COMP2511

🧭 8.2 - Event-Driven & Asynchronous Design, Part 2

Asynchronous Paradigm

What is asynchronous programming?

- Asynchronous programming, broadly speaking is any form of programming where processes are not necessarily sequentially executed
- Different forms of asynchronous implementation
  - Event loops (seen commonly in JavaScript, which is single-threaded)
  - Parallelism (multi-threading)
- We will focus mainly on parallelism in this course
- Parallelism vs Concurrency:
  - **Parallelism**: The simultaneous execution of computations
  - **Concurrency**: The art of managing parallelism

Asynchronous Paradigm

Why program asynchronously?

- Performance - sequential execution is too slow
- We don't want programs to become bottlenecked on **blocking** operations (e.g. reading a file, opening a web socket)
- Requirements to have real-time updates (e.g. instant messaging)
- Allows for better software design (we'll see soon)

Asynchronous Paradigm

In Week 7 we discussed concurrency, synchronisation and the Singleton Pattern - ways of managing the issues arise when programming asynchronously.

Today, we'll discuss applications of asynchronous programming in software design.

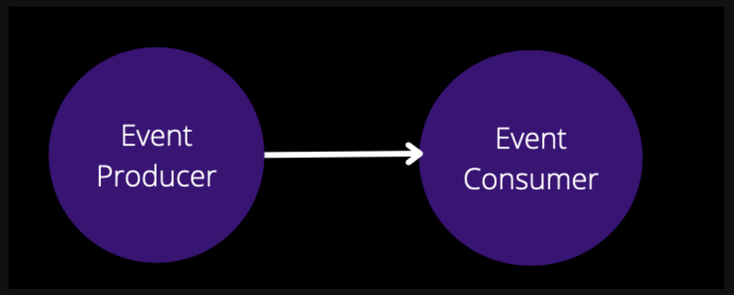
The primary application is Event-Driven Programming and the Observer Pattern.

Promises

https://www.digitalocean.com/community/tutorials/understanding-the-event-loop-callbacks-promises-and-async-await-in-javascript

Event-Driven Revisited

- The Observer Pattern - way of *propagating changes in state* between software components
- **Producers** (AKA subjects) produce events
- **Consumers** (AKA subscribers, observers, listeners) consume events
- Producer/Consumer relationships can be 1:1, 1:M, M:1, M:M
- But first, some background



1. The Event-Driven Paradigm

- Event: A change in state
- In the event-driven paradigm, events are first class citizens

```
1 const element = document.getElementById("myBtn");
2 element.addEventListener("click", myFunction);
3
4 function myFunction(event) {
5         console.log(event.target.value)
6 }
```

2. State as a Series of Events

- We can think of the current state of a system to be *the result of playing out all the events that have occurred*
- Where have we seen this before?
  - In the project?
  - In your programming experience?

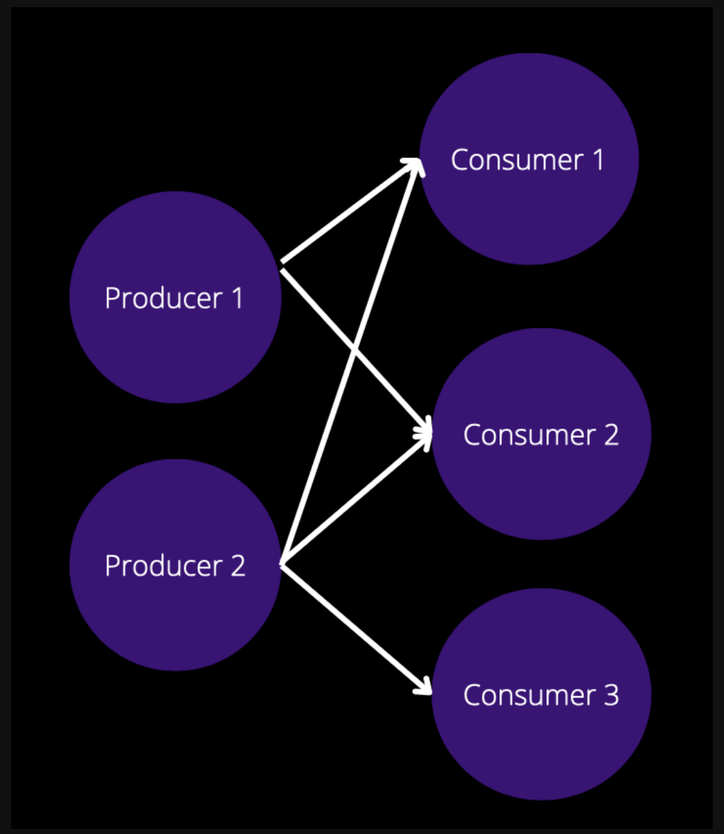
3. Streams Revisited

```
1 List<Integer> ints =
2     strings2.stream()
3             .map(Integer::parseInt)
4             .collect(Collectors.toList());
```

