Kafka as a backend

Introduction to Kafka stream and state stores

Gonzalo Borobio 24.03.2021

Reaktor

Kafka as a backend

Table of contents

Kafka as a backend

Table of contents

Kafka as a backend

Table of contents

A basic MOBA game

MOBA

A basic MOBA game

Multiplayer online battle arena:

=> 2 teams, Team A and Team B

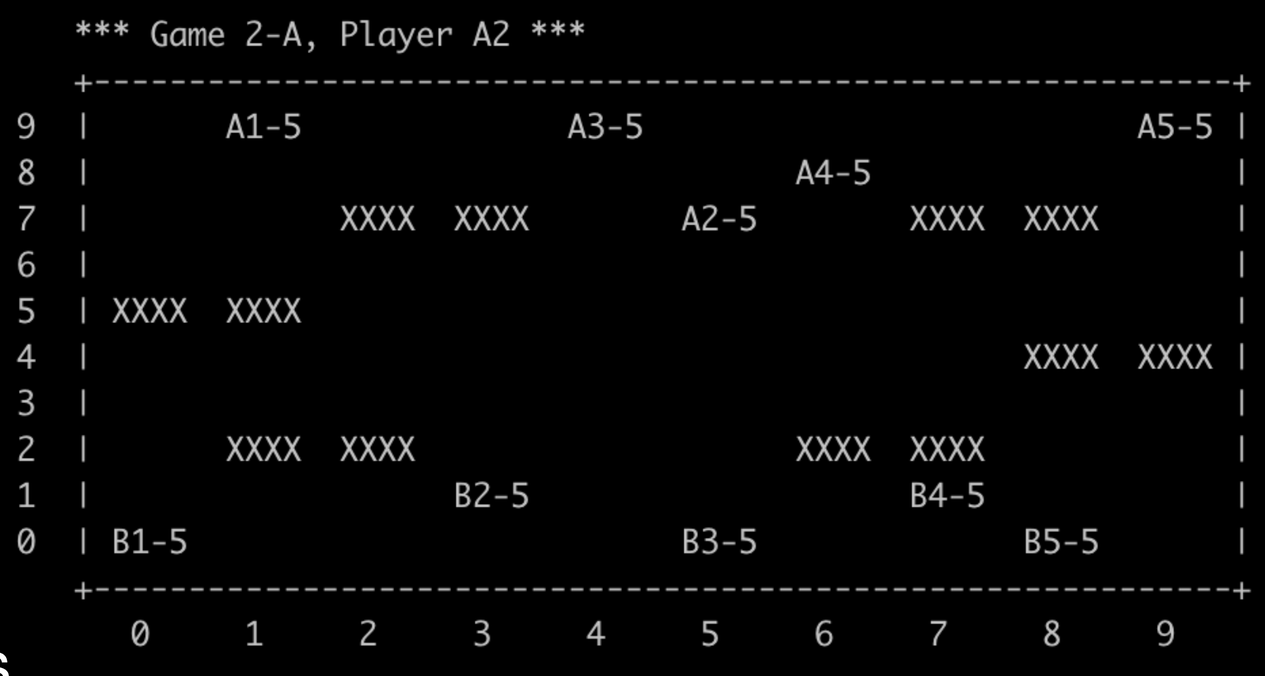
=> 5 players per team

=> each player has 5 grenades

=> move player with arrow keys

=> throw grenades using WASD keys

=> player can either throw or move



Client-server architecture

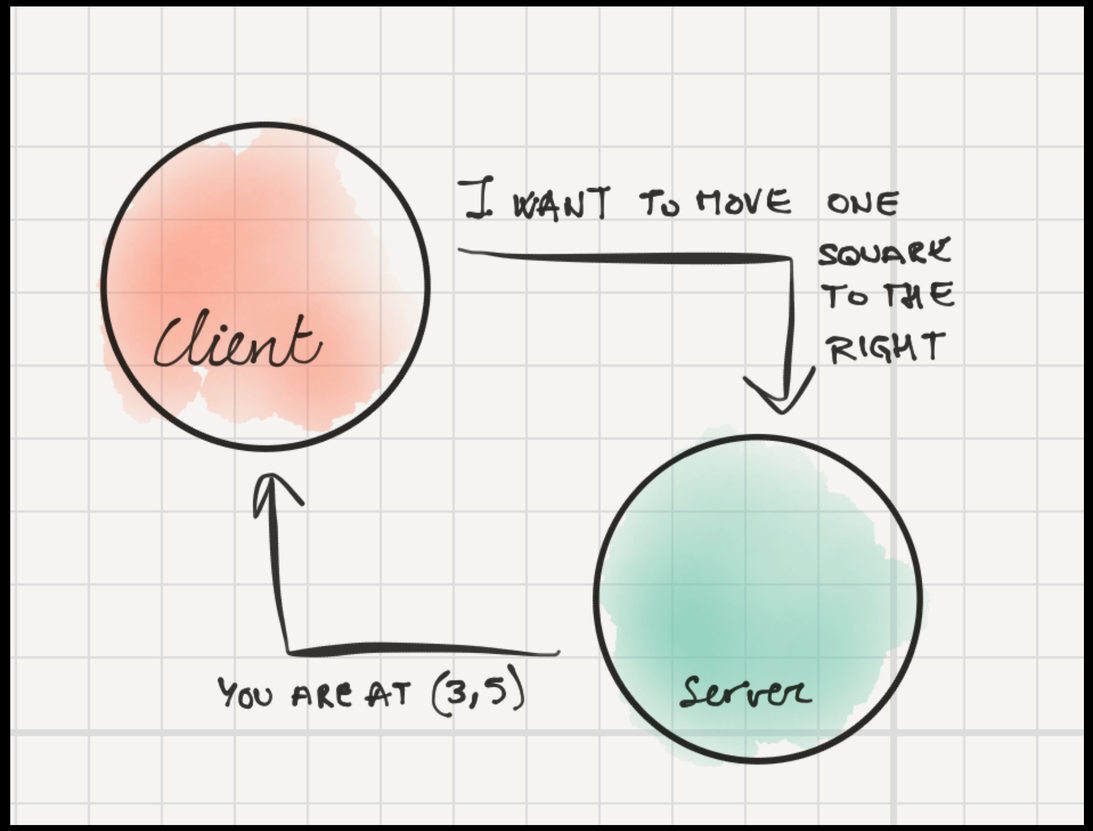
Client-server architecture

Authoritative servers

Clients might cheat 🤥 so we don't trust them.

Instead of saying **I'm at (3,5)**, the client tells the server **I want to move one square to the right**.

