

Elegant reactive streams with coroutines



Claim

Reactive streams are hard to implement and error prone as a result of the poor concurrency primitives provided by the JVM.

Part 1: Theory

Hard to implement

Backpressure

```
import ...  
  
/**  
 * Maps the values of the source publisher one-on-one via a mapper function.  
 *  
 * @param <T> the source value type  
 * @param <R> the result value type  
 *  
 * @see <a href="https://github.com/reactor/reactive-streams-commons">Reactive-Streams-Commons</a>  
 */  
final class FluxMap<T, R> extends FluxOperator<T, R> {  
  
    final Function<? super T, ? extends R> mapper;  
  
    /**  
     * Constructs a FluxMap instance with the given source and mapper.  
     *  
     * @param source the source Publisher instance  
     * @param mapper the mapper function  
     *  
     * @throws NullPointerException if either {@code source} or {@code mapper} is null.  
     */  
    FluxMap(Flux<? extends T> source,  
           Function<? super T, ? extends R> mapper) {  
        super(source);  
        this.mapper = Objects.requireNonNull(mapper, "mapper");  
    }  
  
    @Override  
    @unchecked  
    public void subscribe(CoreSubscriber<? super R> actual) {  
        if (actual instanceof Fuseable.ConditionalSubscriber) {  
            Fuseable.ConditionalSubscriber<? super R> cs =  
                (Fuseable.ConditionalSubscriber<? super R>) actual;  
            source.subscribe(new MapConditionalSubscriber<>(cs, mapper));  
            return;  
        }  
        matthisk.com  
        source.subscribe(new MapSubscriber<>(actual, mapper));  
    }  
}
```

Error prone¹

¹<https://medium.com/jeroen-rosenberg/10-pitfalls-in-reactive-programming-de5fe042dfc6>

