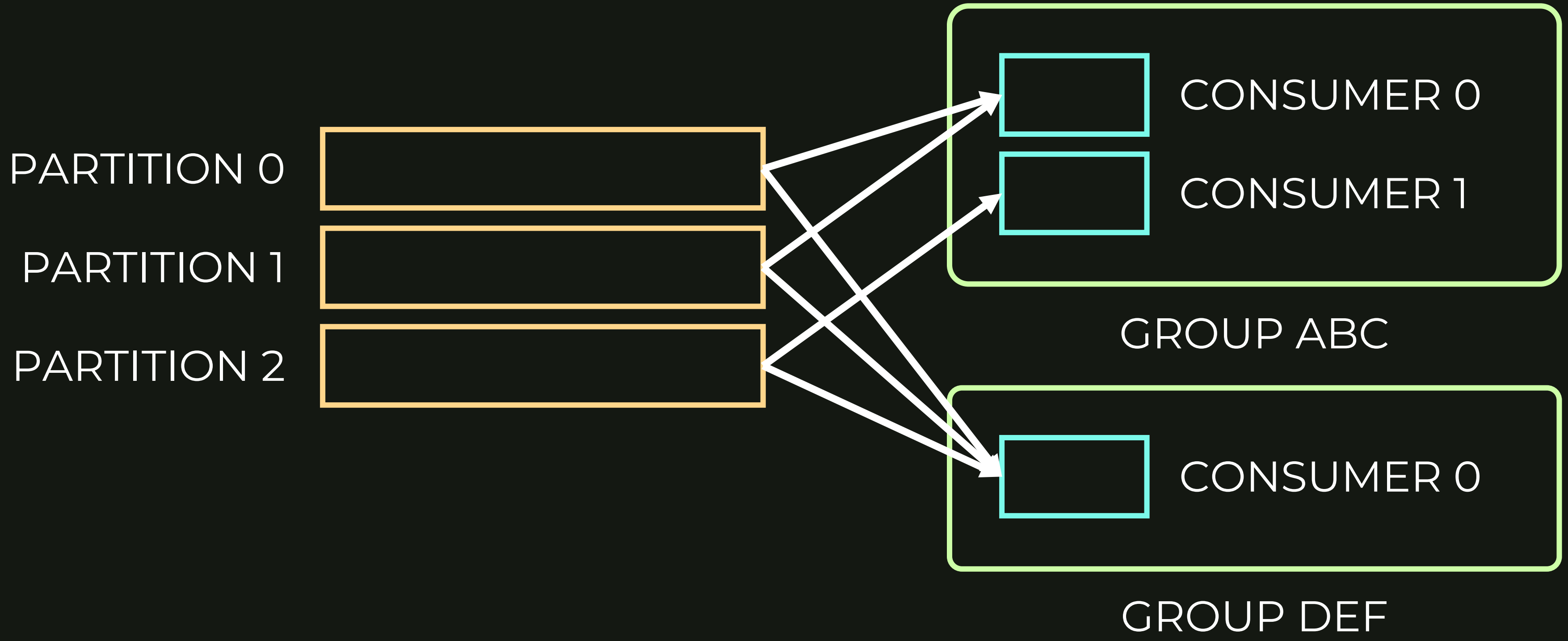
CONSUMERS

Consumers can read from any topic; consumers can read from the beginning of a topic because events persist

On event read, consumer can commit an offset to internal Kafka topic (consumer offsets) so that on subsequent startup, consumer reads from last processed event

CONSUMER GROUPS

PARALLELIZATION OF CONSUMERS



Consumers can be parallelized into several consumers in a consumer group; each topic partition will be assigned exactly one consumer in the group; cluster handles consumer rebalancing

REACTOR KAFKA CONSUMER

CREATING THE CONSUMER

```java
public static KafkaReceiver<Integer, String> createReceiver(String topic) {
    Properties p = new Properties();
    p.put(ConsumerConfig.CLIENT_ID_CONFIG, "client1");
    p.put(ConsumerConfig.GROUP_ID_CONFIG, "group1");
    p.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "localhost:9092,localhost:9093,localhost:9094");
    p.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, IntegerDeserializer.class);
    p.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
    p.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
    p.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");
    ReceiverOptions<Integer, String> receiverOptions = ReceiverOptions.create(p);
    return KafkaReceiver.create(receiverOptions
            .subscription(Collections.singleton(topic)));
}
```

Creates a basic consumer that subscribes to one topic

REACTOR KAFKA CONSUMER

CONSUMING EVENTS

```
public static void main(String[] args) {
    KafkaReceiver<Integer, String> receiver = createReceiver("hello");
    receiver.receive().doOnNext(x -> {
        System.out.println(x.value());
        x.receiverOffset()
            .commit()
            .subscribe();
    }).blockLast();
}
```

A simple consumer that receives events and commits all seen messages