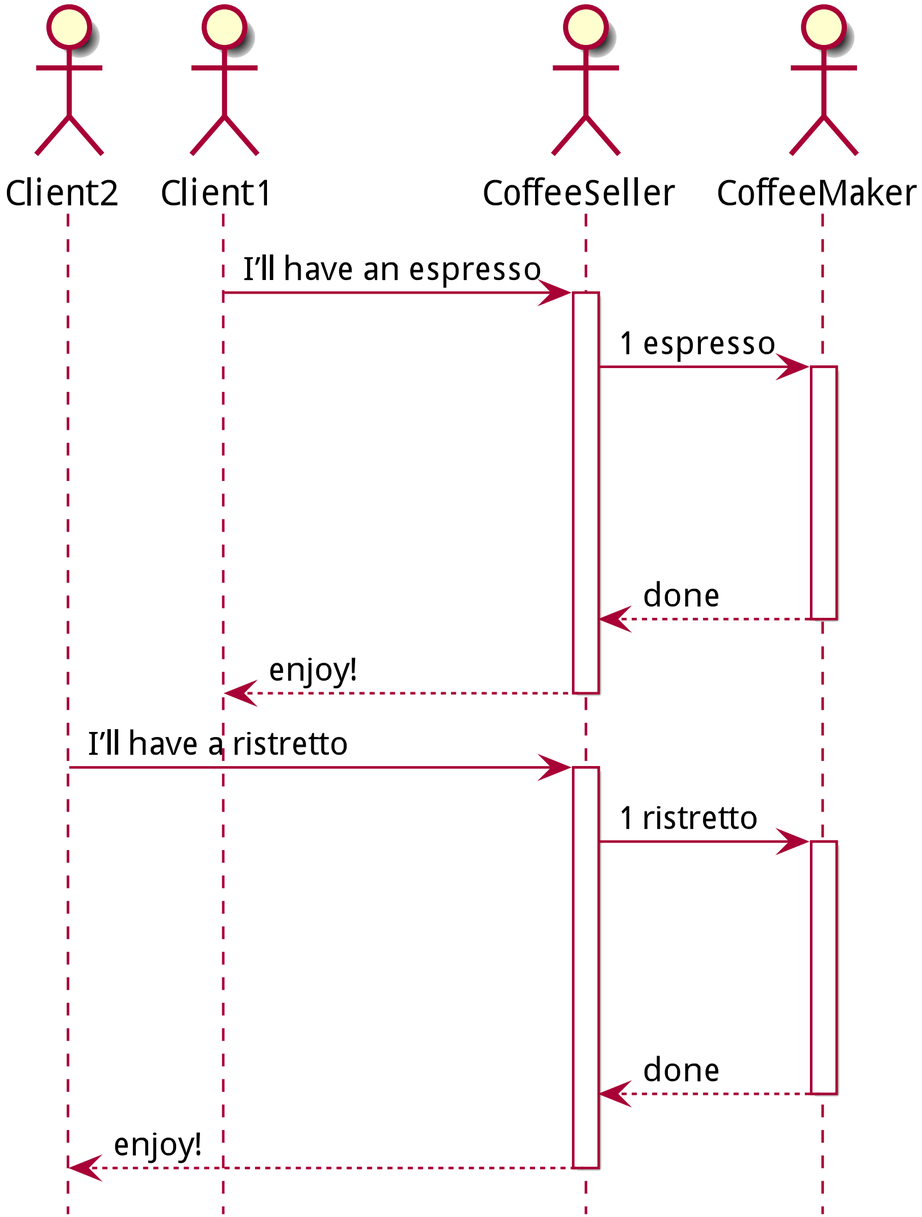
