Distributed in Space and Time

Dr. Roland Kuhn
Akka Tech Lead
@rolandkuhn

The Future is Bright

source: http://nature.desktopnexus.com/wallpaper/58502/

The Future is Bright

- Computing Resources: Effectively Unbounded
  - 16-core CPU packages, 1024 around the corner
  - on-demand provisioning in the cloud
- Demand for Web-Scale Apps: Ravenous
  - everyone has the tools
  - successful ideas are bought for Giga-$

How do I make a Large-Scale App?

- figure out a service to provide to everyone
- this will **require resources** accordingly:
  - storage
  - computation
  - communication
- scalability implies **no shared contention points**
- need to be elastic ➟ **use the cloud**

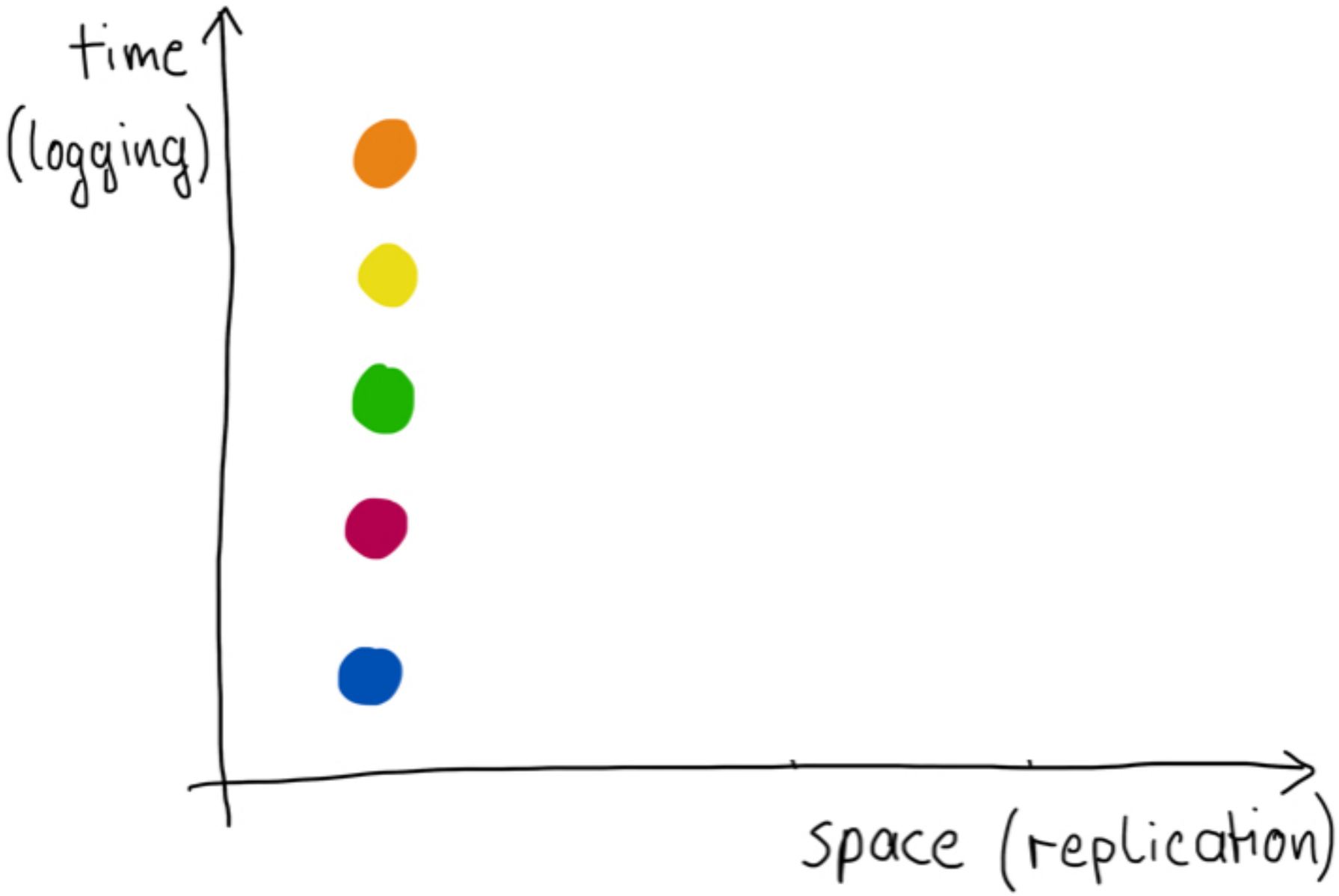
How do I make an Always-Available App?