- Minimise coupling by letting data "flow down the stream"

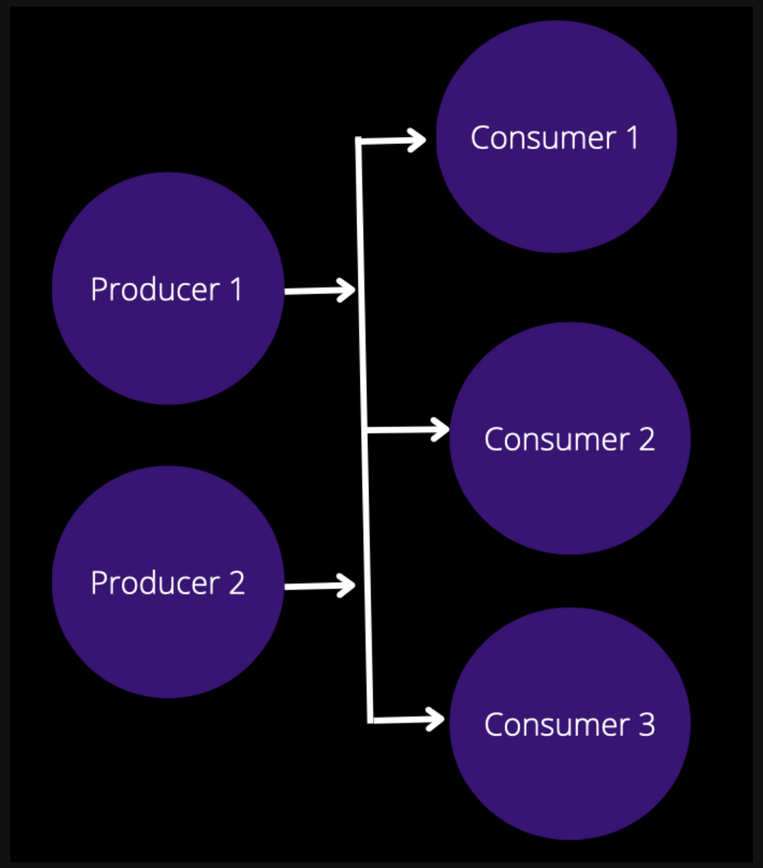
Problems with the Synchronous Observer Pattern

- Problem 1 - What if while we are processing the first event, the second event comes along?
- Problem 2 - What if we start to increase the number of relationships:
  - More producers
  - More consumers
  - More types of events
- Leads to high coupling
- The postman problem



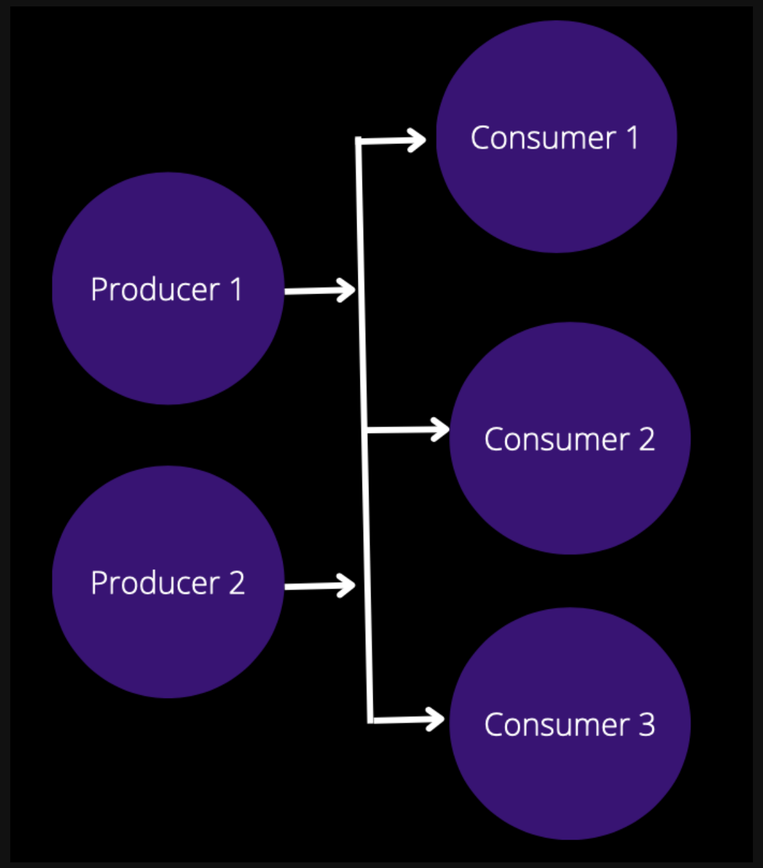
Asynchronous Observer Pattern

- Instead of producers and consumers being directly coupled, they instead communicate over a **shared channel**
- Producers deposit events into the channel
- Events "flow through the channel"
- Consumers read from the shared channel and process the events
- Producers and consumers are all run in parallel