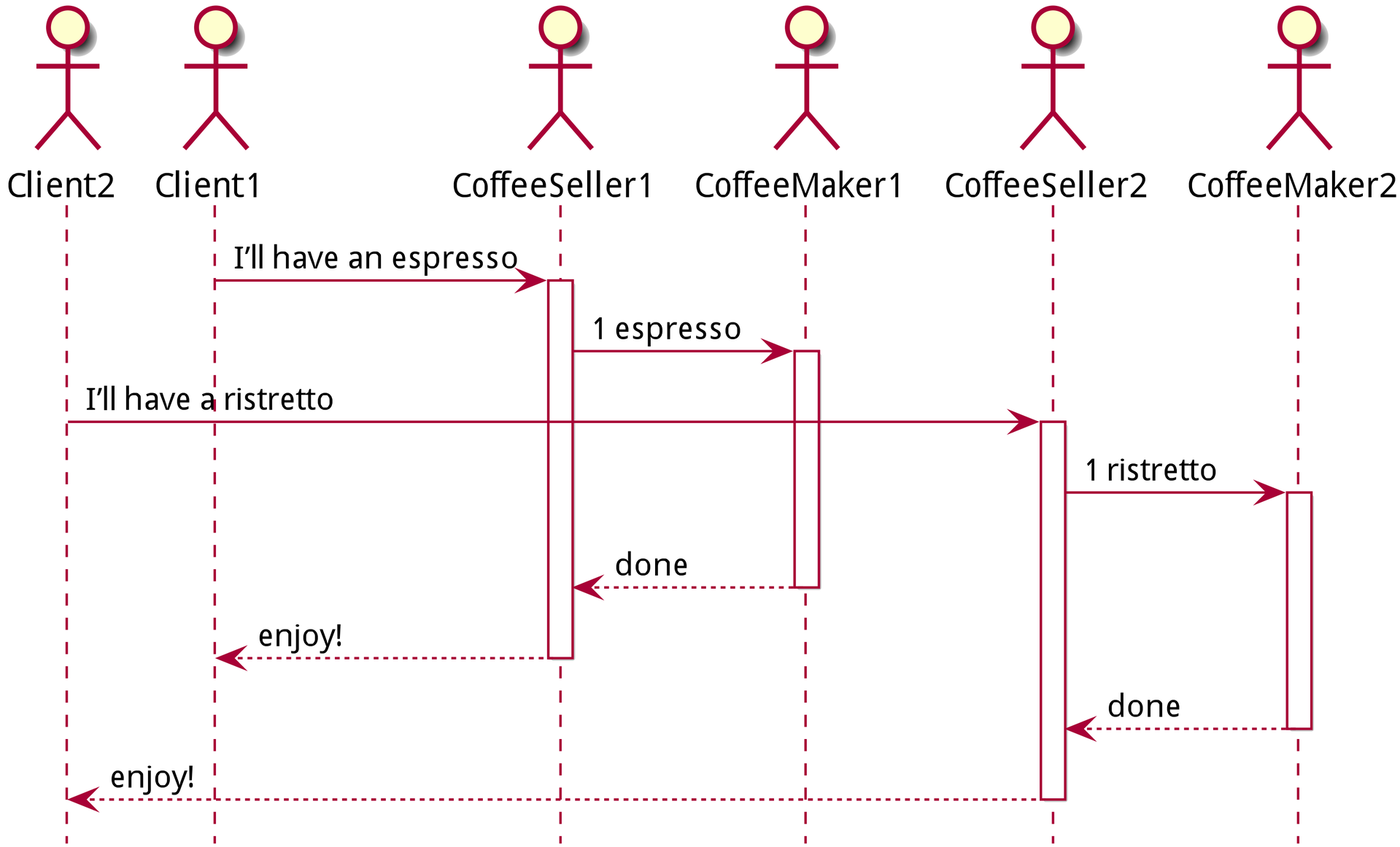
Asynchronous Programming