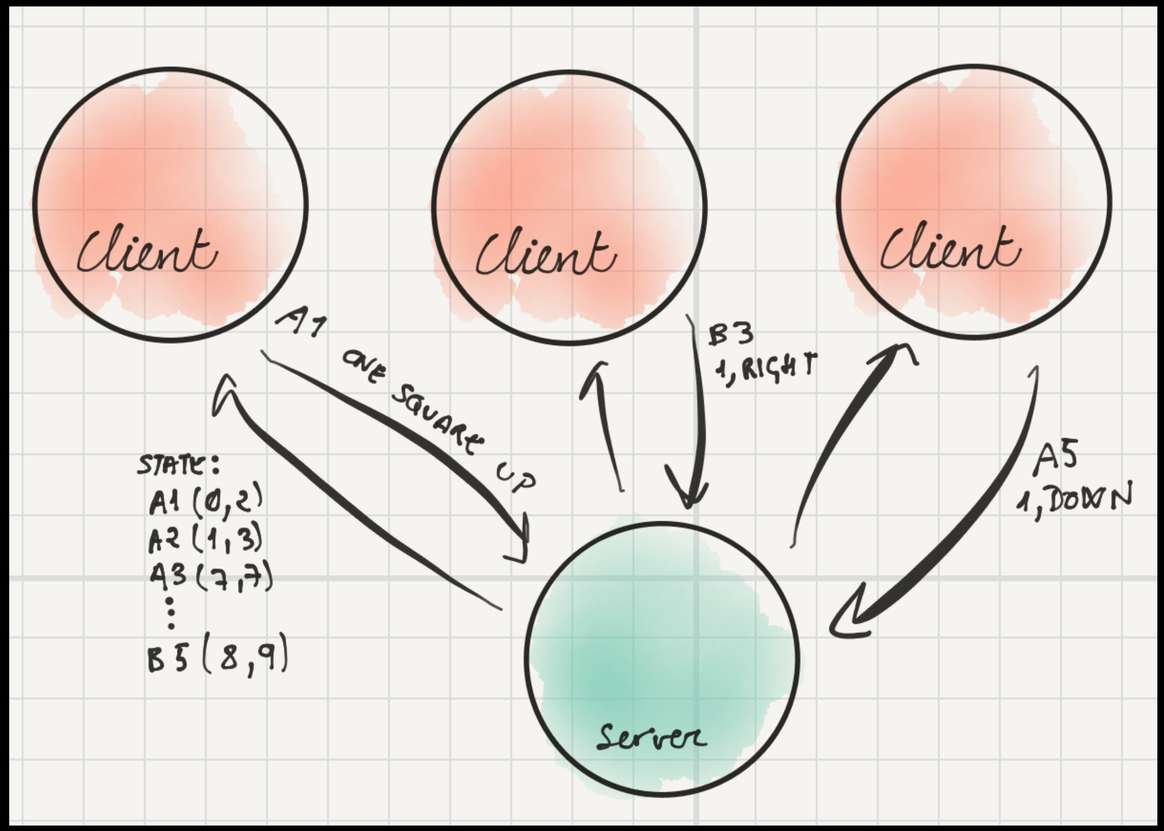
This is known as authoritative servers and dumb clients



Client-server architecture

Authoritative servers

The server validates players moves and returns players' new positions.



A bit of Kafka

Kafka

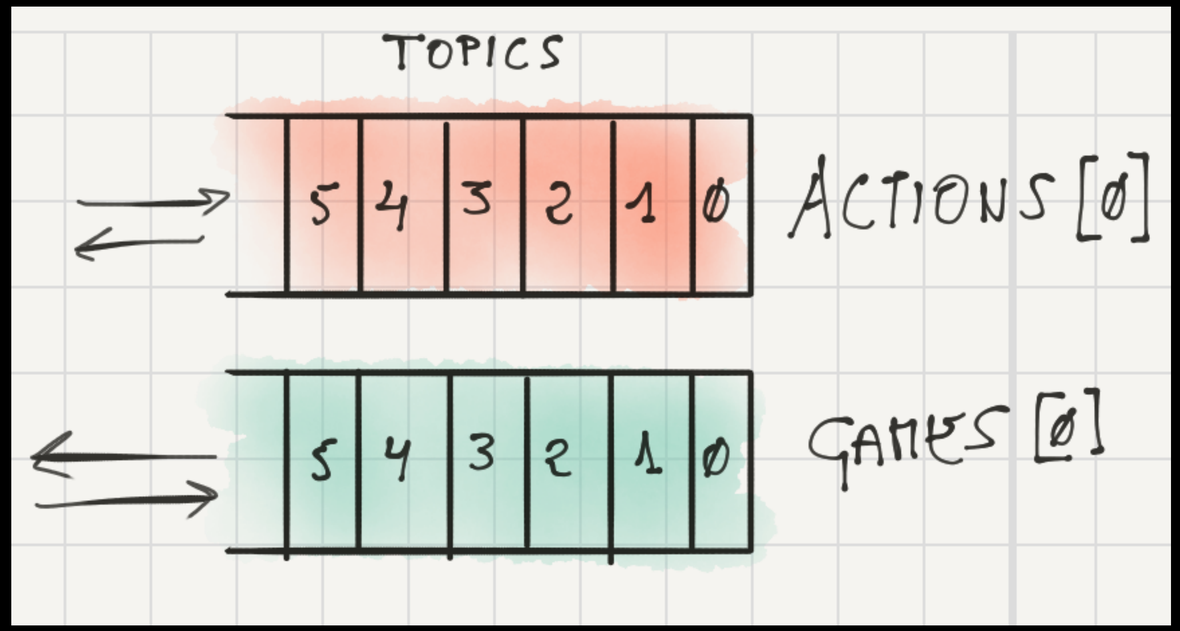
Topics, consumers, producers and stream processors

Kafka is a streaming platform based on a distributed commit log.

Events (or messages) are stored in topics (commit log files) in an ordered fashion. These events are defined as key-value pairs.

Topics are append only, and they are partitioned and replicated.

Every topic has at least 1 partition (depicted by the *[number]*).



