CONCURRENCY IN PRACTICE

A CASE STUDY

ERIK ROZENDAAL

zilverline

INTRODUCTION

WHO AM I?

Erik Rozendaal, software developer, etc.

email: erozendaal@zilverline.com

twitter: @erozendaal

WHO AM I?

Erik Rozendaal, software developer, etc.

(... and I did not write that CQRS framework)

email: erozendaal@zilverline.com

twitter: @erozendaal

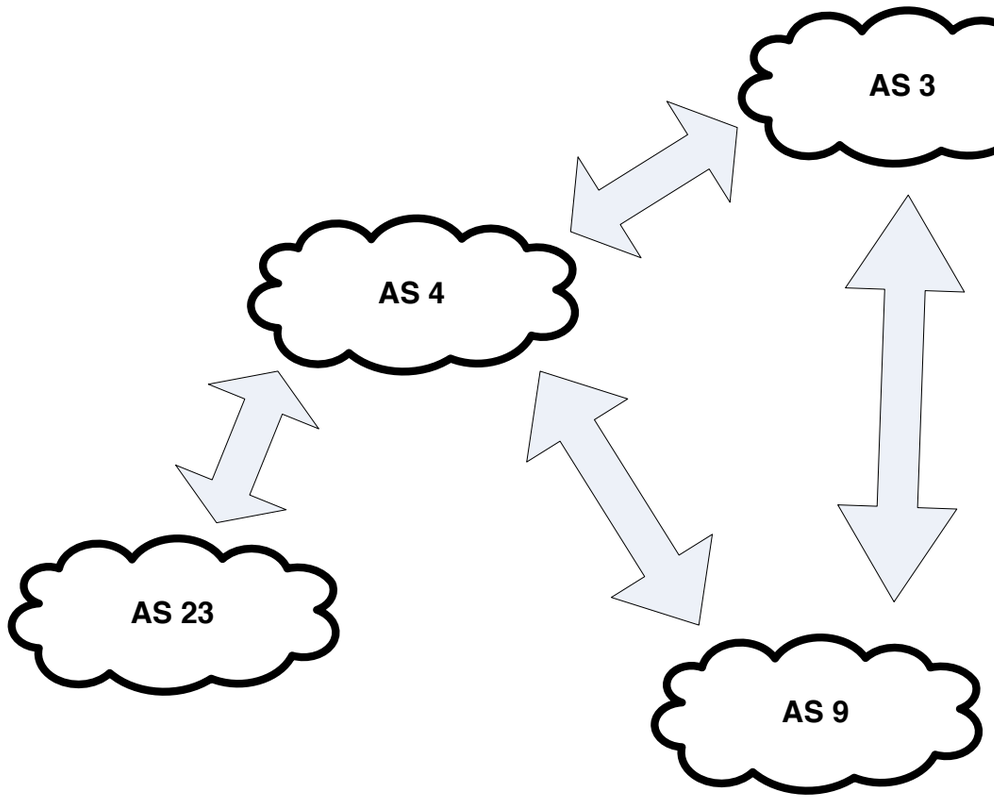
RPKI VALIDATOR

- Open Source (BSD license)
- Developed at the RIPE NCC (www.ripe.net)
- Aimed at Internet router administrators
- http://www.ripe.net/lir-services/resource-management/certification/tools-and-resources