Programming Reactive Systems

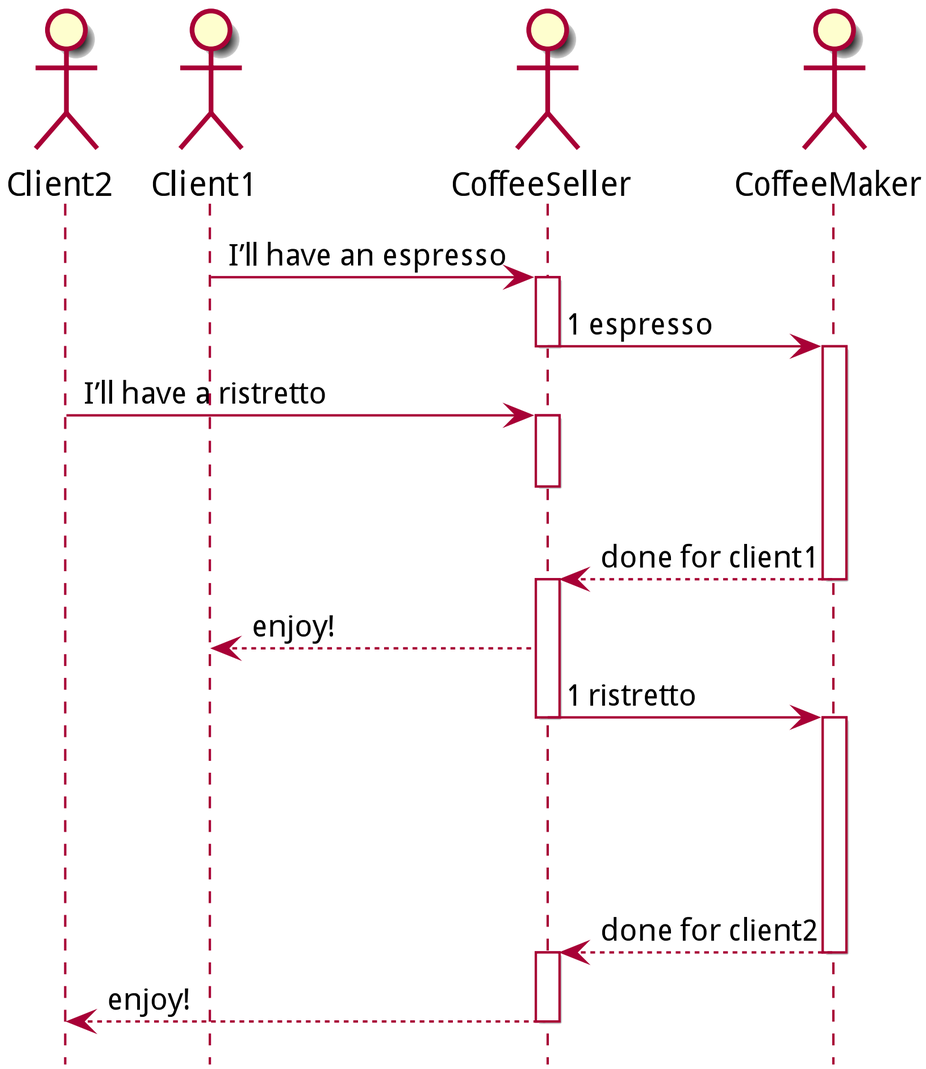
Julien Richard-Foy


Client2
Client1
CoffeeSeller
CoffeeMaker
I'll have an espresso
1 espresso
done
enjoy!
I'll have a ristretto
1 ristretto
done
enjoy!