Poor concurrency primitives

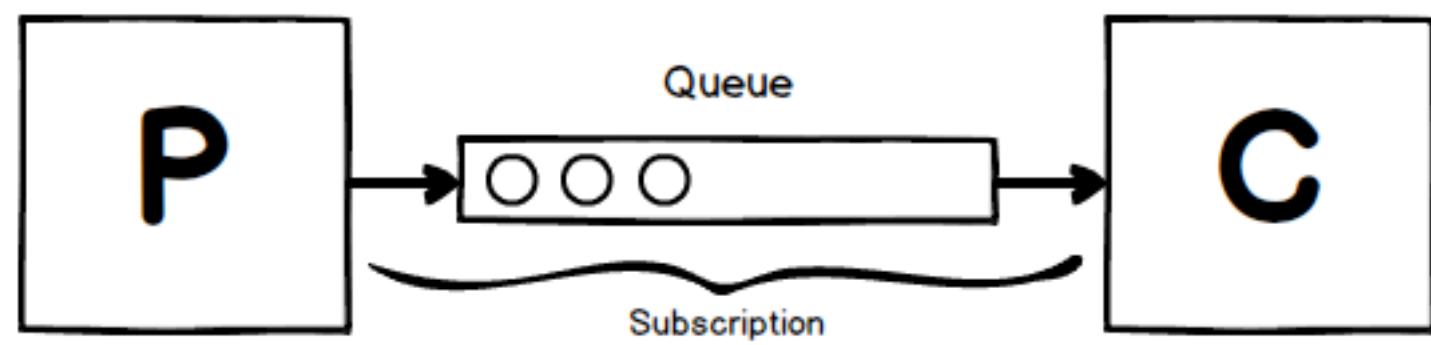
1. Threads

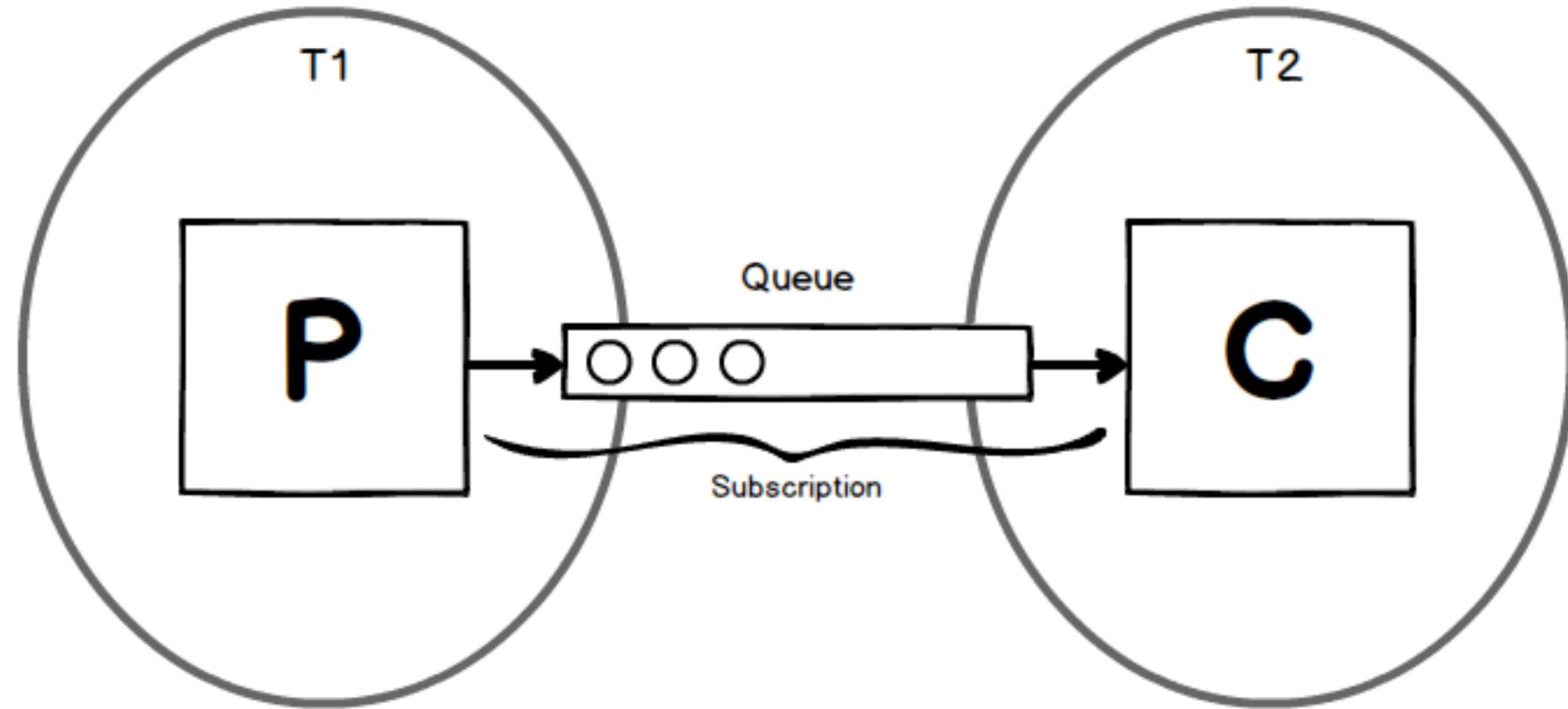
→ Expensive!!!