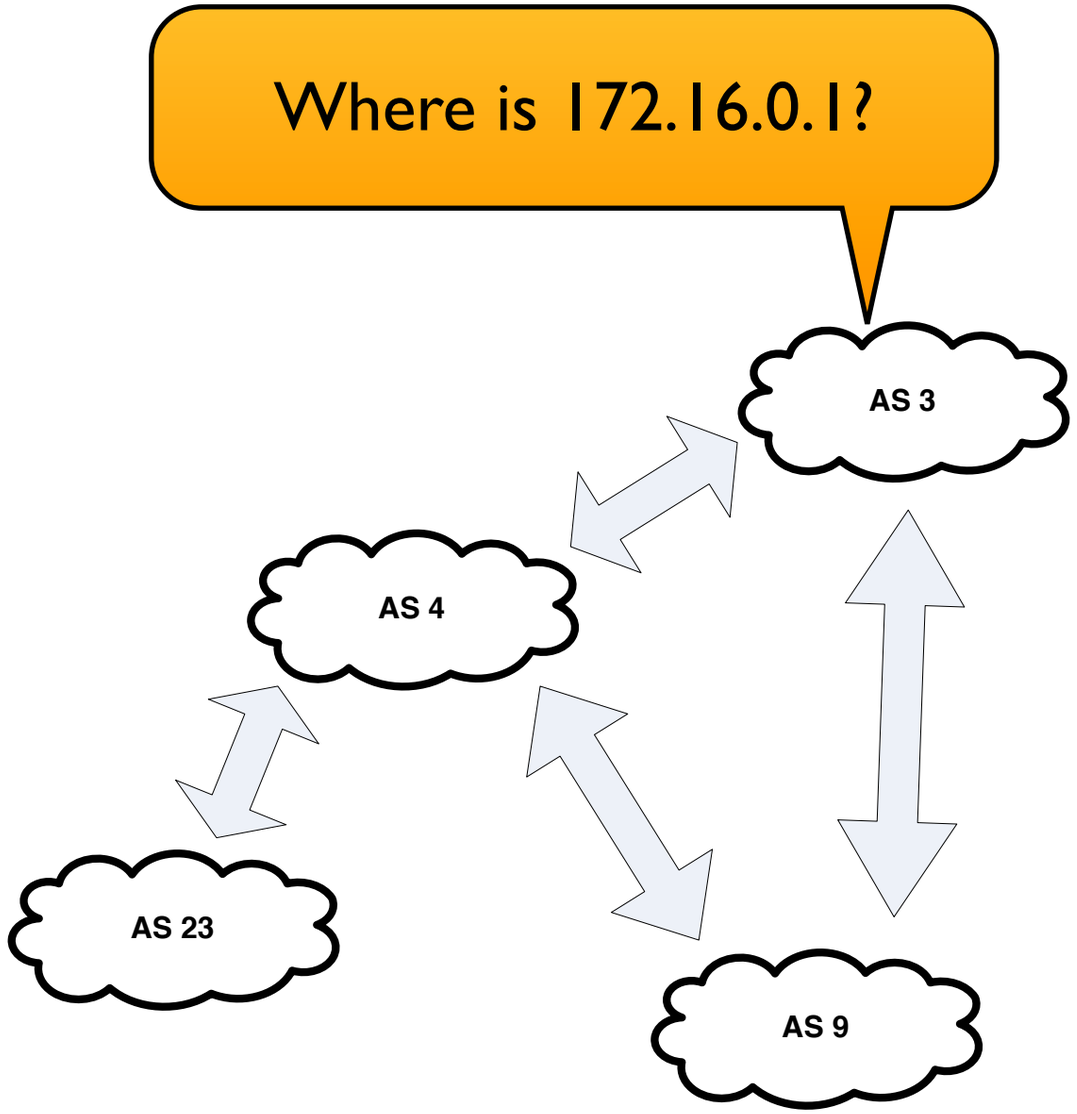
Scalaz

- The RIPE NCC is one of five Regional Internet Registries (RIRs) providing Internet resource allocations, registration services and coordination activities that support the operation of the Internet globally.
- Basically, helps ensure that every Internet Address is uniquely distributed and the Internet keeps working

