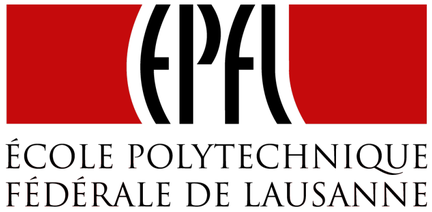


Fan-in and Fan-out operations

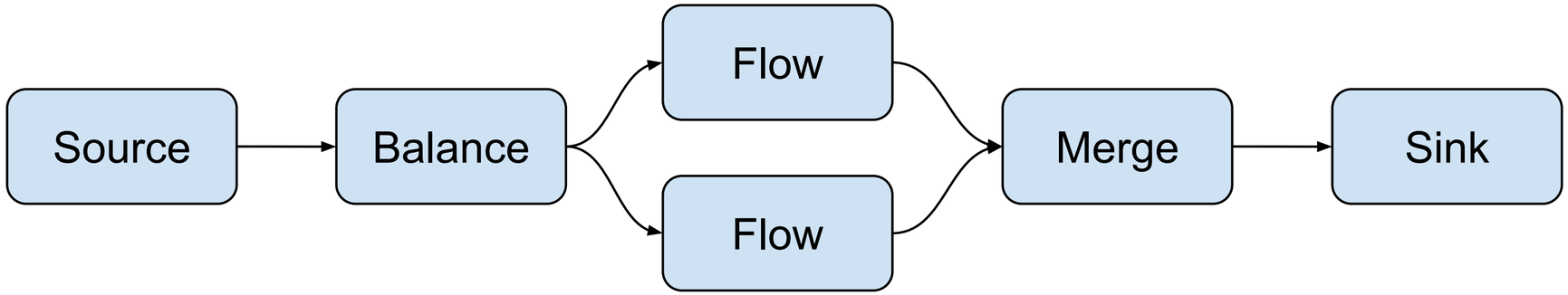
Programming Reactive Systems

Konrad Malawski, Julien Richard-Foy

Splitting and merging streams

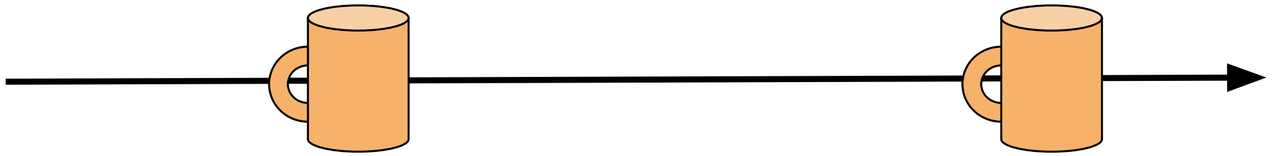
Sources, flows and sinks only allow sequential pipelines of transformations.

- *Fan-in* operations are operations which have multiple *input* ports
- *Fan-out* operations are operations which have multiple *output* ports