Asynchronous Observer Pattern

- What if a consumer's not interested?
  - Can choose to consume certain types of events, and ignore others
- Good design
  - Principle of Least Knowledge
  - No coupling between producers and consumers
  - Can add/remove producers or consumers at will

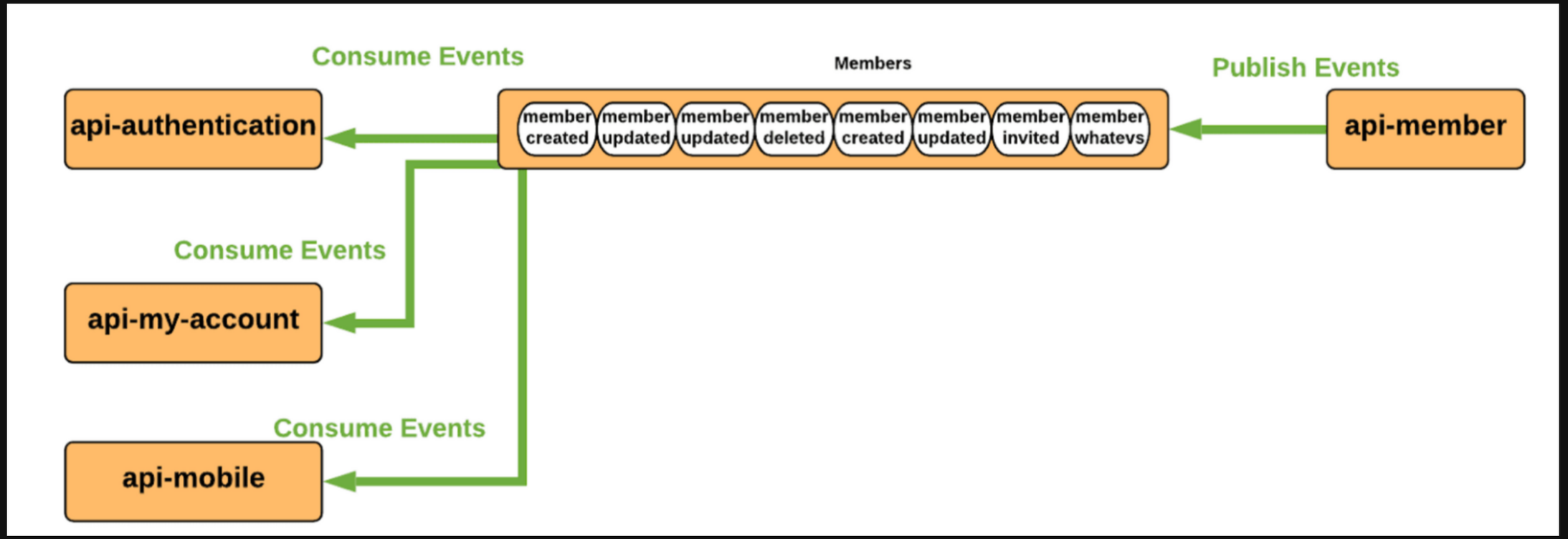


Implementation: Go Channels

- Go provides very good lightweight support for this design pattern with **goroutines** (you can think of them as lightweight threads) and **channels** (essentially a buffer)
  - https://go.dev/play/p/MZo57Ei1NKW
- Need to consider blocking of goroutines on channels and problems of:
  - Deadlock
  - Starvation

Implementation: Apache Kafka

- Communication between repositories or APIs in a service ecosystem



Event Pipelines

- We can "play" events through a pipeline using this model
- What if we wanted to **play back** events?
  - Process interrupted
  - Data malformed
  - Fix up and replay the events
- Need to design consumers for **idempotency** - what if they consume the same event twice?
- Need to determine if consumers **rely on a particular ordering** of events to be processed in

Final Thoughts

- It all just seems like glue?
- Trends of industry and modern software architecture
  - Microservices and abstraction
  - The need for event driven systems
- The role of design patterns in real-world software