Implementing a Distributed Hash Table with Scala and Akka

Tristan Penman

@tristanpenman

This Talk in a nutshell

1. An intro to DHTs, the Chord protocol and its supporting algorithms
2. Demo application
3. Modeling the Chord protocol using actors (while following Akka best practices)
4. Akka patterns (Ask and Pipe)
5. Closing remarks and useful resources

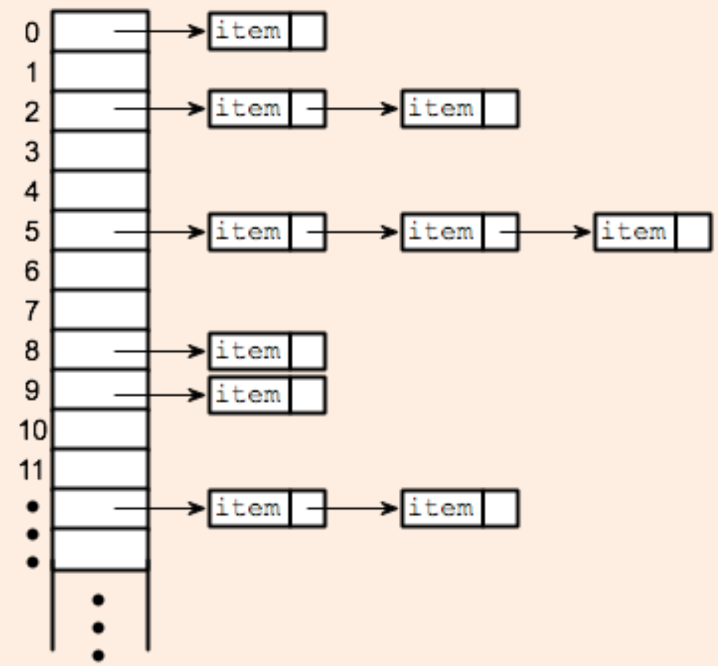
But first… an Akka refresher

- Framework for building concurrent, distributed, and resilient message-driven applications
  - Based on asynchronous message passing
- Emphasises actor-based concurrency
  - Model individual components as ‘Actors’
  - Allows us to build an application from individual components that respond to a set of well-formatted messages

An intro to DHTs, the Chord protocol and its supporting algorithms

Hash Tables in one slide

- A hash table is a data structure used to implement an associative array
- Lookup and insertion operations run in constant-time
- Each element of the array is a bucket that contains one or more keys
  - Think of a bucket as owning a non-contiguous subset of the key-space



Visual representation of a hash table

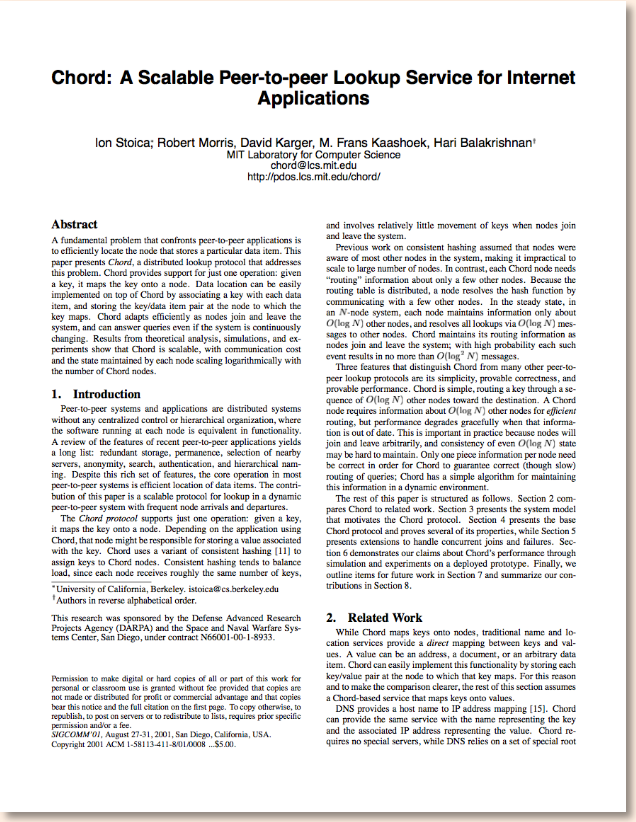
Diagram from: http://math.hws.edu/eck/cs124/javanotes6/c10/s3.html

Distributed Hash Tables

- What do you do when you can't store all of your data on one node? (Or don't want to)
  - Spread the data across multiple nodes, with each node taking responsibility for a portion of the key-space
  - Nodes == buckets
- First problem:
  - How do we figure out which node is responsible for a key?
- Second problem:
  - How do we handle changes to the network topology?
  - Nodes can join or leave the network at any time

The Chord Protocol

- Chord is a protocol and set of algorithms for implementing a Distributed Hash Table
- Key features are:
  - Lookup time is logarithmic in the number of network nodes
  - Asynchronous network stabilization protocol
  - *Consistent Hashing* minimizes disruptions when nodes join or leave the network



Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica*, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan†
MIT Laboratory for Computer Science
chord@lcs.mit.edu
http://pdos.lcs.mit.edu/chord/

Abstract

A fundamental problem that confronts peer-to-peer applications is to efficiently locate the node that stores a particular data item. This paper presents *Chord*, a distributed lookup protocol that addresses this problem. Chord provides support for just one operation: given a key, it maps the key onto a node. Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data item pair at the node to which the key maps. Chord adapts efficiently as nodes join and leave the system, and can answer queries even if the system is continuously changing. Results from theoretical analysis, simulations, and experiments show that Chord is scalable, with communication cost and the state maintained by each node scaling logarithmically with the number of Chord nodes.