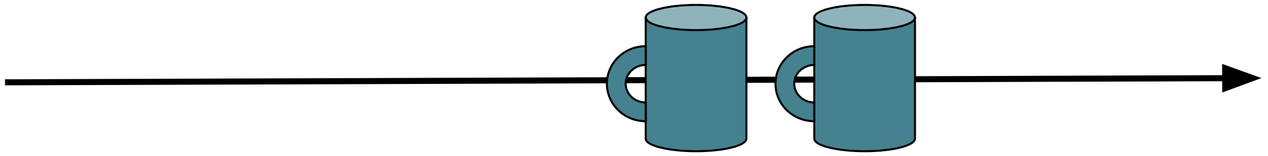


Example of static fan-in operator: `Merge`

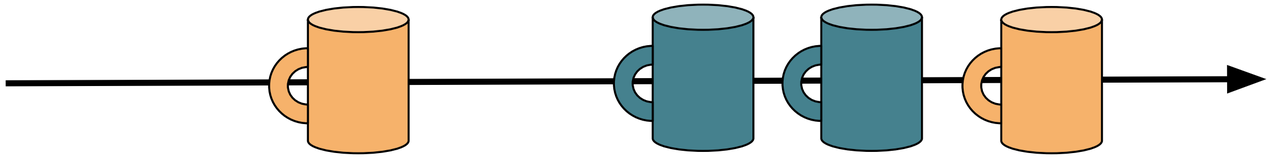
`mokaPot`



`frenchPress`



```
mokaPot.merge(frenchPress)
```

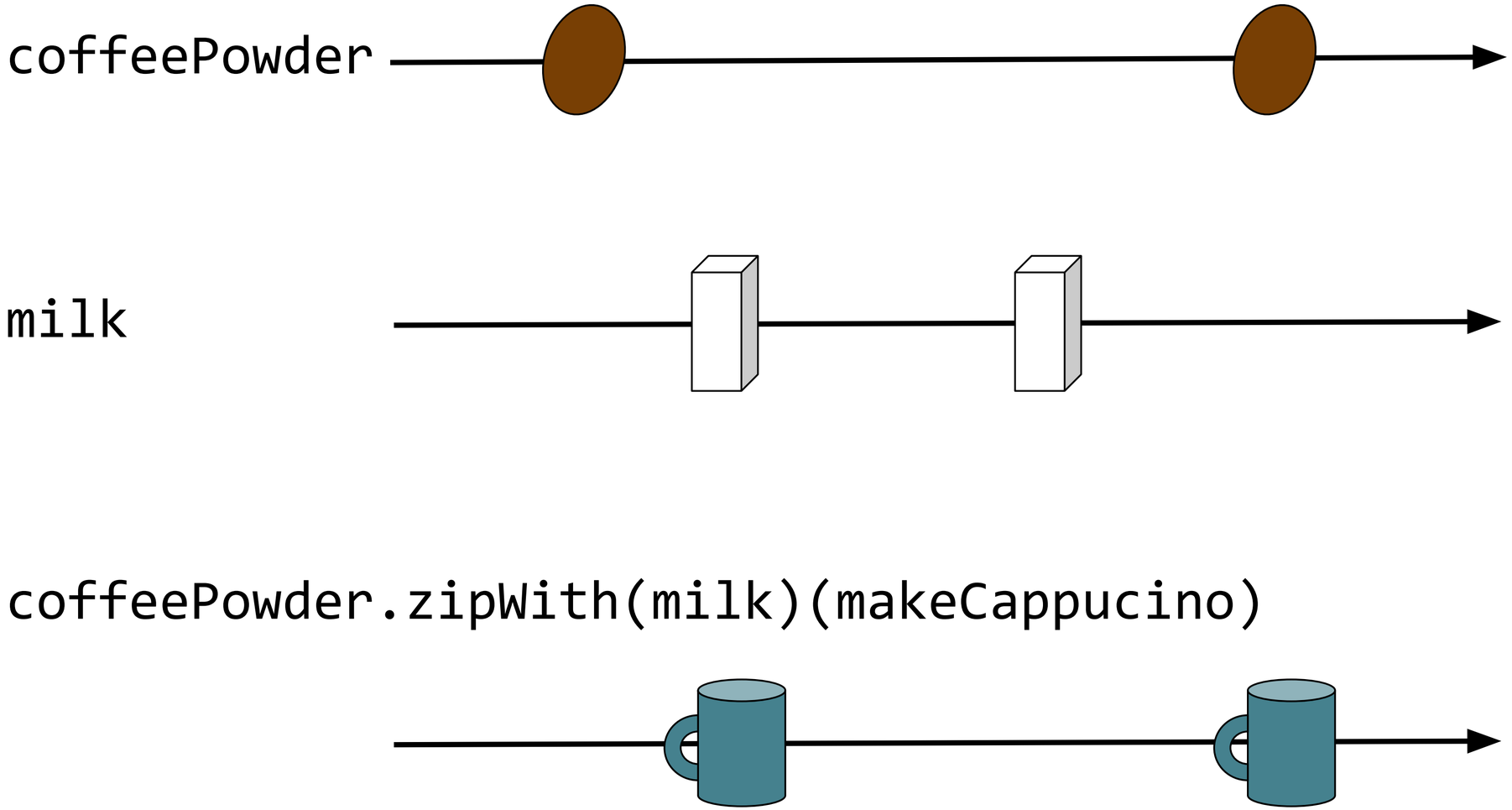


Example of static fan-in operator: `Merge` (2)

```
def coffees(
  mokaPot: Source[Coffee, _],
  frenchPress: Source[Coffee, _]
): Source[Coffee, _] =
  mokaPot.merge(frenchPress)
```

We produce coffees by taking them as they arrive from each brewing system.