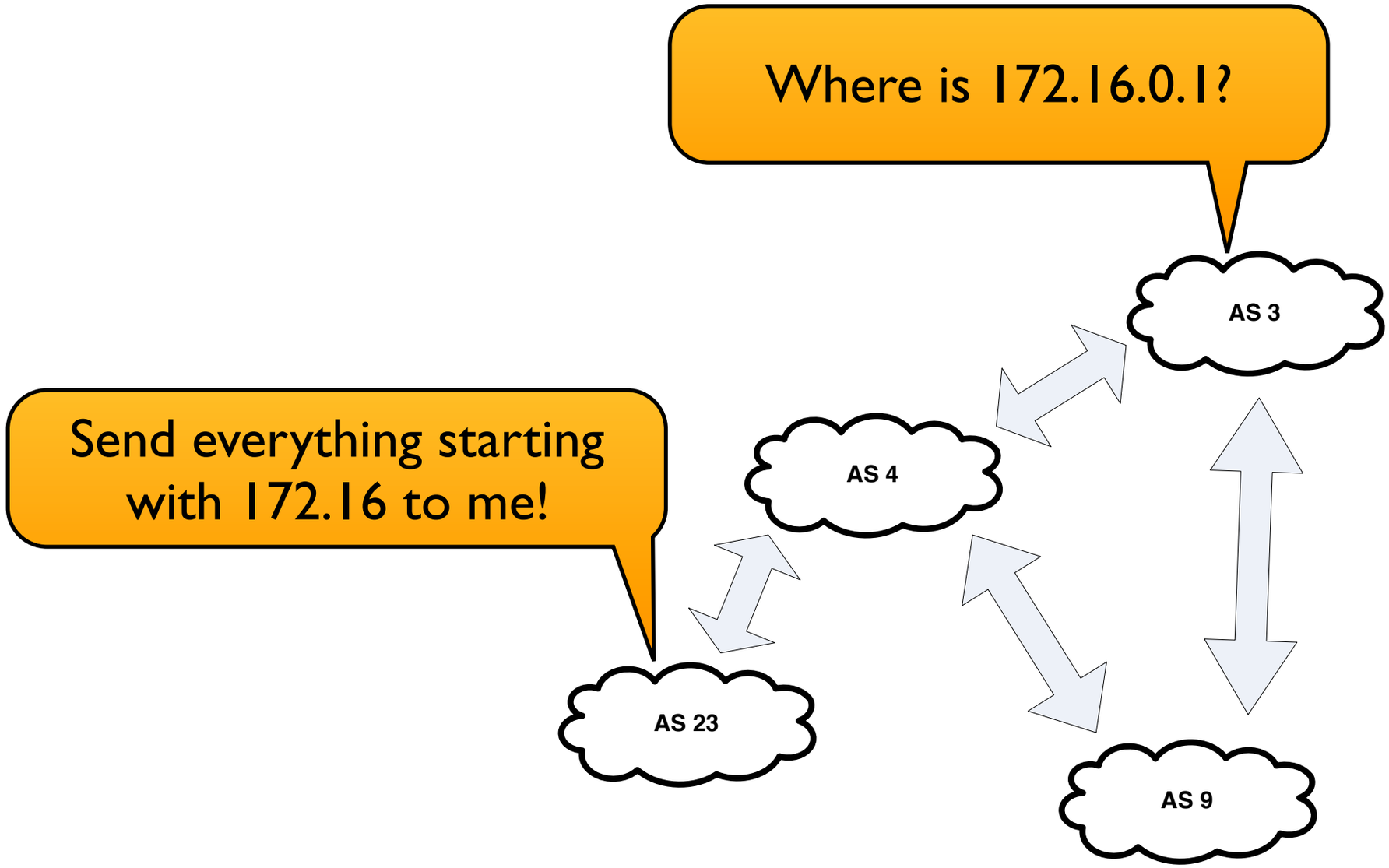
INTERNET ROUTING 101



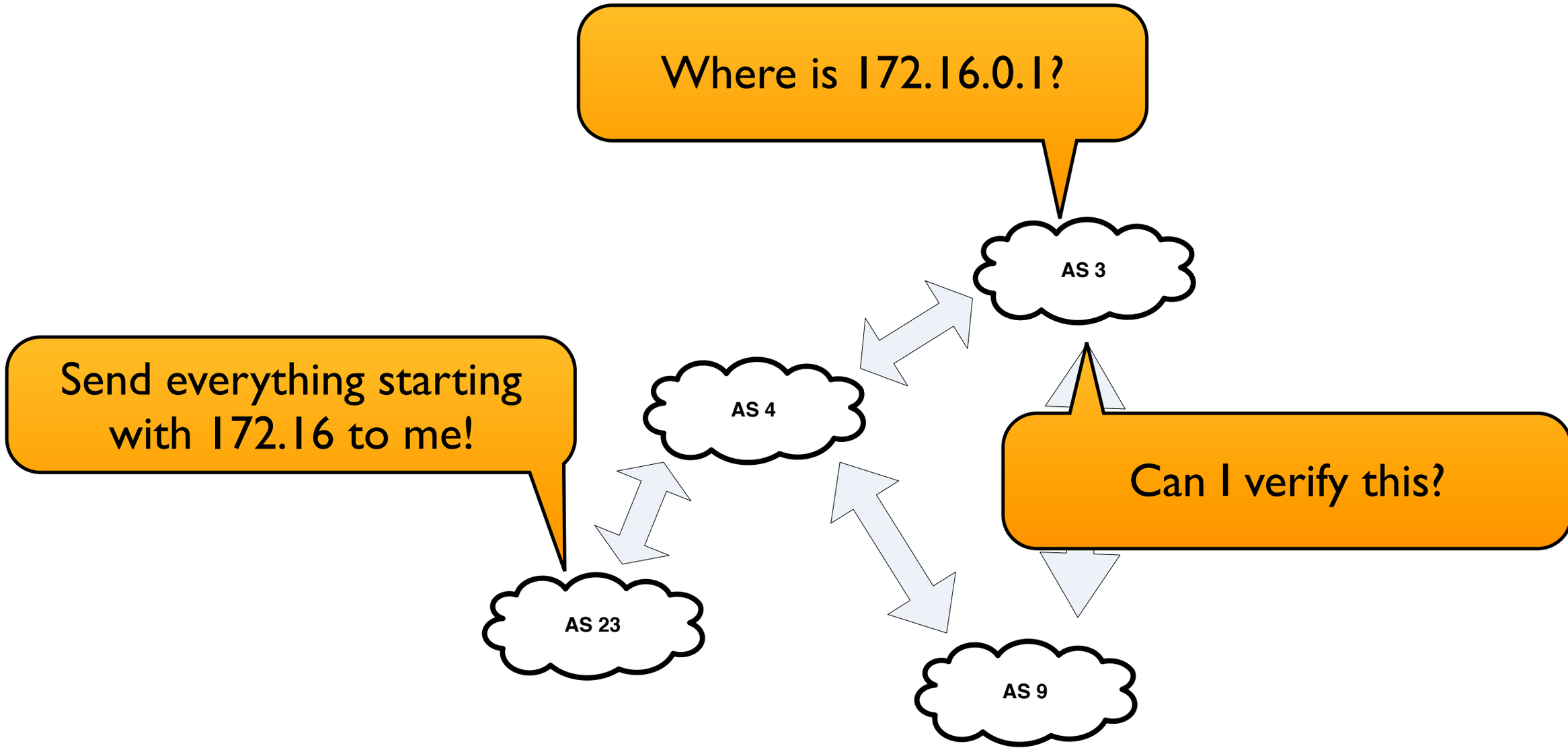
INTERNET ROUTING 101



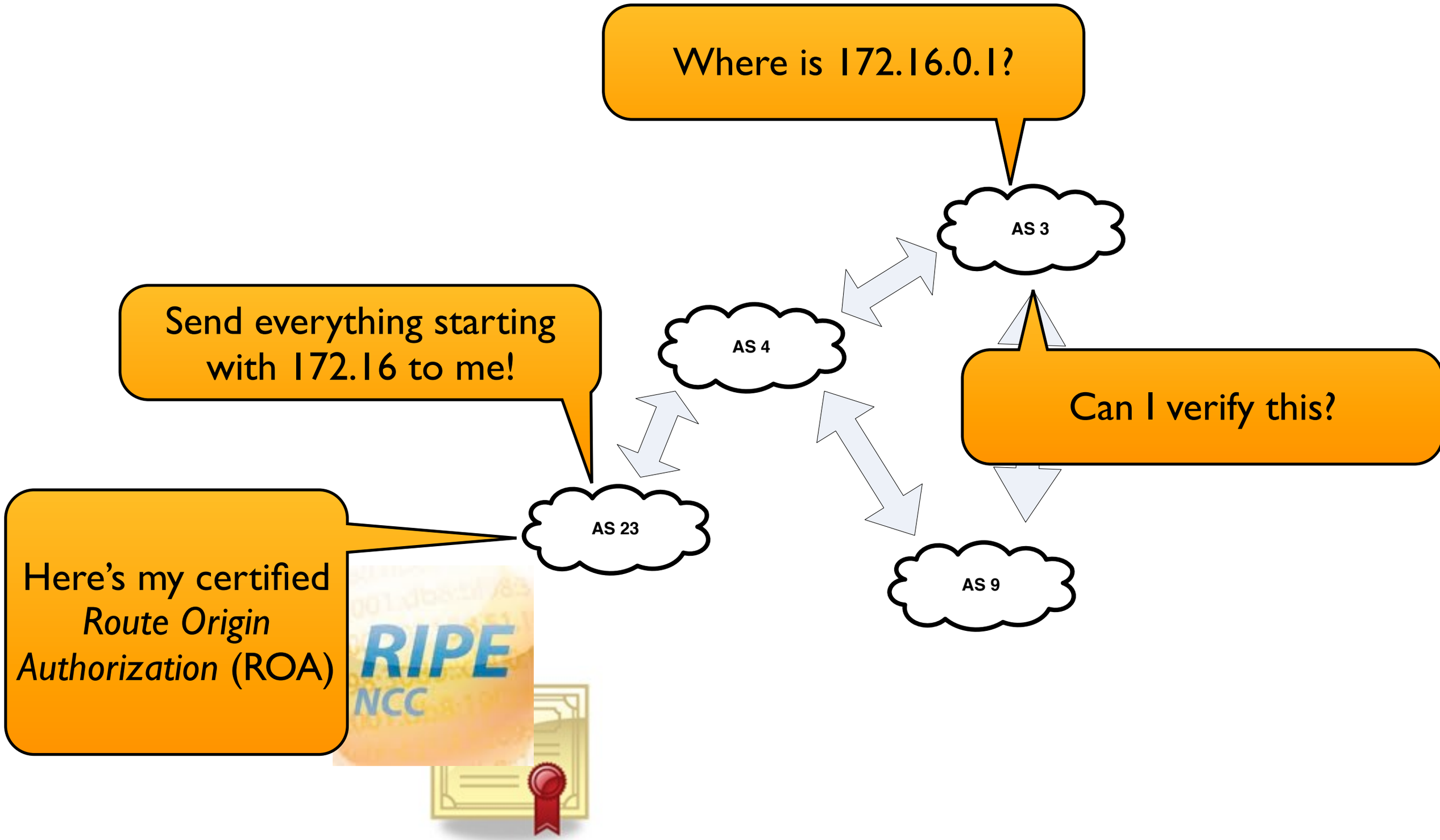
INTERNET ROUTING 101



INTERNET ROUTING 101



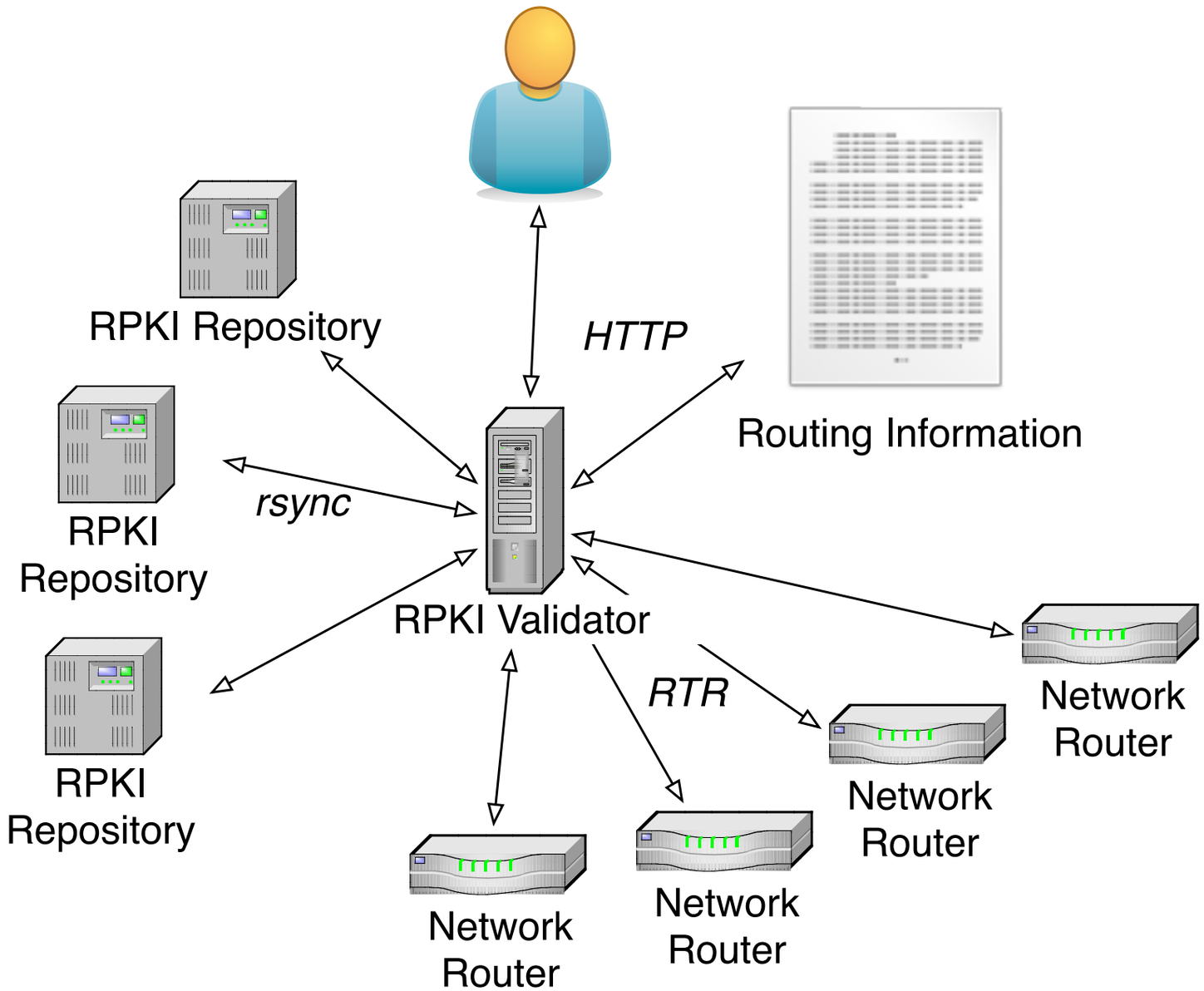
INTERNET ROUTING 101



INTERNET RESOURCE PKI (RPKI)

- Distributed database of cryptographically signed statements about resources
- IETF standard
- Rooted at the five Regional Internet Registries (RIRs)
  - AfriNIC - Africa
  - ARIN - United States, Canada, ...
  - APNIC - Asia, Australia, New Zealand, ...
  - LACNIC - Latin America, ...
  - RIPE NCC - Europe, Russia, Middle East, ...

BIG PICTURE


RPKI Repository
HTTP
Routing Information