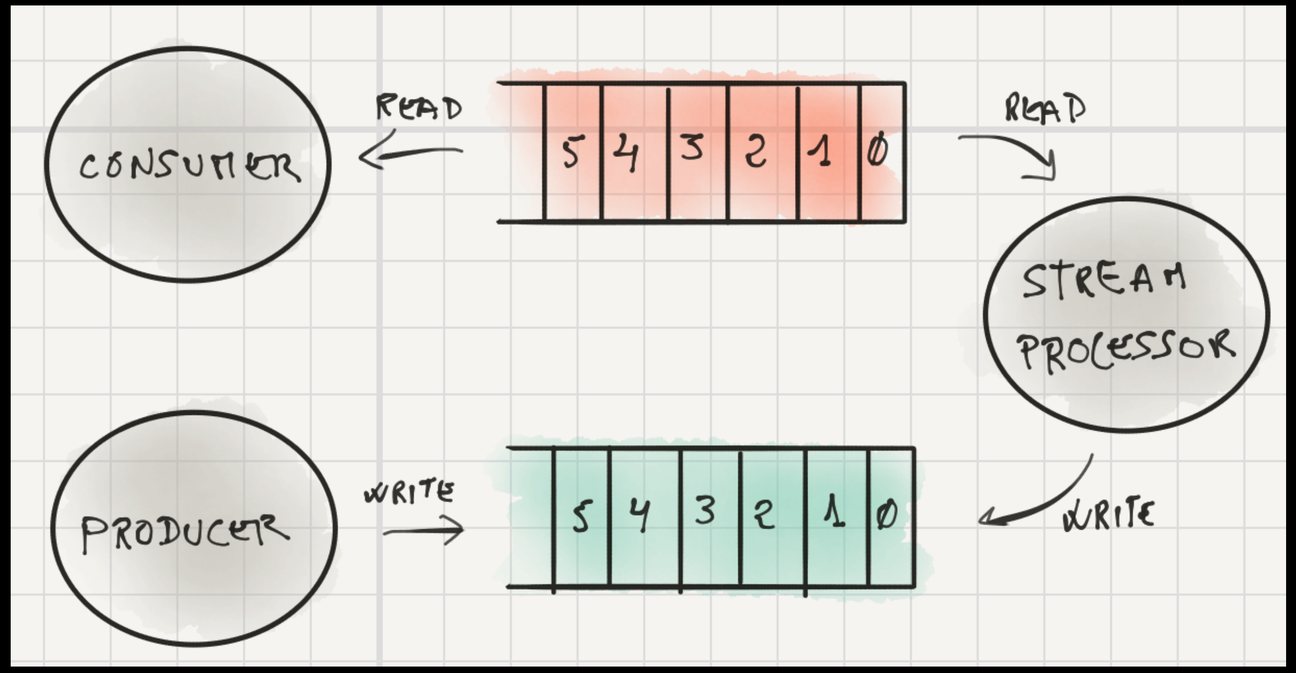
Kafka

Topics, consumers, producers and stream processors

In Kafka, besides of consumers and producers, we have Stream Processors. They use the Kafka Streams API and act as both consumers and producers.

There are many Kafka clients in many programming languages, but the Streams API is only available for JVM languages.

https://docs.confluent.io/platform/current/clients/index.html



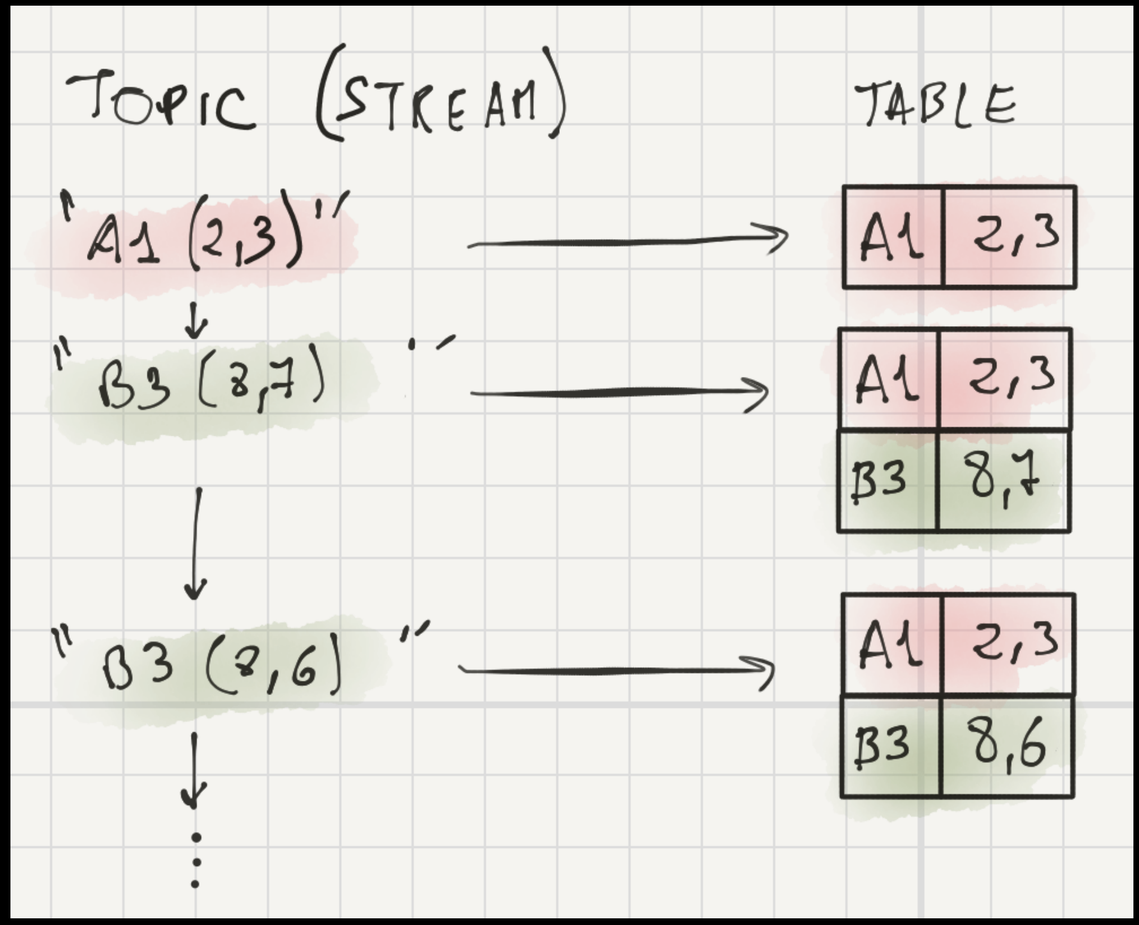
Kafka

State stores

Kafka Streams provides state stores that are used by streaming applications to store and retrieve data.

Streams can be seen as tables, and these are persisted state stores (RocksDB)

Any stateful operation performed in Kafka Streams makes use of state stores: *count()*, *aggregate()*, etc.



Client architecture

Client architecture

Sending players' actions to the server (producer)

The client is written in Go and we are using "goroutines" (i.e. threads) to handle:

