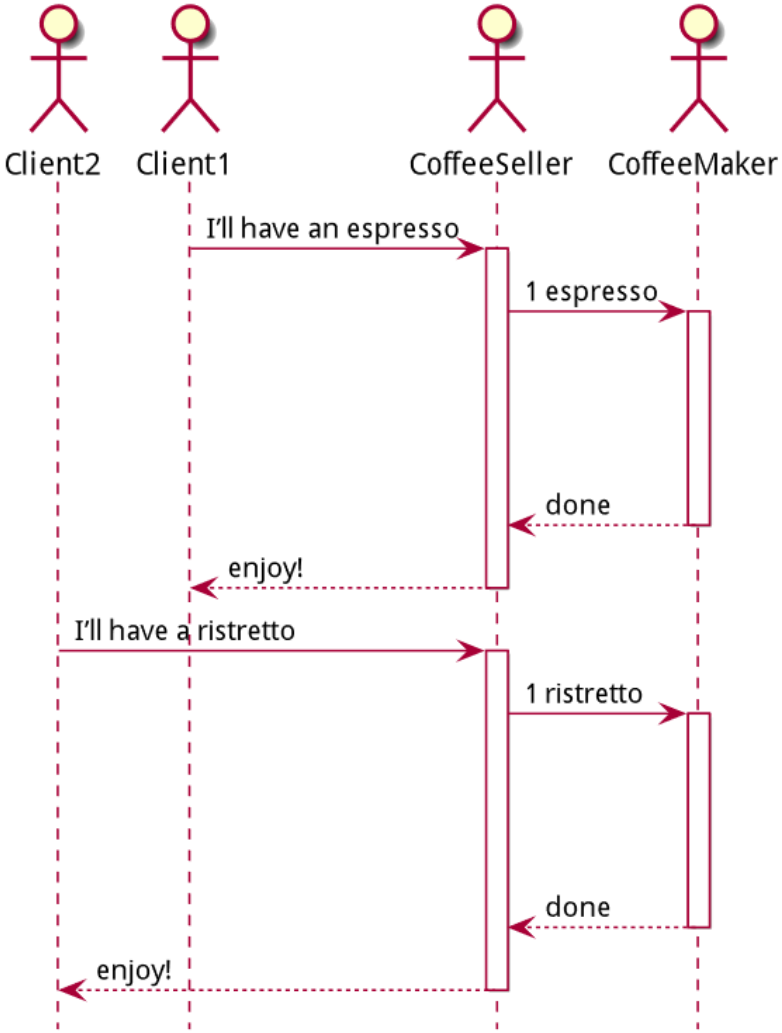