Example of static fan-in operator: `ZipWith`



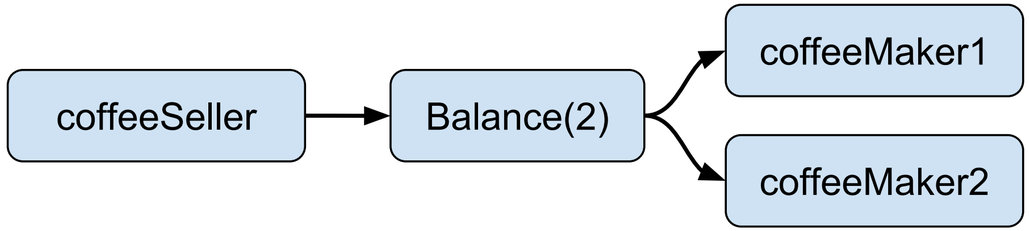
Example of static fan-in operator: `ZipWith` (2)

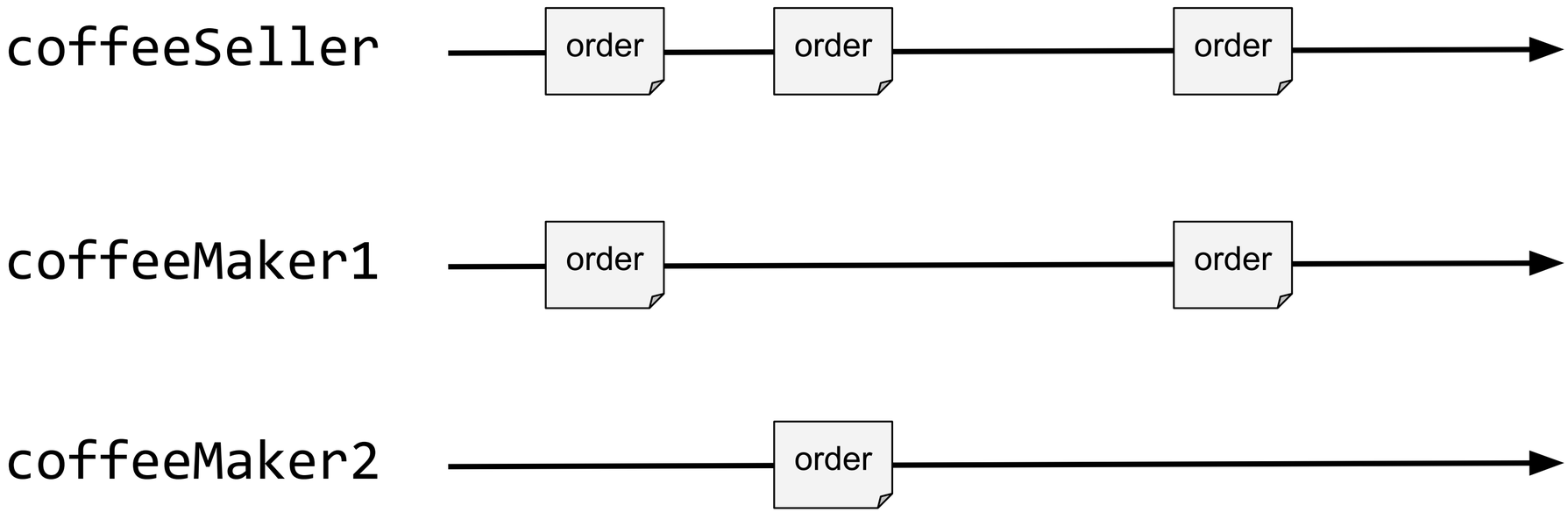
```
def cappuccinos(
  coffeePowderSource: Source[CoffeePowder, _],
  milkSource: Source[Milk, _]
): Source[Cappuccino, _] =
  coffeePowderSource.zipWith(milkSource) {
    case (coffeePowder, milk) => makeCappucino(coffeePowder, milk)
  }
```

We produce *cappuccinos* by taking one item of each source (coffee powder and milk) and by combining them.