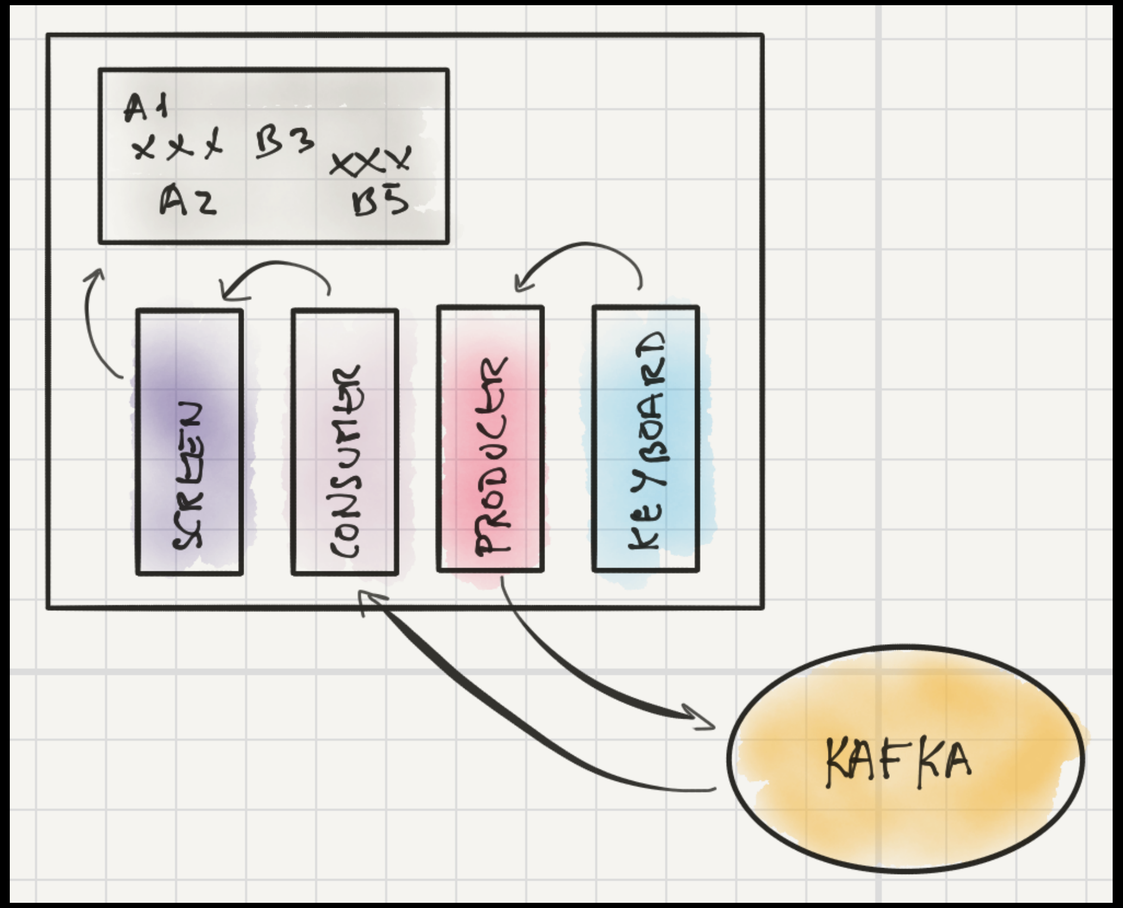
* screen rendering

* keyboard inputs

* receiving messages from kafka

* sending messages to kafka

These "goroutines" use "go channels" to communicate with each other

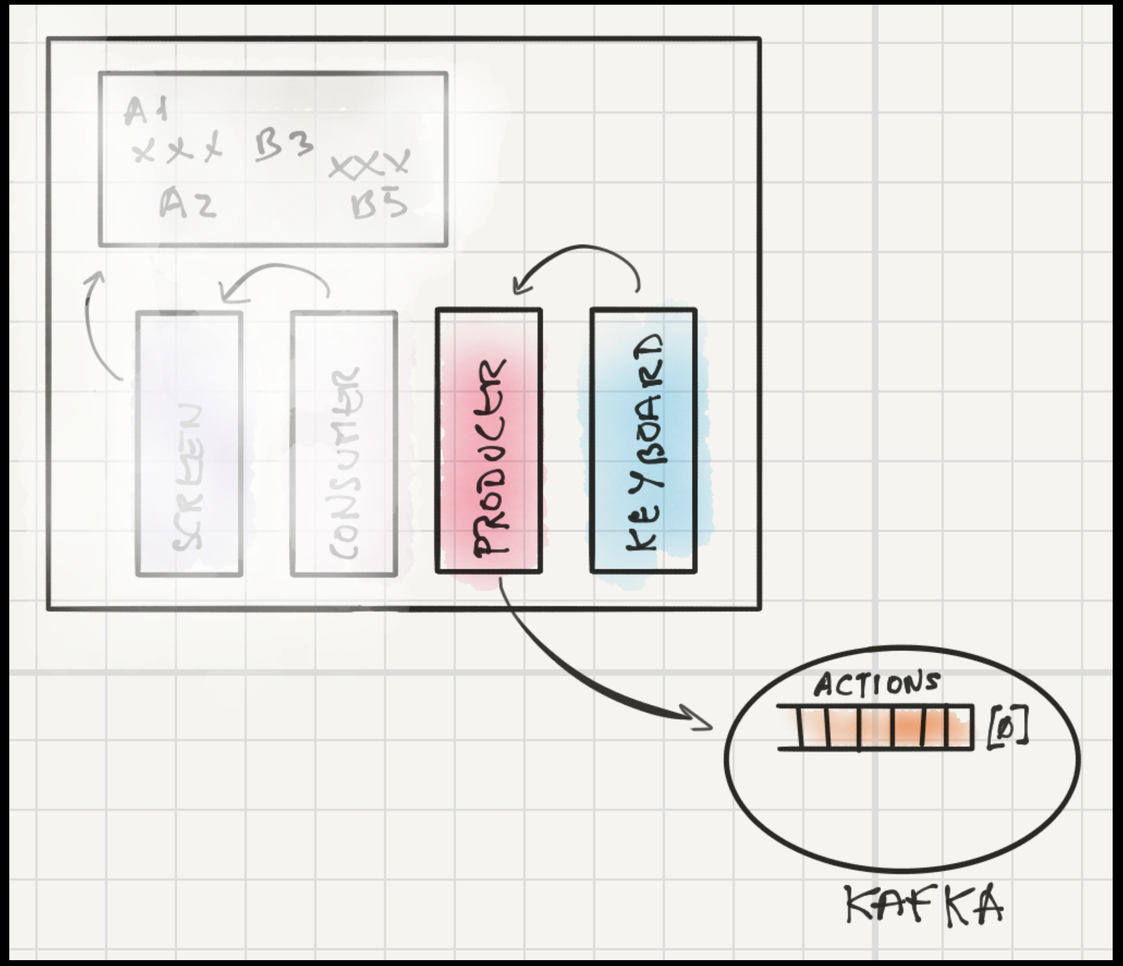


Client architecture

Sending players' actions to the server (producer)

Whenever there's a key pressed, the keyboard goroutine will check if it's a player action and send it to the producer.

The producer goroutine will send the action to a kafka topic called *actions*.