RPKI
Repository
rsync
RPKI Validator
RPKI
Repository
RTR
Network
Router
Network
Router
Network
Router
Network
Router


DEMO

IMMUTABILITY

IMMUTABILITY

- An **immutable object** is an object whose state cannot be modified after it is created
- Immutable objects can be safely shared between multiple threads
- Scala makes it easy to define immutable objects and defaults to full set of immutable collection types

IMMUTABLE COLLECTIONS?

IMMUTABLE COLLECTIONS?

```
def add(x: Int, y: Int) = {
  var result = 0
  result += x
  result += y
  result
}
```

IMMUTABLE COLLECTIONS?

```
def add(x: Int, y: Int) = {
  var result = 0
  result += x
  result += y
  result
}
```

```
def concat(x: List, y: List) = {
  val result = new ArrayList()
  result.addAll(x)
  result.addAll(y)
  result
}
```

IMMUTABLE COLLECTIONS?

```
def add(x: Int, y: Int) = {
  var result = 0
  result += x
  result += y
  result
}
```

```
def add(x: Int, y: Int) =
  x + y
```

```
def concat(x: List, y: List) = {
  val result = new ArrayList()
  result.addAll(x)
  result.addAll(y)
  result
}
```

IMMUTABLE COLLECTIONS?

```
def add(x: Int, y: Int) = {
  var result = 0
  result += x
  result += y
  result
}
```

```
def add(x: Int, y: Int) =
  x + y
```

```
def concat(x: List, y: List) = {
  val result = new ArrayList()
  result.addAll(x)
  result.addAll(y)
  result
}
```

```
def concat(x: List, y: List) =
  x ++ y
```