- must not have single point of failure
- all functions and resources are **replicated**
  - protect against data loss
  - protect against service downtime
  - regional content delivery
- need to distribute ➟ **use the cloud**

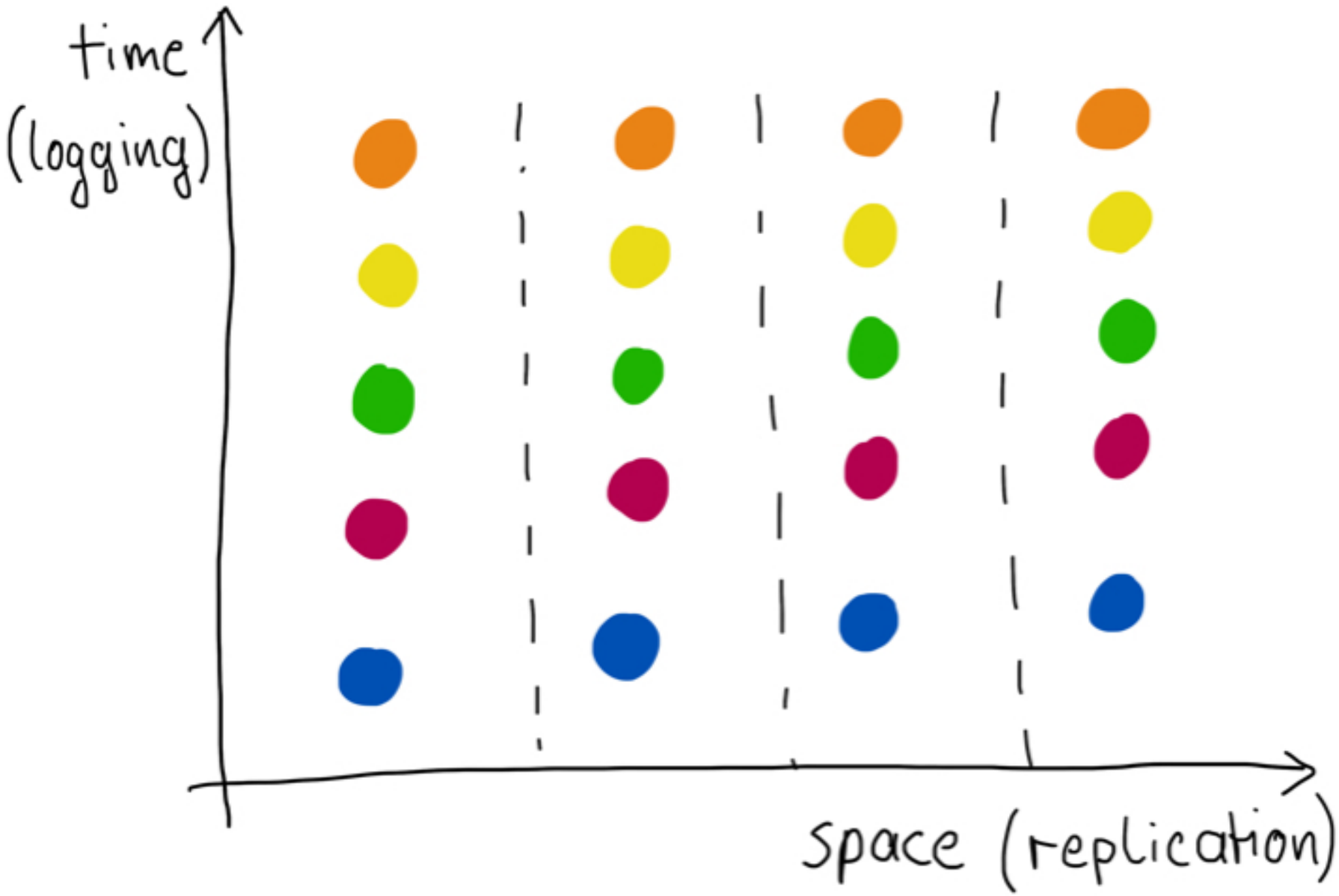
How do I make a Persistent App?