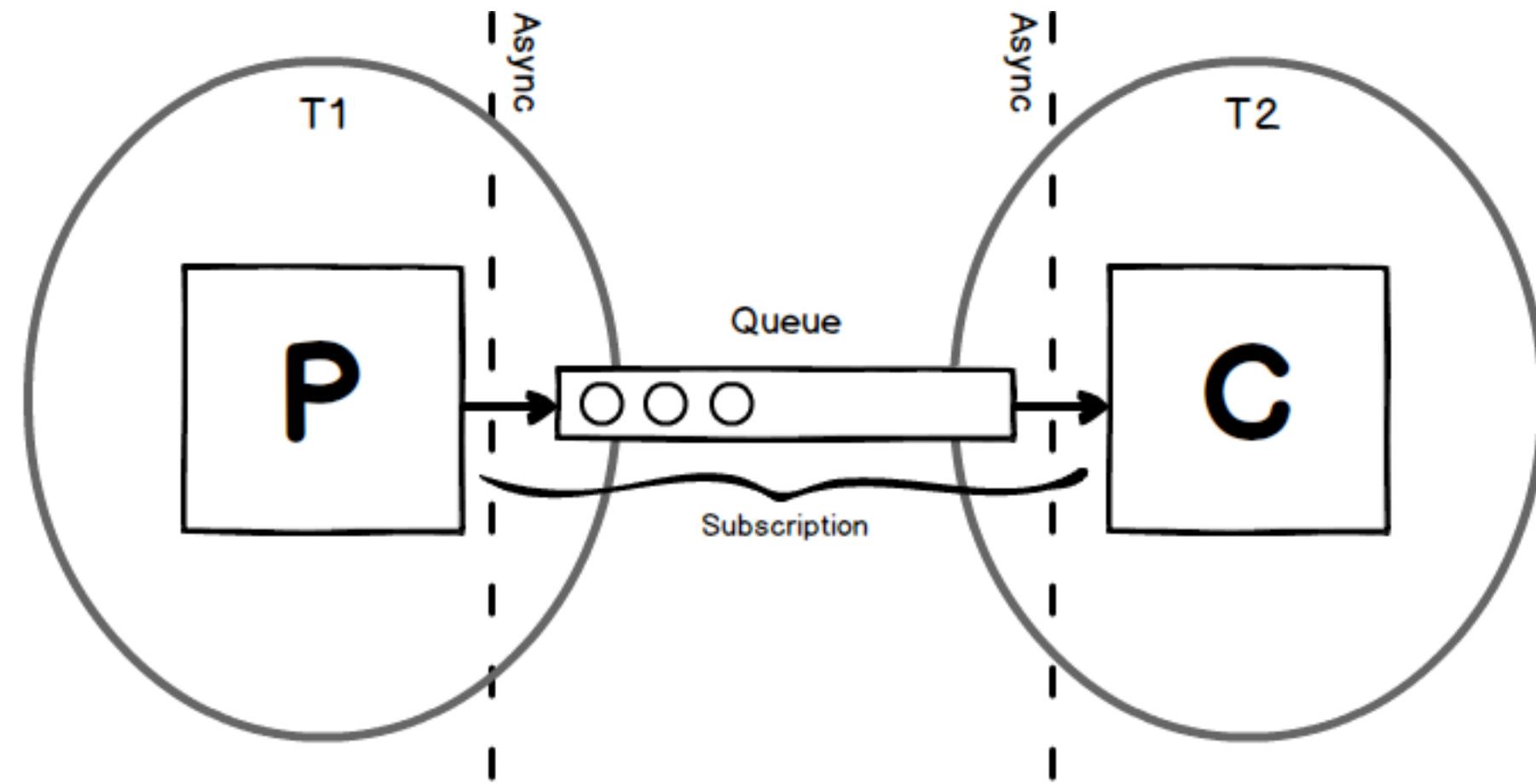
2. Callbacks

→ Backpressure

Why do we need backpressure?







```
// Asynchronous Producer
Flux.create(emitter -> {
    emitter.next("hello");
    // <- How do we suspend our thread here??? ->
    emitter.next("world");
})
// Thread boundary
.publishOn(Schedulers.elastic())
// Consumer
.subscribe(System.out::println);
```

```
// Asynchronous Producer
Flux.create(emitter -> {
    emitter.next("hello");
    // <- How do we suspend our thread here??? ->
    emitter.next("world");
})
// Thread boundary
.publishOn(Schedulers.elastic())
// Consumer
.subscribe(System.out::println);
```

What if there existed a concurrency primitive that allowed us to implement a queue on which operations do not block.