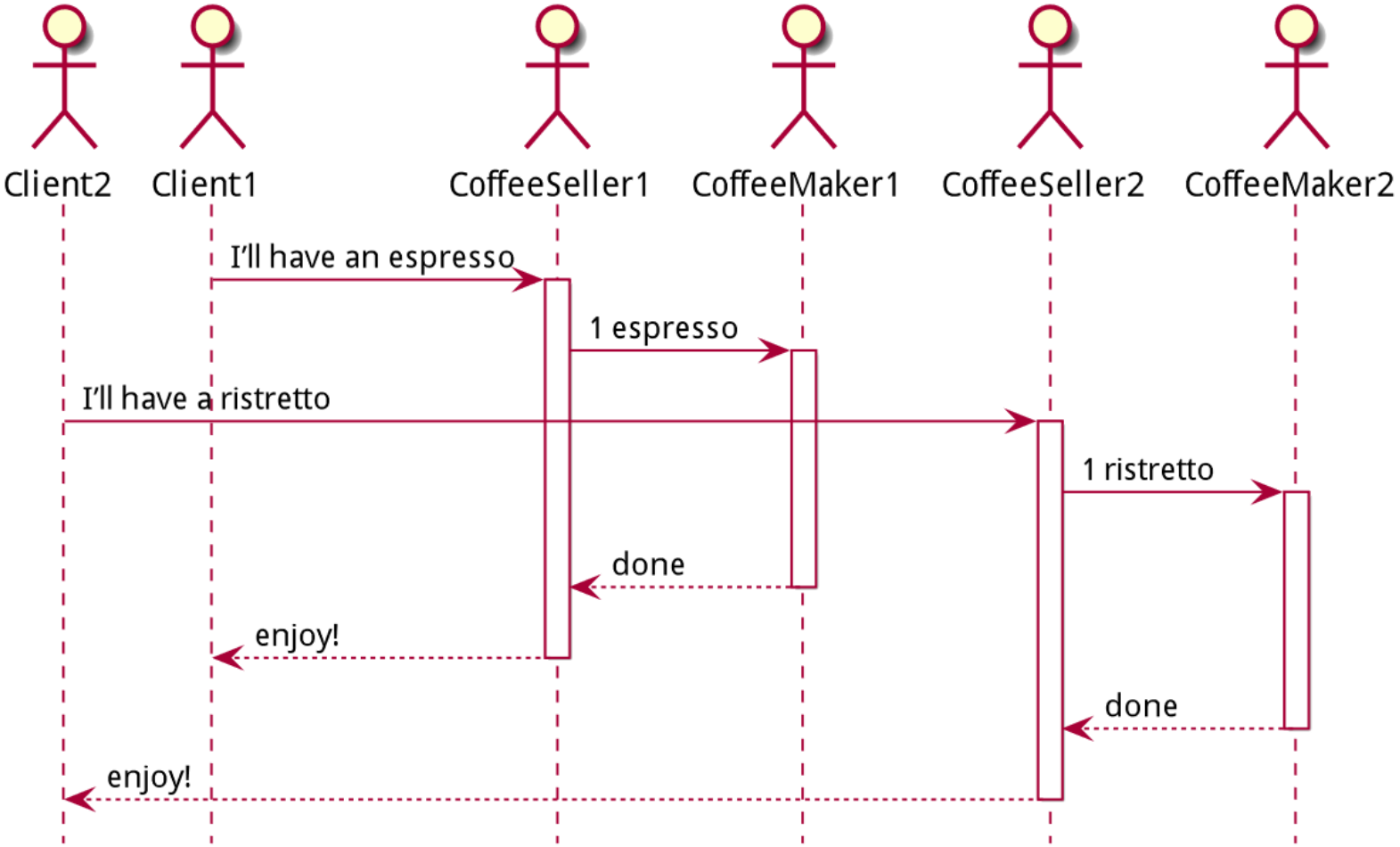
Review Some Concepts of RedBook