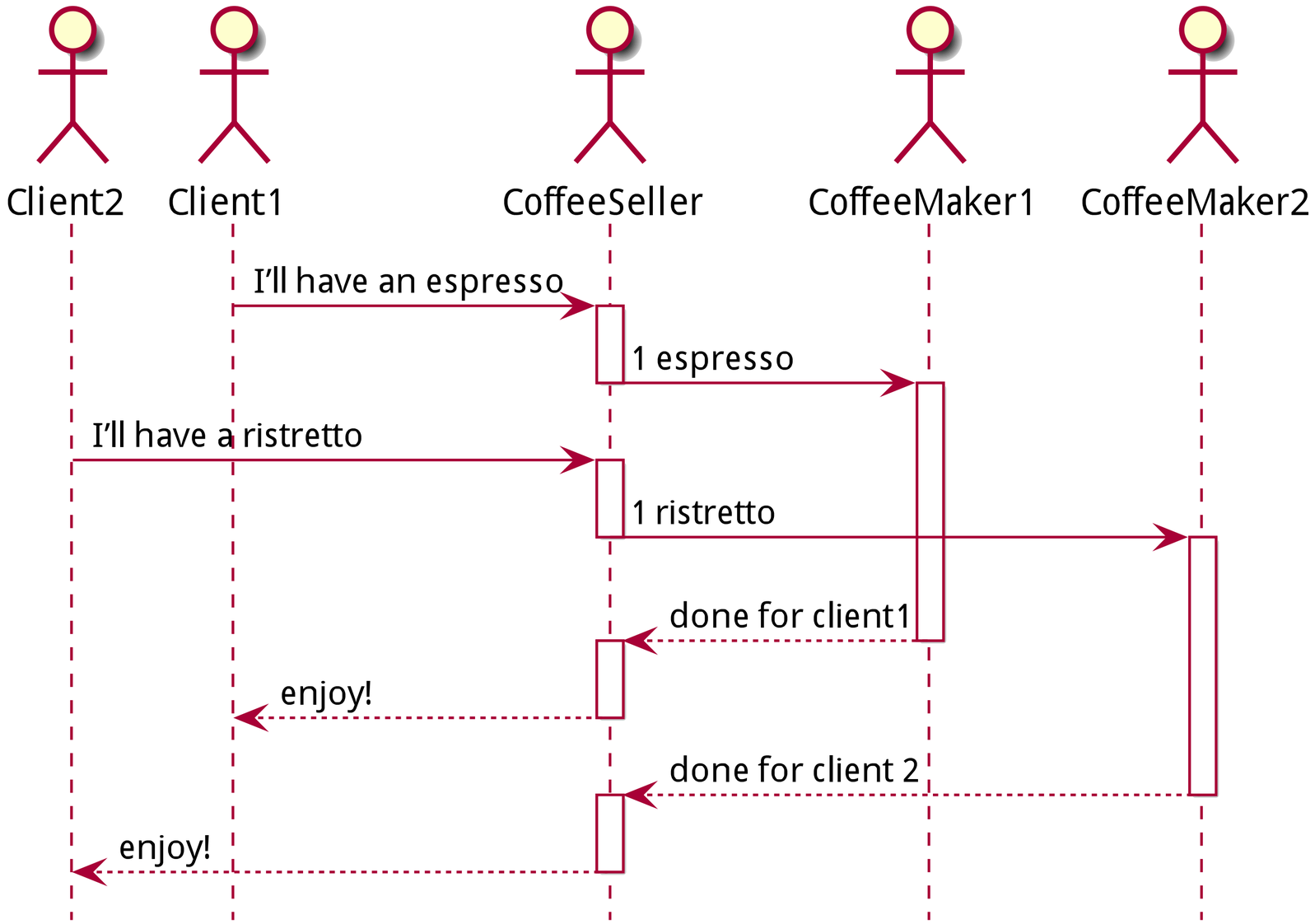
StarBlocks Scaled



ScalaBucks


Client2
Client1
CoffeeSeller
CoffeeMaker
I'll have an espresso
1 espresso
I'll have a ristretto
done for client1
enjoy!
1 ristretto
done for client2
enjoy!

ScalaBucks Scaled



Asynchronous Execution

- Execution of a computation on *another* computing unit, without *waiting* for its termination ;
- Better resource efficiency.

Concurrency Control of Asynchronous Programs

What if a program A *depends on* the result of an asynchronously executed program B?

```
def coffeeBreak(): Unit = {
  val coffee = makeCoffee()
  drink(coffee)
  chatWithColleagues()
}
```

Callback

```
def makeCoffee(coffeeDone: Coffee => Unit): Unit = {
  // work hard ...
  // ... and eventually
  val coffee = ...
  coffeeDone(coffee)
}

def coffeeBreak(): Unit = {
  makeCoffee { coffee =>
   drink(coffee)
  }
  chatWithColleagues()
}
```

From Synchronous to Asynchronous Type Signatures

A synchronous type signature can be turned into an asynchronous type signature by:

- returning `Unit`
- and taking as parameter a **continuation** defining what to do after the return value has been computed

```
def program(a: A): B

def program(a: A, k: B => Unit): Unit
```

Combining Asynchronous Programs (1)

```
def makeCoffee(coffeeDone: Coffee => Unit): Unit = ...

def makeTwoCoffees(coffeesDone: (Coffee, Coffee) => Unit): Unit = ???
```

Combining Asynchronous Programs (2)

```
def makeCoffee(coffeeDone: Coffee => Unit): Unit = ...

def makeTwoCoffees(coffeesDone: (Coffee, Coffee) => Unit): Unit = {
  var firstCoffee: Option[Coffee] = None
  val k = { coffee: Coffee =>
    firstCoffee match {
      case None           => firstCoffee = Some(coffee)
      case Some(coffee2) => coffeesDone(coffee, coffee2)
    }
  }
  makeCoffee(k)
  makeCoffee(k)
}
```

Callbacks All the Way Down (1)

What if another program *depends on* the coffee break to be done?

```
def coffeeBreak(): Unit = ...
```

- We need to make `coffeeBreak` take a callback too!

Callbacks all the Way Down (2)

```
def coffeeBreak(breakDone: Unit => Unit): Unit = ...

def workRoutine(workDone: Work => Unit): Unit = {
  work { work1 =>
    coffeeBreak { _ =>
      work { work2 =>
        workDone(work1 + work2)
      }
    }
  }
}
```

Callbacks all the Way Down (2)

```
def coffeeBreak(breakDone: Unit => Unit): Unit = ...

def workRoutine(workDone: Work => Unit): Unit = {
  work { work1 =>
    coffeeBreak { _ =>
      work { work2 =>
        workDone(work1 + work2)
      }
    }
  }
}
```

- Order of execution follows the indentation level!

Handling Failures

- In synchronous programs, failures are handled with exceptions ;
- What happens if an asynchronous call fails?
  - We need a way to propagate the failure to the call site

Handling Failures

- In synchronous programs, failures are handled with exceptions ;
- What happens if an asynchronous call fails?
  - We need a way to propagate the failure to the call site

```
def makeCoffee(coffeeDone: Try[Coffee] => Unit): Unit = ...
```

Summary

In this video, we have seen:

- How to *sequence* asynchronous computations using **callbacks**
- Callbacks introduce complex type signatures
- The continuation passing style is tedious to use