Let’s see some code: producer

Server architecture

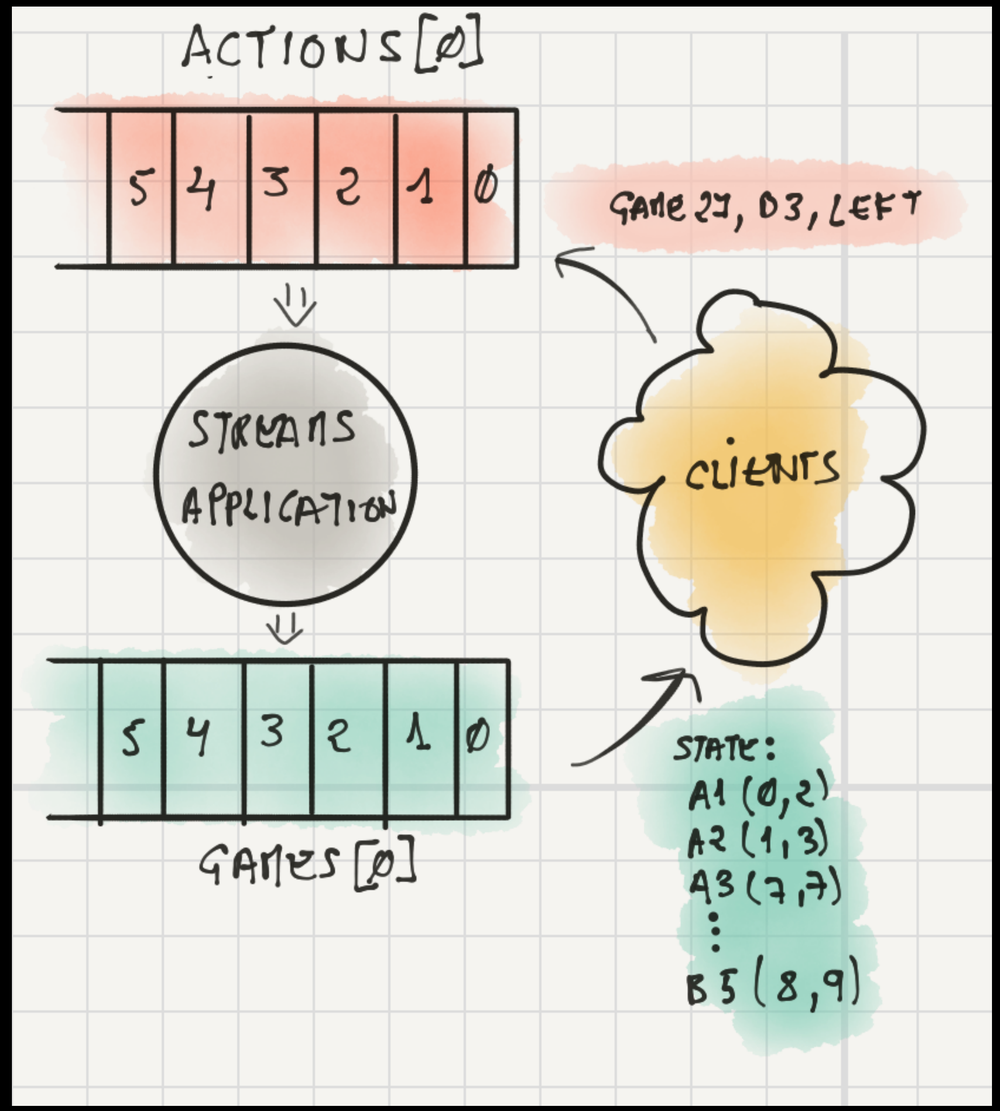
Server architecture

Kafka Streams

The server implementation will be a Kafka Streams application.

This application will receive players' actions through a topic called *actions*, it will validate them, and it will produce a new game state into a topic called *games*.

Kafka Streams applications can only be written in a JVM, and we'll use Kotlin.



Let’s see some code: streams application

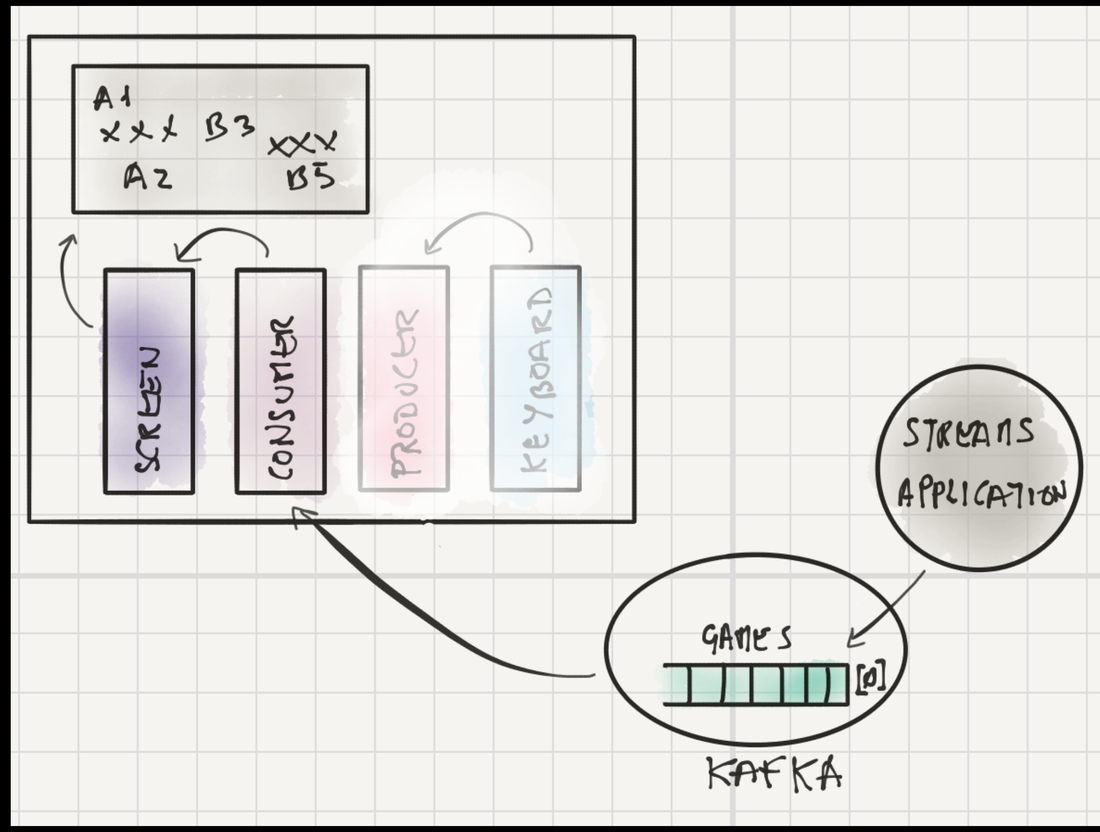
Client architecture

Client architecture

Receiving games' states from the server (consumer)

The streams application is sending the games' states into a topic called *games*.