- runtime replication usually not sufficient
  - system of record needed
- one central database is not tenable
- persist “locally” and **replicate changes**

How do I make a Persistent App?



How do I make a Persistent App?



How do I make a Persistent App?

- runtime replication usually not sufficient
  - system of record needed
- one central database is not tenable
- persist "locally" and **replicate changes**
  - temporally distributed change records
  - spatially distributed change log
  - granularity can be one Aggregate Root (Event Sourcing)
  - extract business intelligence from history
- scale & distribute persistence ➟ **in the cloud**

As Noted Earlier: The Future is Bright

source: http://nature.desktopnexus.com/wallpaper/58502/

… or is it?

Photograph: Patrick Cromerford, 2011

Distribution: a Double-Edged Sword

- did that message make it over the network?
- is that other machine running?
- are our data centers in sync?
- did we lose messages in that hardware crash?

The Essence of the Problem

A distributed system is one whose parts can fail independently of each other.

Fighting Back!

source: www.hear2heal.com (Kokopelli Rain Dance)

PAXOS and Friends

- problem: distributed consensus
- hard solutions favor **Consistency**
  - 2PC, PAXOS, RAFT
  - if a node is unreachable then consensus cannot be reached
- soft solutions favor **Availability**
  - quorum, allow nodes to drop out
  - partitions can lead to more than one active set
- you **need to choose** which one is appropriate

Fighting Message Loss

- resending is the only cure
  - sender stores the message and retries
  - recipient must send acknowledgement eventually
- fighting unreliable medium is sender's obligation
  - sender's buffer must be persistent
  - resending must begin anew after a crash

At-Least-Once Delivery with Akka

```
class ALOD(other: ActorPath) extends PersistentActor with AtLeastOnceDelivery {
  val IDs = Iterator from 1

  def receiveCommand = {
    case Command(x) =>
      persist(ImportantResult(x)) { ir =>
        sender() ! Done(x)
        deliver(other, FollowUpCommand(x, IDs.next(), _))
      }
    case rc: ReceiptConfirmed =>
      persist(rc) { rc => confirmDelivery(rc.deliveryId) }
  }

  def receiveRecover = {
    case ImportantResult(x) => deliver(other, FollowUpCommand(x, IDs.next(), _))
    case ReceiptConfirmed(_, id) => confirmDelivery(id)
  }
}
```

At-Least-Once Delivery with Akka

```
class ALOD(other: ActorPath) extends PersistentActor with AtLeastOnceDelivery {
  val IDs = Iterator from 1

  def receiveCommand = {
    case Command(x) =>
      persist(ImportantResult(x)) { ir =>
        sender() ! Done(x)
        deliver(other, FollowUpCommand(x, IDs.next(), _))
      }
    case rc: ReceiptConfirmed =>
      persist(rc) { rc => confirmDelivery(rc.deliveryId) }
  }

  def receiveRecover = {
    case ImportantResult(x) => deliver(other, FollowUpCommand(x, IDs.next(), _))
    case ReceiptConfirmed(_, id) => confirmDelivery(id)
  }
}
```