IMMUTABLE COLLECTIONS?

```
def add(x: Int, y: Int) = {
  var result = 0
  result += x
  result += y
  result
}
```

```
def add(x: Int, y: Int) =
  x + y
```

```
def concat(x: List, y: List) = {
  val result = new ArrayList()
  result.addAll(x)
  result.addAll(y)
  result
}
```

```
def concat(x: List, y: List) =
  x ++ y
```

Immutability is the difference between
`java.util.Calendar` and `org.joda.time.DateTime`

MEMORY IMAGE

```
case class MemoryImage(
  trustAnchors    : Vector[TrustAnchor],
  validatedObjects: Vector[ValidatedObject],
  filters         : Vector[Filter],
  whitelist       : Vector[WhitelistEntry],
  version         : Int = 0)
```

```
case class TrustAnchor(
  locator           : TrustAnchorLocator,
  status            : ProcessingStatus,
  enabled           : Boolean = true)

// Etc.
```

MEMORY IMAGE

- Initially access was controlled using a single `AtomicReference` containing the most recent instance
- http://martinfowler.com/bliki/MemoryImage.html

MEMORY IMAGE IMPLEMENTATION

```
object MemoryImage {
  private[this] val memoryImage =
    new AtomicReference(MemoryImage(...))
```

MEMORY IMAGE IMPLEMENTATION

```
object MemoryImage {
  private[this] val memoryImage =
    new AtomicReference(MemoryImage(...))

  // Reading
  def get: MemoryImage = memoryImage.get
```

MEMORY IMAGE IMPLEMENTATION

```
object MemoryImage {
  private[this] val memoryImage =
    new AtomicReference(MemoryImage(...))

  // Reading
  def get: MemoryImage = memoryImage.get

  // Updating
  @tailrec
  def modify(f: MemoryImage => MemoryImage): MemoryImage = {
    val current = memoryImage.get
    val updated = f(current)
    if (memoryImage.compareAndSet(current, updated)) updated
    else modify(f) // Retry
  }
}
```

MEMORY IMAGE IMPLEMENTATION

```
object MemoryImage {
  private[this] val memoryImage =
    new AtomicReference(MemoryImage(...))

  // Reading
  def get: MemoryImage = memoryImage.get

  // Updating
  @tailrec
  def modify(f: MemoryImage => MemoryImage): MemoryImage = {
    val current = memoryImage.get
    val updated = f(current)
    if (memoryImage.compareAndSet(current, updated)) updated
    else modify(f) // Retry
  }
}

// Example update
MemoryImage.modify { memoryImage =>
  memoryImage.copy(filters = /* updated filters */)
}
```

ATOMIC REFERENCE

- Fast and lock-free!
- Callback to modify may be run multiple times, so avoid side-effects

ATOMIC REFERENCE

- Fast and lock-free!
- Callback to modify may be run multiple times, so avoid side-effects
- ... but `AtomicReferences` do not compose, hurting modularity

ATOMIC REFERENCE

- Fast and lock-free!
- Callback to modify may be run multiple times, so avoid side-effects
- ... but `AtomicReferences` do not compose, hurting modularity
  - Try updating two `AtomicReferences` atomically...

ATOMIC REFERENCE

- Fast and lock-free!
- Callback to modify may be run multiple times, so avoid side-effects
- ... but `AtomicReferences` do not compose, hurting modularity
  - Try updating two `AtomicReferences` atomically...
  - (the same is true for locks)

SOFTWARE TRANSACTIONAL MEMORY

- Take the idea of a database transaction (ACID) and apply it to your in-memory data structures (ACI)
- Composable: bigger transactions can be created from existing, smaller transactions
- Not just for concurrency: mutations are automatically cleaned up on transaction rollback

http://nbronson.github.com/scala-stm/

SOFTWARE TRANSACTIONAL MEMORY

SOFTWARE TRANSACTIONAL MEMORY

```
import scala.concurrent.stm._
```

SOFTWARE TRANSACTIONAL MEMORY

```
import scala.concurrent.stm._

// Global reference to current memory image
val memoryImage = Ref(MemoryImage(initial state))
```

SOFTWARE TRANSACTIONAL MEMORY

```
import scala.concurrent.stm._

// Global reference to current memory image
val memoryImage = Ref(MemoryImage(initial state))

// Example read & update
atomic { implicit txn =>
  memoryImage() = memoryImage().copy(
    filters = updated filters)
}
```

SOFTWARE TRANSACTIONAL MEMORY

```
import scala.concurrent.stm._

// Global reference to current memory image
val memoryImage = Ref(MemoryImage(initial state))

// Example read & update
atomic { implicit txn =>
  memoryImage() = memoryImage().copy(
    filters = updated filters)
}
```

Parentheses to read the current value

SOFTWARE TRANSACTIONAL MEMORY

```
import scala.concurrent.stm._

// Global reference to current memory image
val memoryImage = Ref(MemoryImage(initial state))

// Example read & update
atomic { implicit txn =>
  memoryImage() = memoryImage().copy(
    filters = updated filters)
}
```