Clients' consumer goroutine will consume from this topic and send a message to the screen goroutine that will render the new state in the screen



Let’s see some code: consumer

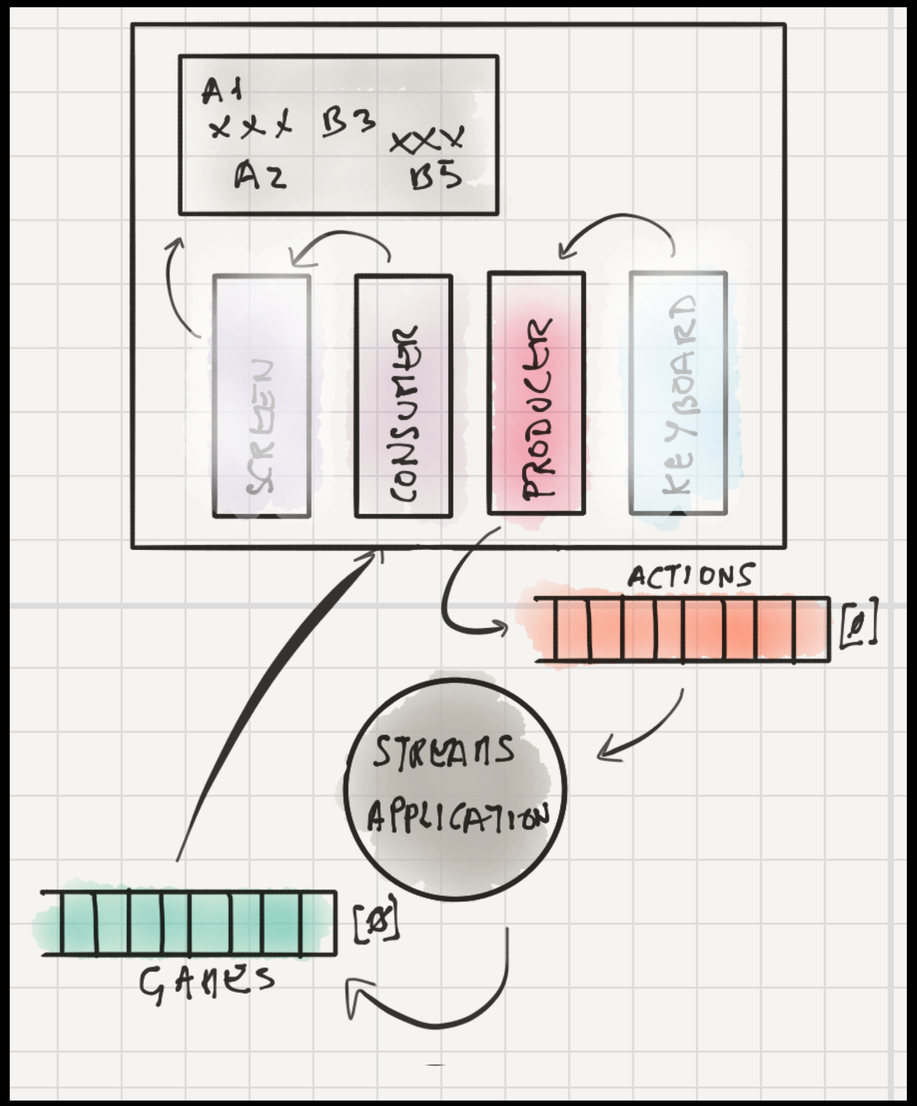
Game is a success: scaling the game

Scaling

Partitions over topics

The game is a success! 🎉 🎊

Many games are now being played at once, and having a single streams application is a bottle neck, so let's scale out.



Scaling

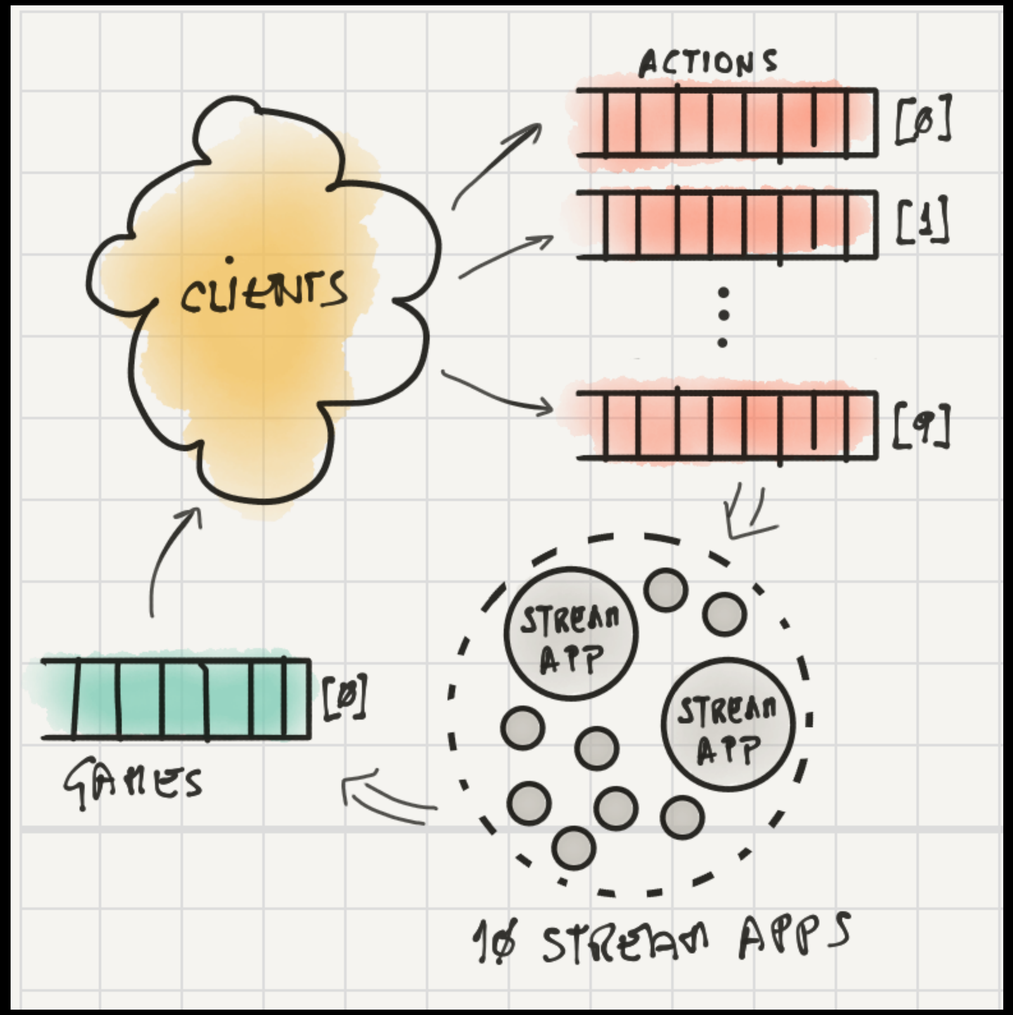
Partitions over topics

Streams applications scale using partitions, so let's create a new topic called *actions.partitions* with 10 partitions.

Then, let's spin up 10 instances of the streams application and Kafka will assign one *actions.partitions* partition to each stream application instance(*).