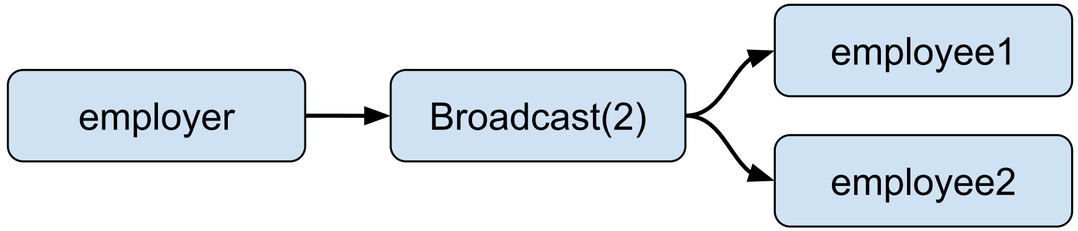
- What happens if the milk source is faster than the coffee powder source?

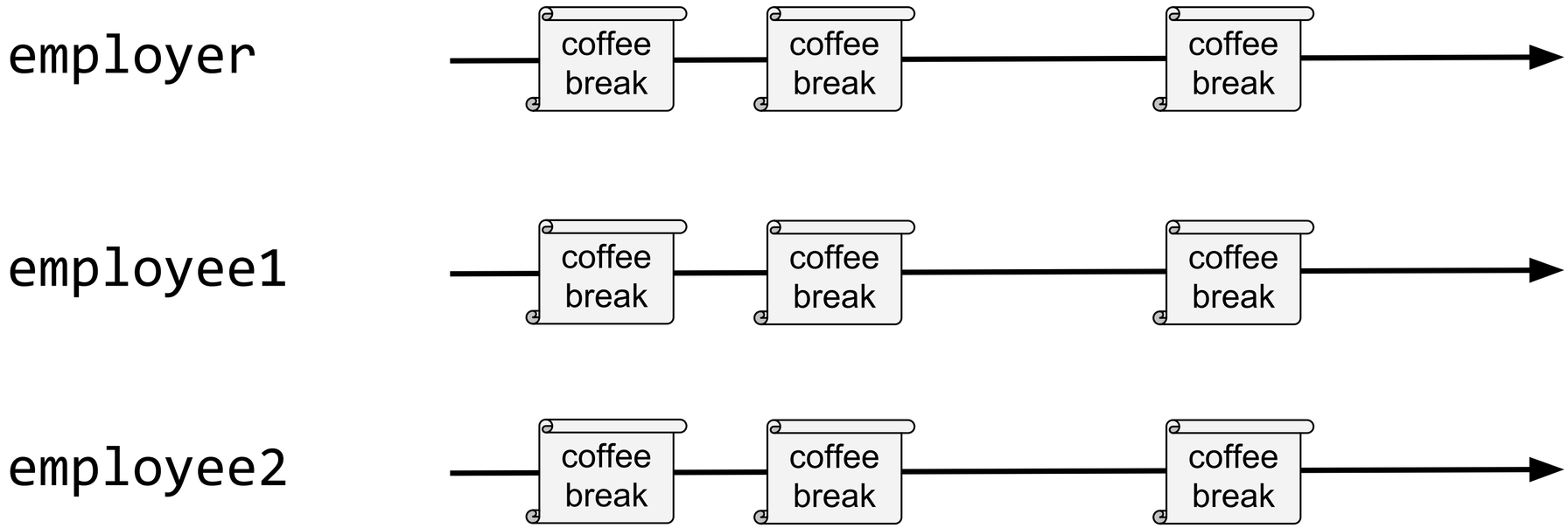
Example of static fan-out operator: `Balance`





Example of static fan-out operator: `Broadcast`





Dynamic fan-in

- The number of input ports of the `Merge` and `ZipWith` fan-in operators is fixed at the time the graph is *described*,
- However, there are some situations where you want to be able to add new input ports after the graph has started *running*.

Example: a coffee factory allowing workers to join or leave the production line at any time without interrupting the production of coffee.