At-Least-Once Delivery with Akka

```
class ALOD(other: ActorPath) extends PersistentActor with AtLeastOnceDelivery {
  val IDs = Iterator from 1

  def receiveCommand = {
    case Command(x) =>
      persist(ImportantResult(x)) { ir =>
        sender() ! Done(x)
        deliver(other, FollowUpCommand(x, IDs.next(), _))
      }
    case rc: ReceiptConfirmed =>
      persist(rc) { rc => confirmDelivery(rc.deliveryId) }
  }

  def receiveRecover = {
    case ImportantResult(x) => deliver(other, FollowUpCommand(x, IDs.next(), _))
    case ReceiptConfirmed(_, id) => confirmDelivery(id)
  }
}
```

At-Least-Once Delivery with Akka

```scala
class ALOD(other: ActorPath) extends PersistentActor with AtLeastOnceDelivery {
  val IDs = Iterator from 1

  def receiveCommand = {
    case Command(x) =>
      persist(ImportantResult(x)) { ir =>
        sender() ! Done(x)
        deliver(other, FollowUpCommand(x, IDs.next(), _))
      }
    case rc: ReceiptConfirmed =>
      persist(rc) { rc => confirmDelivery(rc.deliveryId) }
  }

  def receiveRecover = {
    case ImportantResult(x) => deliver(other, FollowUpCommand(x, IDs.next(), _))
    case ReceiptConfirmed(_, id) => confirmDelivery(id)
  }
}
```

At-Least-Once Delivery with Akka

```
class ALOD(other: ActorPath) extends PersistentActor with AtLeastOnceDelivery {
  val IDs = Iterator from 1

  def receiveCommand = {
    case Command(x) =>
      persist(ImportantResult(x)) { ir =>
        sender() ! Done(x)
        deliver(other, FollowUpCommand(x, IDs.next(), _))
      }
    case rc: ReceiptConfirmed =>
      persist(rc) { rc => confirmDelivery(rc.deliveryId) }
  }

  def receiveRecover = {
    case ImportantResult(x) => deliver(other, FollowUpCommand(x, IDs.next(), _))
    case ReceiptConfirmed(_, id) => confirmDelivery(id)
  }
}
```

Mysterious Double-Bookings

- acknowledgements can be lost
- sender will faithfully duplicate the message
- therefore named **at-least-once**
- recipient is responsible for dealing with this

Exactly-Once Delivery

- processing occurs only once for each message
- all middleware does it*
  - EIP: Guaranteed Delivery just means persistence
  - JMS: configure session for AUTO_ACKNOWLEDGE
  - Google's MillWheel

Exactly-Once Delivery with Akka

```
class PersistThenAct extends PersistentActor {
  var nextId = 1
  def receiveCommand = {
    case FollowUpCommand(x, id, deliveryId) =>
      val rc = ReceiptConfirmed(id, deliveryId)
      if (id == nextId) {
        persist(rc) { rc =>
          sender() ! rc
          doTheAction()
          nextId += 1
        }
      } else if (id < nextId) sender() ! rc
      // otherwise an older command was lost, so wait for the resend
  }
  def receiveRecover = {
    case ReceiptConfirmed(id, _) => nextId = id + 1
  }
  private def doTheAction(): Unit = () // actually do the action
}
```

Exactly-Once Delivery with Akka

```
class PersistThenAct extends PersistentActor {
  var nextId = 1
  def receiveCommand = {
    case FollowUpCommand(x, id, deliveryId) =>
      val rc = ReceiptConfirmed(id, deliveryId)
      if (id == nextId) {
        persist(rc) { rc =>
          sender() ! rc
          doTheAction()
          nextId += 1
        }
      } else if (id < nextId) sender() ! rc
      // otherwise an older command was lost, so wait for the resend
  }
  def receiveRecover = {
    case ReceiptConfirmed(id, _) => nextId = id + 1
  }
  private def doTheAction(): Unit = () // actually do the action
}
```

Exactly-Once Delivery with Akka

```
class PersistThenAct extends PersistentActor {
  var nextId = 1
  def receiveCommand = {
    case FollowUpCommand(x, id, deliveryId) =>
      val rc = ReceiptConfirmed(id, deliveryId)
      if (id == nextId) {
        persist(rc) { rc =>
          sender() ! rc
          doTheAction()
          nextId += 1
        }
      } else if (id < nextId) sender() ! rc
      // otherwise an older command was lost, so wait for the resend
  }
  def receiveRecover = {
    case ReceiptConfirmed(id, _) => nextId = id + 1
  }
  private def doTheAction(): Unit = () // actually do the action
}
```