Parentheses and assignment to update

Parentheses to read the current value

SOFTWARE TRANSACTIONAL MEMORY

```
import scala.concurrent.stm._

// Global reference to current memory image
val memoryImage = Ref(MemoryImage(initial state))

// Example read & update
atomic { implicit txn =>
  memoryImage() = memoryImage().copy(
    filters = updated filters)
}
```

Represents current transaction

Parentheses and assignment to update

Parentheses to read the current value

STM PITFALLS

- Atomic block may be retried, so only mutate data managed by STM. Bad:

```
var start = false
atomic { implicit txn =>
  memoryImage().trustAnchors.
    find    { ta => ta.locator == trustAnchorLocator }.
    filter  { ta => ta.enabled && ta.status.isIdle }.
    foreach { ta =>
      memoryImage() = memoryImage().
        startProcessingTrustAnchor(ta.locator)
      start = true
    }
}
if (start) runValidation()
```

STM PITFALLS

- Atomic block may be retried, so only mutate data managed by STM. Good:

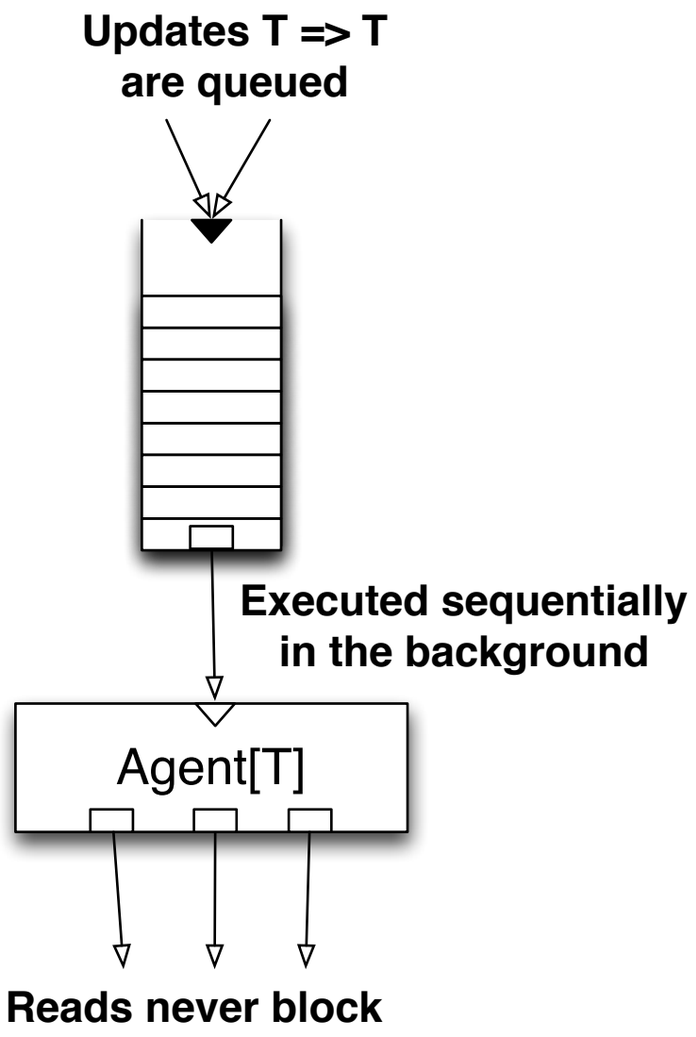
```
val start = atomic { implicit txn =>
  memoryImage().trustAnchors.
    find    { ta => ta.locator == trustAnchorLocator }.
    filter  { ta => ta.enabled && ta.status.isIdle }.
    map     { ta =>
      memoryImage() = memoryImage().
        startProcessingTrustAnchor(ta.locator)
    }.isDefined
}

if (start) runValidation()
```

AGENTS

AGENTS OF T

- Always share current state, reading is “free”
- Queue of pending updates, executed in the background, sequentially



AGENT EXAMPLE

```
private val _validatedAnnouncements =
  Agent(Vector.empty[ValidatedAnnouncement])
```

AGENT EXAMPLE

```
private val _validatedAnnouncements =
  Agent(Vector.empty[ValidatedAnnouncement])

def validatedAnnouncements = _validatedAnnouncements()
```

AGENT EXAMPLE

```
private val _validatedAnnouncements =
  Agent(Vector.empty[ValidatedAnnouncement])

def validatedAnnouncements = _validatedAnnouncements()

def revalidate(announcements: Seq[BgpAnnouncement],
               roas         : Seq[Roa]) {
  _validatedAnnouncements.sendOff {
    _ => validate(announcements, roas)
  }
}
```

AGENTS INTEGRATE WITH STM

- Allows you to update some state and send a computation to an Agent when a STM transaction commits
- Comparable to using a transactional database and message queue, but in-memory

AGENT EXAMPLE

```
private val memoryImage =
  Ref(MemoryImage(initial state))
private val bgpAnnouncements =
  Ref(Vector.empty[BgpAnnouncement])
private val validatedAnnouncements =
  Agent(Vector.empty[ValidatedAnnouncement])
```

```
// Update and start announcement validation
atomic { implicit txn =>
  memoryImage() = memoryImage().copy(filters = Vector.empty)
  val roas = memoryImage().validatedObjects.roas
  val announcements = bgpAnnouncements()
  _validatedAnnouncements.sendOff { _ =>
    validate(announcements, roas)
  }
}
```

FUTURES

FUTURE OF T

- Represents a value of type T that may not be available yet
  - Expensive computation, network access, asynchronous I/O, etc.