Dynamic fan-in operator: MergeHub

```
class CoffeeFactory()(implicit mat: Materializer) {
  // Assumes that the factory can warehouse the produced coffee
  private def warehouse(coffee: Coffee): Unit = ...

  private val workersHub =
    MergeHub.source[Coffee]           // : Source[Coffee, Sink[Coffee, _]]
      .to(Sink.foreach(warehouse))    // : RunnableGraph[Sink[Coffee, _]]
      .run()                          // : Sink[Coffee, _]

  def connectWorker(worker: Source[Coffee, _]): Unit =
    worker.to(workersHub).run()
}
```

Dynamic fan-in operator: `MergeHub` (2)

`MergeHub` creates a `Source` that emits elements merged from a dynamic set of producers:

- The `Source` returned by `MergeHub.source` produces no elements by itself, but running it materializes a `Sink`,
- That `Sink` can be materialized (ie. run) arbitrary many times,
- Each of the `Sink` materialization will feed the elements it receives to the original `Source`.

Dynamic fan-out operator: BroadcastHub

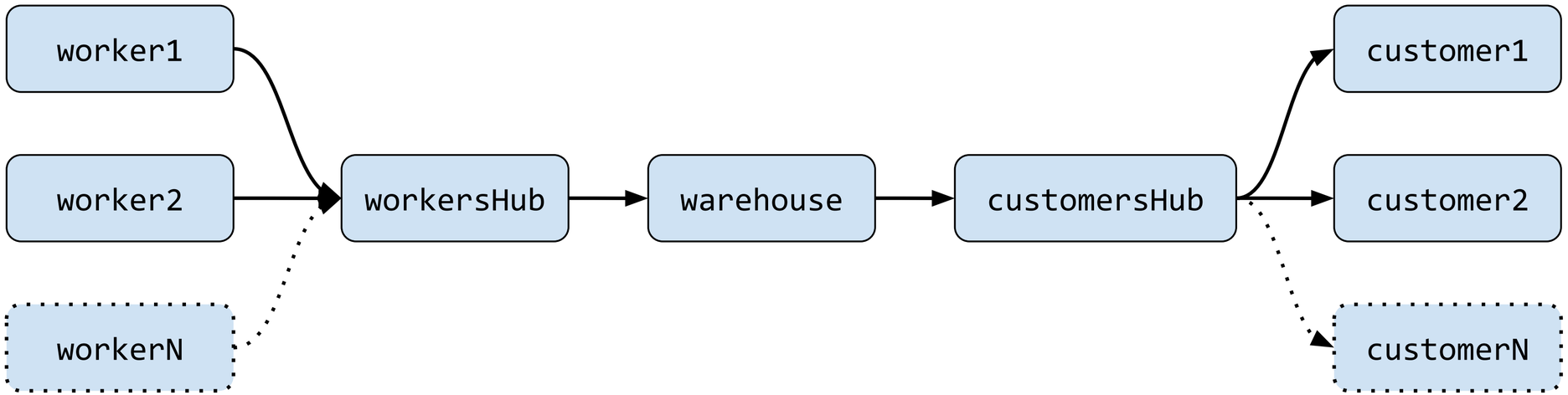
```scala
private def warehouse: Flow[Coffee, StoredCoffee, _] = ...

private val (workersHub, customersHub) =
  MergeHub.source[Coffee]
    .via(warehouse)
    .toMat(BroadcastHub.sink[StoredCoffee])(Keep.both)
    .run()

def connectWorker(worker: Source[Coffee, _]): Unit =
  worker.to(workersHub).run()

def connectCustomer(customer: Sink[StoredCoffee, _]): Unit =
  customersHub.to(customer).run()
```

Dynamic fan-out operator: `BroadcastHub` (2)



Dynamic fan-out operator: `BroadcastHub` (3)

`BroadcastHub` works in similar way than `MergeHub`.

It creates a `Sink` that receives elements from an upstream producer and broadcasts them to a dynamic set of consumers:

- After the `Sink` returned by `BroadcastHub.sink` is materialized, it returns a `Source`,
- That `Source` can be materialized arbitrary many times,
- Each of the `Source` materialization receives elements from the original `Sink`.

Summary

In this video we learnt:

- what *fan-in* and *fan-out* operations are
- how to use the static and dynamic fan-in and fan-out operations in Akka Streams
- how those operations relate to processing rate