Channels

Channels

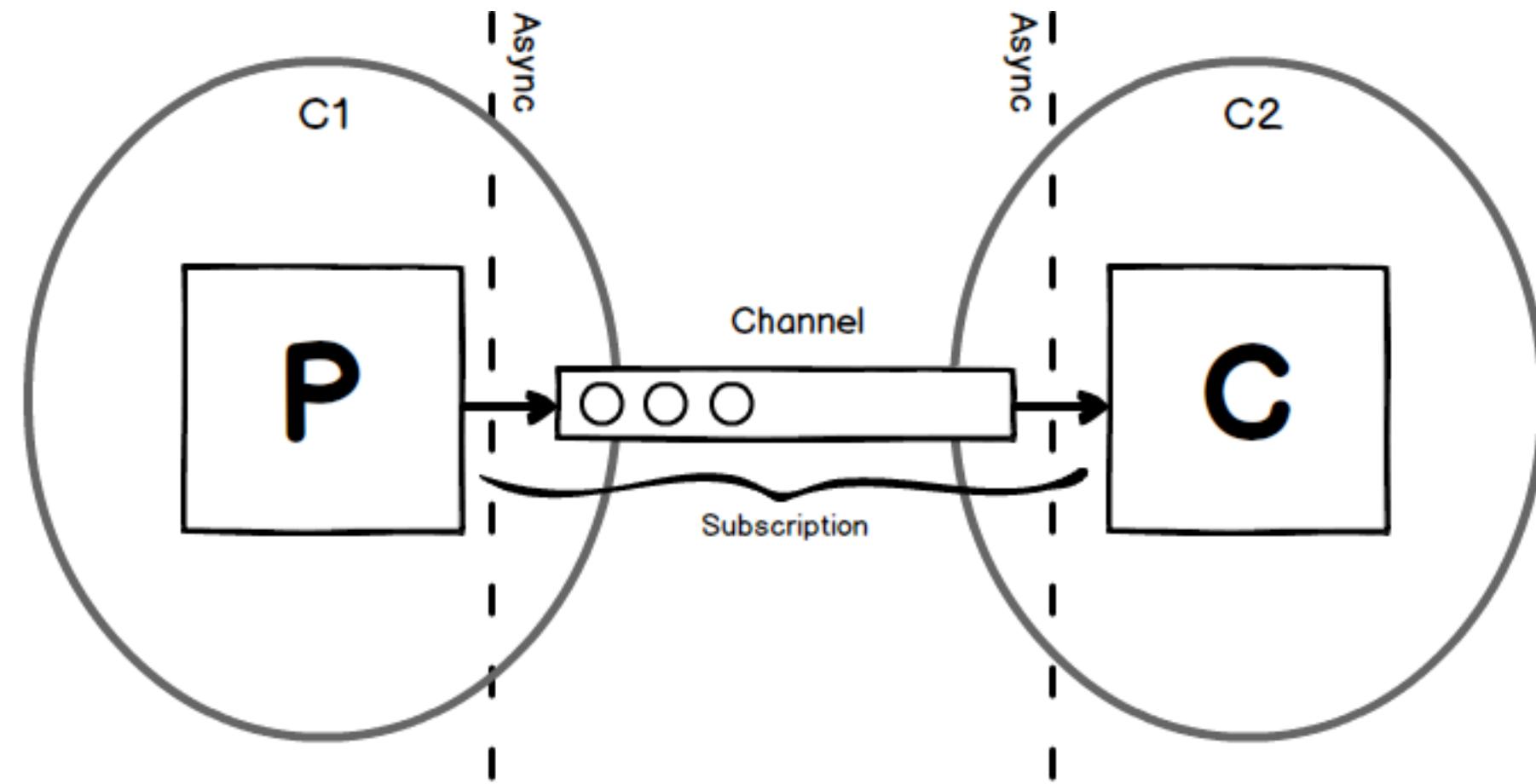
A Channel is conceptually similar to BlockingQueue.
Where instead of a blocking put it has a suspending
send, and instead of a blocking take it has a suspending
receive.