All of these stream applications will write to the single *games* topic partition

(*) https://blog.cloudera.com/scalability-of-kafka-messaging-using-consumer-groups/

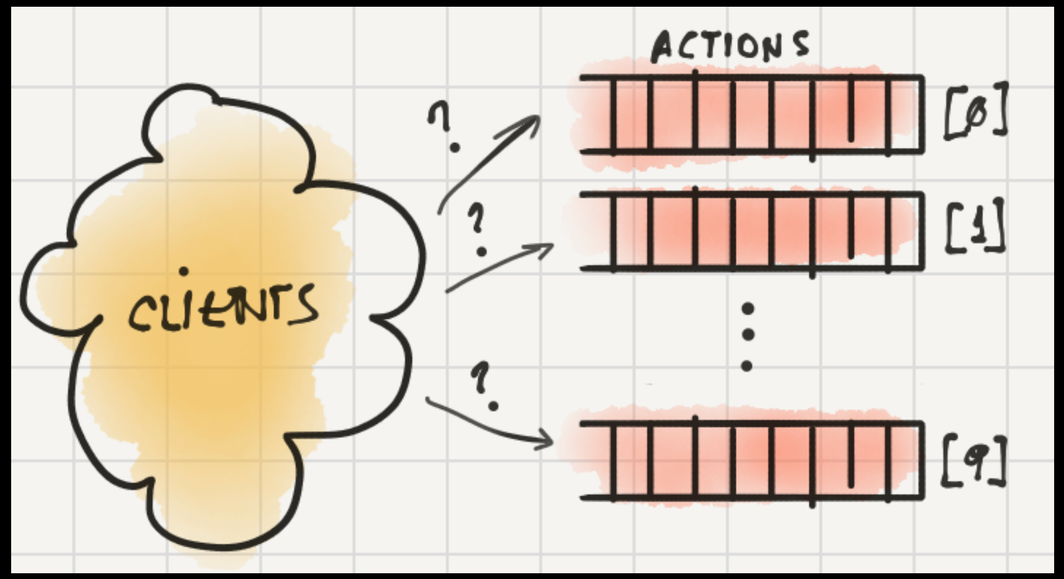


Scaling

Producers partition strategy

Clients should now decide to which *actions.partitions* partition they should write player's actions.

Kafka also takes care of that 😎 and it's configurable(*). By default, it hashes the key with murmur2 algorithm and divides by the amount of partitions. Same key is always assigned to the same partition.



(*)https://docs.confluent.io/5.0.0/clients/librdkafka/CONFIGURATION_8md.html

https://docs.confluent.io/platform/current/clients/producer.html#ak-producer-configuration

Let’s see some code: streams application

Scaling

Partitions over topics

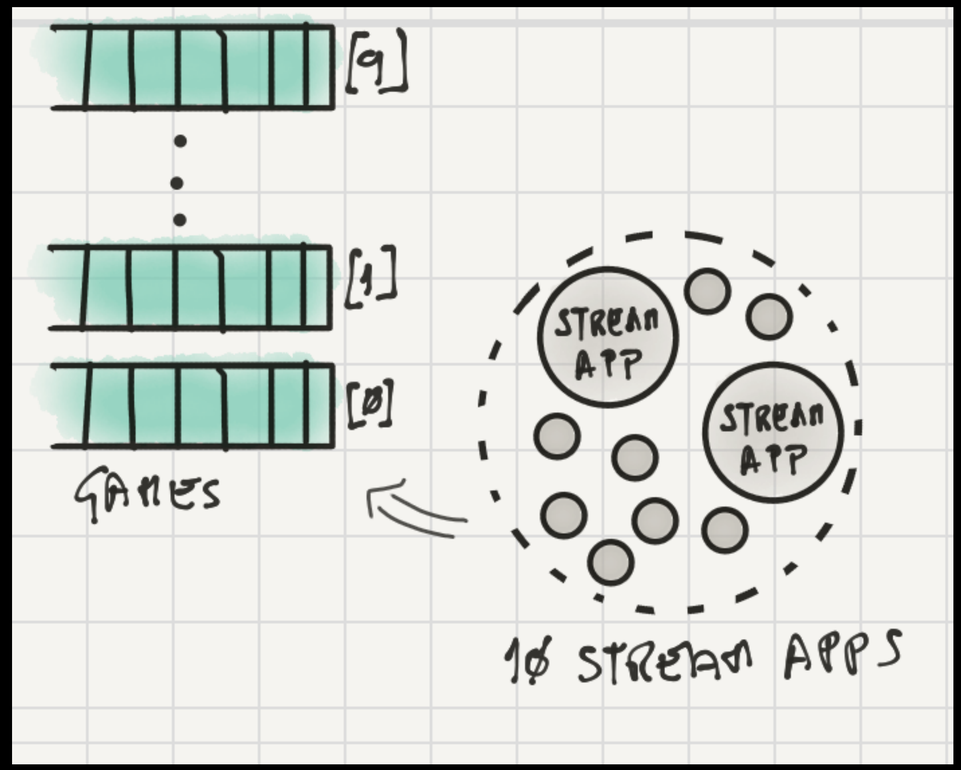
A single *games* topic partition causes that clients receive messages that they are not interested.

Two options:

a) to have a *games* topic per game

b) to have many partitions for the *games* topic

Let’s do b) so we can see how to route messages to partitions.



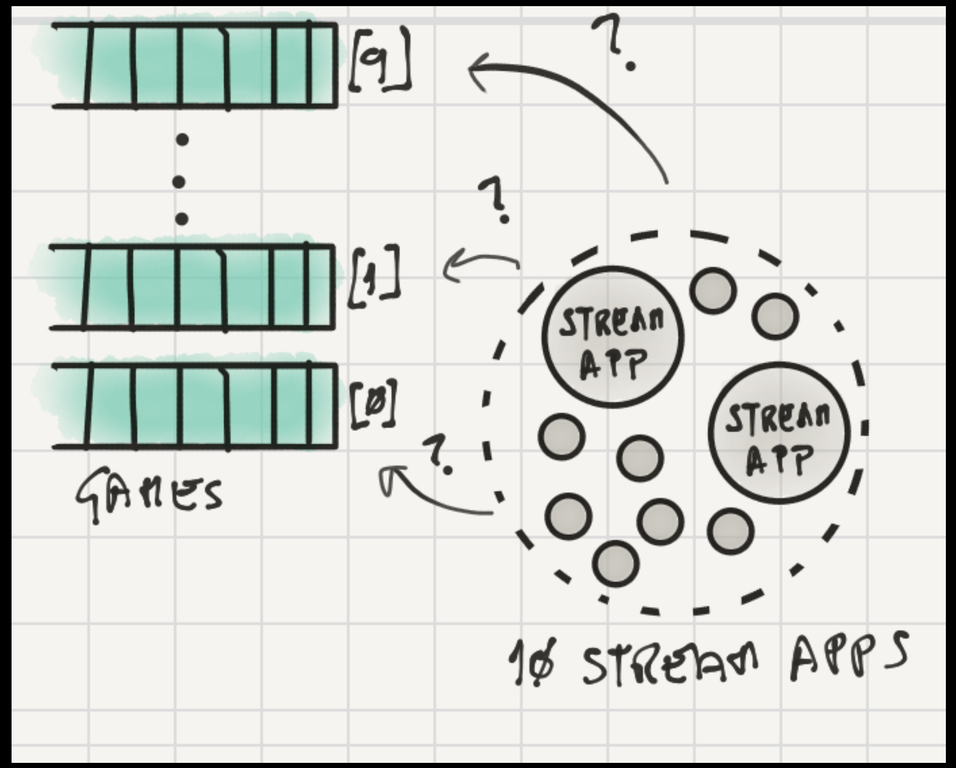
Scaling

Partitions over topics

We want to always route all messages from a specific game to then same partition, so clients can subscribe to a single partition.