FUTURE OF T

- Represents a value of type T that may not be available yet
  - Expensive computation, network access, asynchronous I/O, etc.
- Can be composed, unlike background threads:

FUTURE OF T

- Represents a value of type T that may not be available yet
  - Expensive computation, network access, asynchronous I/O, etc.
- Can be composed, unlike background threads:

```
def traverse[A, B](items: List[A])
                  (f: A => Future[B])
                  (implicit executor: ExecutionContext):
      Future[List[B]]
```

FUTURE OF T

- Represents a value of type T that may not be available yet
  - Expensive computation, network access, asynchronous I/O, etc.
  - Composed, unlike background threads:

E.g. list of URLs to retrieve

```
def traverse[A, B](items: List[A])
                  (f: A => Future[B])
                  (implicit executor: ExecutionContext):
      Future[List[B]]
```

FUTURE OF T

- Represents a value of type T that may not be available yet
  - Expensive computation, network access,

round

threads:

```
def traverse[A, B](items: List[A])
                  (f: A => Future[B])
                  (implicit executor: ExecutionContext):
        Future[List[B]]
```

FUTURE OF T

- Represents a value of type T that may not be available yet
  - Expensive computation, network access, threads:

E.g. list of URLs to retrieve

E.g. fetch URL

How much concurrency?

```
def traverse[A, B](items: List[A])
                  (f: A => Future[B])
                  (implicit executor: ExecutionContext):
        Future[List[B]]
```

FUTURE OF T

- Represents a value of type T that may not be available yet
  - Expensive computation, network access, threads:

E.g. list of URLs to retrieve

E.g. fetch URL

How much concurrency?

```
def traverse[A, B](items: List[A])
                  (f: A => Future[B])
                  (implicit executor: ExecutionContext):
        Future[List[B]]
```

Completed when all complete

CONCURRENT DOWNLOAD

```
val bgpRisDumpUrls = List(
  "http://www.ris.ripe.net/dumps/riswhoisdump.IPv4.gz",
  "http://www.ris.ripe.net/dumps/riswhoisdump.IPv6.gz")
def downloadBgpRisDump(url: String): Future[BgpRisDump] = ...
```

```
Future.traverse(bgpRisDumpsUrls) { url =>
  downloadBgpRisDump(url)
}.foreach { bgpRisDump =>
  // All files have been downloaded, potentially in parallel.
}
```

PARALLELISM

Concurrency: program with multiple, independent threads of control. Non-deterministic, since the outcome may depend on the particular interleaving at runtime.

Concurrency: program with multiple, independent threads of control. Non-deterministic, since the outcome may depend on the particular interleaving at runtime.

Parallelism: runs on multiple processors, hopefully making it run faster. No other affect on program outcome.

PARALLEL COLLECTIONS

Problem: validate ~435,000 BGP announcements against ~2,000 route origin authorizations

```
val result = announcements.map { announcement =>
  val matching = roas.findMatching(announcement.prefix)
  val (validates, invalidates) =
    matching.partition { roa => roa.isValid(announcement) }

  ValidatedAnnouncement(
    announcement, validates, invalidates)
}
```

PARALLEL COLLECTIONS

Problem: validate ~435,000 BGP announcements against ~2,000 route origin authorizations

```
val result = announcements.par.map { announcement =>
  val matching = roas.findMatching(announcement.prefix)
  val (validates, invalidates) =
    matching.partition { roa => roa.isValid(announcement) }

  ValidatedAnnouncement(
    announcement, validates, invalidates)
}.seq
```

PARALLEL COLLECTIONS

- Sequential: ~1.4 seconds,
  Parallel: ~0.8 seconds (75% faster)
- Can be a quick win for CPU-bound tasks
- Deterministic in absence of side-effects, only the performance changes (same, better, worse)
- Often preferable to implementing a smarter algorithm
- We also use this in UI table filtering

ACTORS

ACTORS

- Aren't Akka and Scala all about actors?
- Planning to try actors to replace Validator-to-Router communication implementation
  - Currently uses Netty `ChannelHandlers`
  - Low-level, hard to test, hard to get right
  - Replace with Akka I/O manager and one actor per router?

CONCLUSION

CONCLUSION

- Immutability is golden and so are side-effect free functions
- Concurrency is (still) hard
- But parallelism much easier, almost free
- No single solution to the concurrency problem, use the right tool for the problem at hand
- Scala (like Clojure and Haskell) provides lots of tools that mostly integrate well

... AND NOT COVERED

- Basic Java concurrency (synchronized, notify, wait), java.util.concurrent
- The LMAX Disruptor and the single writer principle
- Dataflow concurrency
- Reactive programming
- Events, event bus, event loops
- Etc.

QUESTIONS?

ERIK ROZENDAAL

zilverline