Futures example

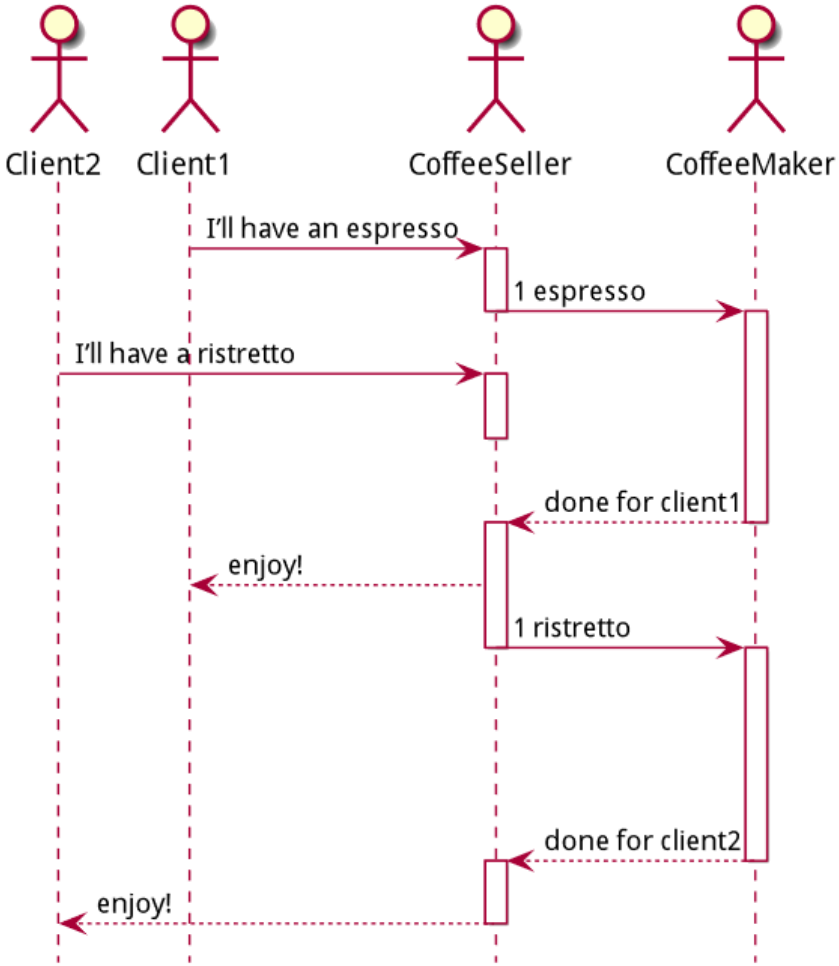
Sync Calls - StarBlocks



Parallel Calls - ParallelCosta



Async Calls – Future Cafe



Async Calls – Advantages

Execution of a computation on another computing unit, without waiting for its termination;

Better Resource Efficiency

Coffee Sync to Async Code

```
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()
}
```

When its ready

Sync to Async With Callback

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

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

Combining Asynchronous

```
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)
}
```

Handling Failures

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

Whats Wrong with callbacks?

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

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

BLACK HOLES & REVELATIONS!

Futures a better Approach

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

```
def program(a: A): Future[B]
```

Effect encapsulated

```
type Future[+T] = (Try[T] => Unit) => Unit
```

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

Future definition

```
type Future[+T] = (Try[T] => Unit) => Unit

// by reifying the alias into a proper trait
trait Future[+T] extends ((Try[T] => Unit) => Unit) {
  def apply(k: Try[T] => Unit): Unit
}

// by renaming ‘apply‘ to ‘onComplete‘
trait Future[+T] {
  def onComplete(k: Try[T] => Unit): Unit

}
```

Make Coffee with Futures

```
def makeCoffee(): Future[Coffee] = ...

def coffeeBreak(): Unit = {
  makeCoffee().onComplete {
    case Success(coffee) => drink(coffee)
    case Failure(reason) => ...
  }
  chatWithColleagues()
}
```

Functor/Monad Future

```
trait Future[+A] {
  def onComplete(k: Try[A] => Unit): Unit
  // transform successful results
  def map[B](f: A => B): Future[B]
  def flatMap[B](f: A => Future[B]): Future[B]
  def zip[B](fb: Future[B]): Future[(A, B)]
  // transform failures
  def recover(f: Exception => A): Future[A]
  def recoverWith(f: Exception => Future[A]): Future[A]
}
```

Sequencing Futures. Runs f/fb after the future runs

Runs in parallel

Zip vs Flatmap

```scala
def makeTwoCoffees(): Future[(Coffee, Coffee)] =
  makeCoffee() zip makeCoffee()
```

```scala
def makeTwoCoffees(): Future[(Coffee, Coffee)] =
  makeCoffee().flatMap { coffee1 =>
    makeCoffee().map(coffee2 => (coffee1, coffee2))
  }
```

For Comprehension

```scala
def work(): Future[Work] = ...
def coffeeBreak(): Future[Unit] =

def workRoutine(): Future[Work] =
  work().flatMap { work1 =>
    coffeeBreak().flatMap { _ =>
      work().map { work2 =>
        work1 + work2
      }
    }
  }
}
```

=

```scala
def work(): Future[Work] = ...
def coffeeBreak(): Future[Unit] = ...

def workRoutine(): Future[Work] =
  for {
    work1 <- work()
    _     <- coffeeBreak()
    work2 <- work()
  } yield work1 + work2
```

ExecutionContext

Dispatcher: FixedThreadPool/Single Thread etc...

```scala
trait Future[+A] {
  def onComplete(k: Try[A] => Unit)(implicit ec: ExecutionContext): Unit
}

import scala.concurrent.ExecutionContext.Implicits.global
```

Lift a CallBack to Future

```
def makeCoffee(
  coffeeDone: Coffee => Unit,
  onFailure: Exception => Unit
): Unit

def makeCoffee2(): Future[Coffee] = {
  val p = Promise[Coffee]()
  makeCoffee(
    coffee => p.trySuccess(coffee),
    reason => p.tryFailure(reason)
  )
  p.future
}
```

Questions