A simple routing algorithm:

partition = game_id mod num_partitions



Let’s see some code: partition routing

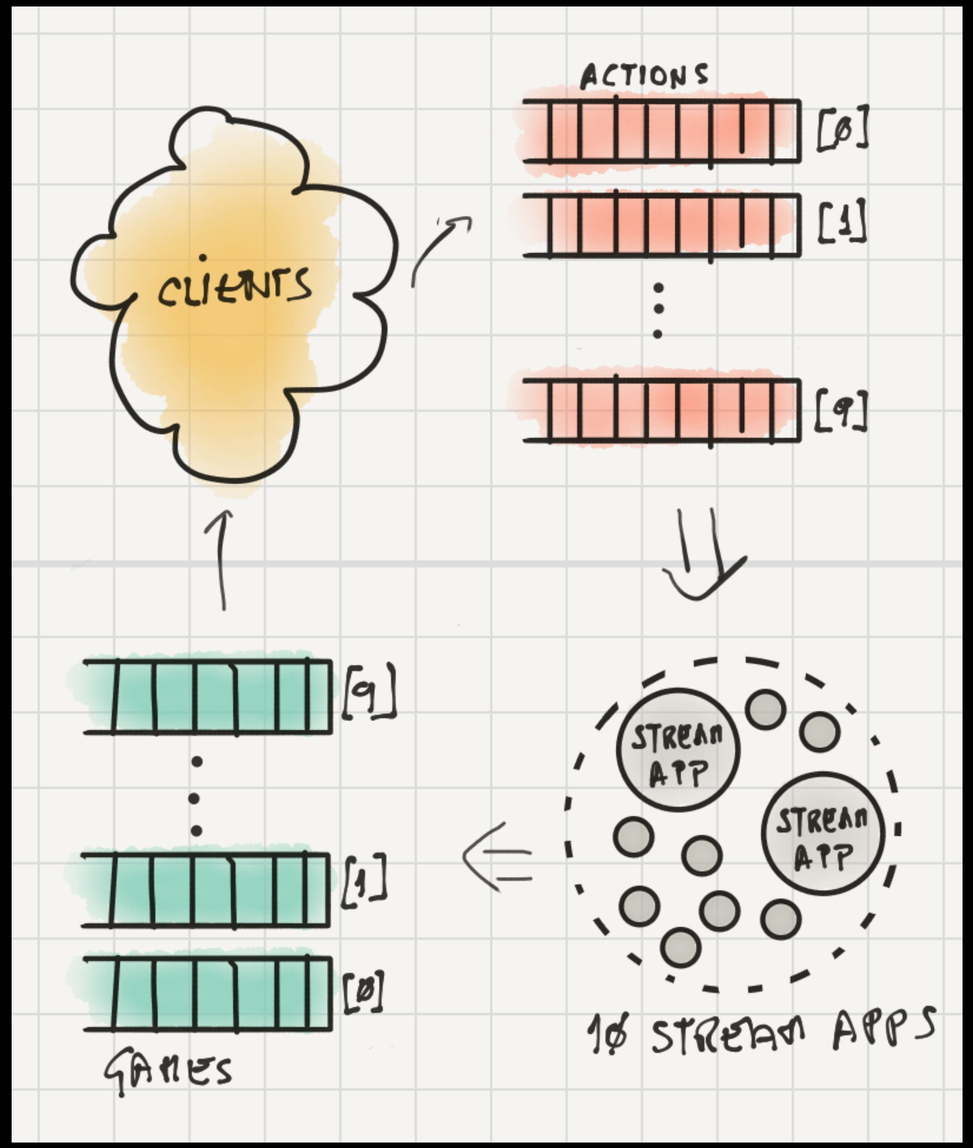
Scaling

Consume from a specific partition

Final Kafka architecture: now clients write their players' actions to a specific *actions* partitions, and read their game status from a specific *games* partition.

We will use the same algorithm to determine from which *games* partition a client should read its game statuses:

partition = game_id mod num_partitions



Let’s see some code: reading from a specific partition

Bonus: exposing games through a RESTful API

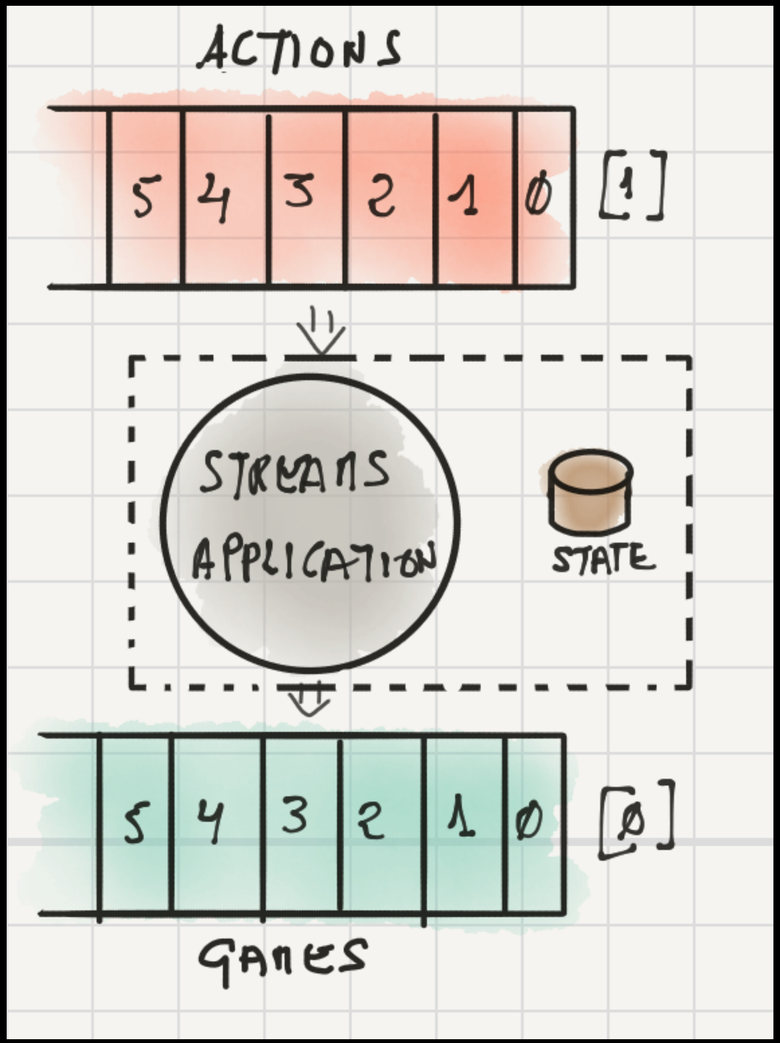
Bonus

Interactive queries

As mentioned before, stateful operations in Kafka Streams make use of state stores.

Our topic *games* is stored in a state store.

Kafka Streams provides an API to query these stores: interactive queries.



Let’s see some code: interactive queries