Exactly-Once Delivery with Akka

```
class PersistThenAct extends PersistentActor {
  var nextId = 1
  def receiveCommand = {
    case FollowUpCommand(x, id, deliveryId) =>
      val rc = ReceiptConfirmed(id, deliveryId)
      if (id == nextId) {
        persist(rc) { rc =>
          sender() ! rc
          doTheAction()
          nextId += 1
        }
      } else if (id < nextId) sender() ! rc
      // otherwise an older command was lost, so wait for the resend
  }
  def receiveRecover = {
    case ReceiptConfirmed(id, _) => nextId = id + 1
  }
  private def doTheAction(): Unit = () // actually do the action
}
```

Exactly-Once Delivery with Akka

```
class PersistThenAct extends PersistentActor {
  var nextId = 1
  def receiveCommand = {
    case FollowUpCommand(x, id, deliveryId) =>
      val rc = ReceiptConfirmed(id, deliveryId)
      if (id == nextId) {
        persist(rc) { rc =>
          sender() ! rc
          doTheAction()
          nextId += 1
        }
      } else if (id < nextId) sender() ! rc
      // otherwise an older command was lost, so wait for the resend
  }
  def receiveRecover = {
    case ReceiptConfirmed(id, _) => nextId = id + 1
  }
  private def doTheAction(): Unit = () // actually do the action
}
```

Exactly-Once Delivery with Akka

```
class PersistThenAct extends PersistentActor {
  var nextId = 1
  def receiveCommand = {
    case FollowUpCommand(x, id, deliveryId) =>
      val rc = ReceiptConfirmed(id, deliveryId)
      if (id == nextId) {
        doTheAction()
        persist(rc) { rc =>
          sender() ! rc
          nextId += 1
        }
      } else if (id < nextId) sender() ! rc
      // otherwise an older command was lost, so wait for the resend
  }
  def receiveRecover = {
    case ReceiptConfirmed(id, _) => nextId = id + 1
  }
  private def doTheAction(): Unit = () // actually do the action
}
```

The Hard Truth

CPU can stop between any two instructions, therefore exactly-once semantics cannot be Guaranteed.*

**with a capital G, i.e. with 100% absolute coverage*

Mitigating that Hard Truth

- probability of loss or duplication can be small
- many systems are fine with almost-exactly-once
- still need to decide on which side to err
  - persist-then-act ➟ almost-exactly-but-at-most-once
  - act-then-persist ➟ almost-exactly-but-at-least-once

We Can Do Better!

- an operation is idempotent if multiple application produces the same result as single application
- (side-)effects must also be idempotent
  - unique transaction identifiers etc.
- this achieves **effectively-exactly-once**
- can save persistence round-trip latency by optimistic execution

Effectively-Exactly-Once with Akka

```
class Idempotent(other: ActorRef) extends PersistentActor {
  var nextId = 1
  def receiveCommand = {
    case FollowUpCommand(x, id, deliveryId) =>
      val rc = ReceiptConfirmed(id, deliveryId)
      if (id == nextId) {
        // optimistically execute
        if (isValid(x)) {
          updateState(x)
          other ! Command(x) // assuming idempotent receiver
        } else sender() ! Invalid(x)
        nextId += 1
        // then take care of reliable delivery handling
        persistAsync(rc) { rc =>
          sender() ! rc
        }
      } else if (id < nextId) sender() ! rc
  }
  ...
}
```

The Future is Bright Again!

source: http://freewallpapersbackgrounds.com/

Consistency à la Carte

- not all parts of a system have the same needs
- expend the effort only where necessary
  - idempotency does not come for free
  - confirmation handling cannot always be abstracted away
- incur the runtime overhead only where needed
  - persistence round-trips take time
  - consensus protocols take more time
- include required semantics in system design

Remember that the trade-off is always between consistency and availability.

And latency.