What are we suspending here?

Coroutines

Coroutines are like light-weight threads, that allow us to *yield* and *re-enter* methods

```
suspend fun enterExit() {  
    println("hello")  
    delay(500) // suspend  
    println("world") // re-enter  
}
```



Part 2: Kotlin Flow

Kotlin Flow

```
interface Flow<out T> {  
    suspend fun collect(collector: FlowCollector<T>)  
}
```

Kotlin Flow

```
interface Flow<out T> {  
    suspend fun collect(collector: FlowCollector<T>)  
}
```

```
interface FlowCollector<in T> {  
    suspend fun emit(value: T)  
}
```

Creating a Flow

```
fun <T> flow(block: suspend FlowCollector<T>.() -> Unit): Flow<T>
```

Creating a Flow

```
fun <T> flow(block: suspend FlowCollector<T>.() -> Unit): Flow<T>

fun main() = flow {
    emit("hello")
    delay(100) // 🎉 Suspend the producer without blocking
    emit("world")
}
```

Creating a Flux

```
Flux.from(subscriber -> {
    subscriber.onSubscribe(new Subscription() {
        @Override
        public void request(long n) {
            // Send `n` items to `subscriber.onNext`
        }
    });
});
```

Using a Flow

```
// Asynchronous Producer
flow {
    emit("hello")
    // <- How do we suspend our coroutine here??? ->
    emit("world")
}
// Thread boundary
.flowOn(Dispatchers.IO)
// Consumer
.collect { println(it) }
```

Using a Flow

```
// Asynchronous Producer
flow {
    emit("hello")
    delay(100)
    emit("world")
}
// Thread boundary
.flowOn(Dispatchers.IO)
// Consumer
.collect { println(it) }
```

Operators

```
fun <A, B> Flow<A>.map(transform: suspend (value: A) -> B) = flow {  
    collect { emit(transform(it)) }  
}
```

Operators

```
import ...

/**
 * Maps the values of the source publisher one-on-one via a mapper function.
 *
 * @param <T> the source value type
 * @param <R> the result value type
 *
 * @see <a href="https://github.com/reactor/reactive-streams-commons">Reactive-Streams-Commons</a>
 */
final class FluxMap<T, R> extends FluxOperator<T, R> {
    fun <A, B> Flow<A>.map(transform: suspend (value: A) -> B) = flow {
        final Function<? super T, ? extends R> mapper;
        collect { emit(transform(it)) }

    /**
     * Constructs a FluxMap instance with the given source and mapper.
     *
     * @param source the source Publisher instance
     * @param mapper the mapper function
     *
     * @throws NullPointerException if either {@code source} or {@code mapper} is null.
     */
    FluxMap(Flux<? extends T> source,
           Function<? super T, ? extends R> mapper) {
        super(source);
        this.mapper = Objects.requireNonNull(mapper, message: "mapper");
    }

    @Override
    @unchecked
    public void subscribe(CoreSubscriber<? super R> actual) {
        if (actual instanceof Fuseable.ConditionalSubscriber) {
            Fuseable.ConditionalSubscriber<? super R> cs =
                (Fuseable.ConditionalSubscriber<? super R>) actual;
            source.subscribe(new MapConditionalSubscriber<>(cs, mapper));
            return;
        }
        matthisk.com
        source.subscribe(new MapSubscriber<>(actual, mapper));
    }
}
```

What have we learned

- Having to account for backpressure is not something inherent to async stream processing
- Poor concurrency primitives make for complexer application level code

References

- blocking threads suspending coroutines
- kotlin flows and coroutines
- simple design of kotlin flow
- cold flows hot channels