1. Introduction

Peer-to-peer systems and applications are distributed systems without any centralized control or hierarchical organization, where the software running at each node is equivalent in functionality. A review of the features of recent peer-to-peer applications yields a long list: redundant storage, permanence, selection of nearby servers, anonymity, search, authentication, and hierarchical naming. Despite this rich set of features, the core operation in most peer-to-peer systems is efficient location of data items. The contribution of this paper is a scalable protocol for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures.

The *Chord protocol* supports just one operation: given a key, it maps the key onto a node. Depending on the application using Chord, that node might be responsible for storing a value associated with the key. Chord uses a variant of consistent hashing [11] to assign keys to Chord nodes. Consistent hashing tends to balance load, since each node receives roughly the same number of keys, and involves relatively little movement of keys when nodes join and leave the system.

Previous work on consistent hashing assumed that nodes were aware of most other nodes in the system, making it impractical to scale to large number of nodes. In contrast, each Chord node needs "routing" information about only a few other nodes. Because the routing table is distributed, a node resolves the hash function by communicating with a few other nodes. In the steady state, in an N-node system, each node maintains information only about $O(\log N)$ other nodes, and resolves all lookups via $O(\log N)$ messages to other nodes. Chord maintains its routing information as nodes join and leave the system; with high probability each such event results in no more than $O(\log^2 N)$ messages.

Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance. Chord is simple, routing a key through a sequence of $O(\log N)$ other nodes toward the destination. A Chord node requires information about $O(\log N)$ other nodes for *efficient* routing, but performance degrades gracefully when that information is out of date. This is important in practice because nodes will join and leave arbitrarily, and consistency of even $O(\log N)$ state may be hard to maintain. Only one piece information per node need be correct in order for Chord to guarantee correct (though slow) routing of queries; Chord has a simple algorithm for maintaining this information in a dynamic environment.

The rest of this paper is structured as follows. Section 2 compares Chord to related work. Section 3 presents the system model that motivates the Chord protocol. Section 4 presents the base Chord protocol and proves several of its properties, while Section 5 presents extensions to handle concurrent joins and failures. Section 6 demonstrates our claims about Chord's performance through simulation and experiments on a deployed prototype. Finally, we outline items for future work in Section 7 and summarize our contributions in Section 8.

2. Related Work

While Chord maps keys onto nodes, traditional name and location services provide a *direct* mapping between keys and values. A value can be an address, a document, or an arbitrary data item. Chord can easily implement this functionality by storing each key/value pair at the node to which that key maps. For this reason and to make the comparison clearer, the rest of this section assumes a Chord-based service that maps keys onto values.

DNS provides a host name to IP address mapping [15]. Chord can provide the same service with the name representing the key and the associated IP address representing the value. Chord requires no special servers, while DNS relies on a set of special root

* University of California, Berkeley. istoica@cs.berkeley.edu
† Authors in reverse alphabetical order.

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8933.

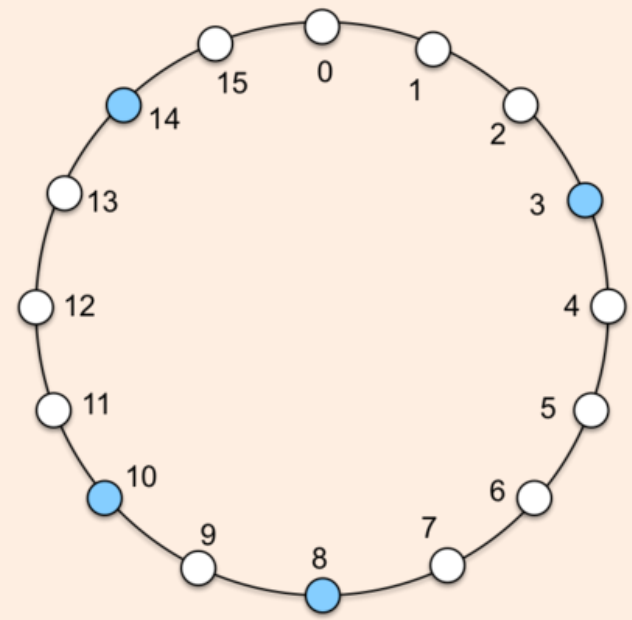
Final paper (published in Transactions on Networking): https://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf

Application Layer

- "Does not specify application layer behavior"
  - Does not prescribe replication techniques
  - Same applies to load balancing of requests coming into the network
  - Redistribution of data associated with keys when nodes leave or join the network is the responsibility of the application

Visualizing Chord

- Instead of an array, we have a key-space which can be visualized as a ring
- A hash function is used to map keys onto locations on the ring
- Nodes are also mapped to locations on the ring
  - Typically determined by applying a hash function to their IP address and port number



Empty circles represent distinct locations on the ring. Blue-filled circles indicate that nodes exist at those locations.

Diagram from: https://www.cs.rutgers.edu/~pxk/417/notes/23-lookup.html

Consistent Hashing

- Chord assigns responsibility for segments of the ring to individual nodes
  - This scheme is called Consistent Hashing
  - Allows nodes to be added or removed from the network while minimizing the number of keys that will need to be reassigned
- We can figure out which node owns a given key by applying the hash function, then choosing the node whose location on the ring is equal to or greater than that of the key
  - This node is the 'successor' of the key.

Adding a node

Node 6 has been added to the network

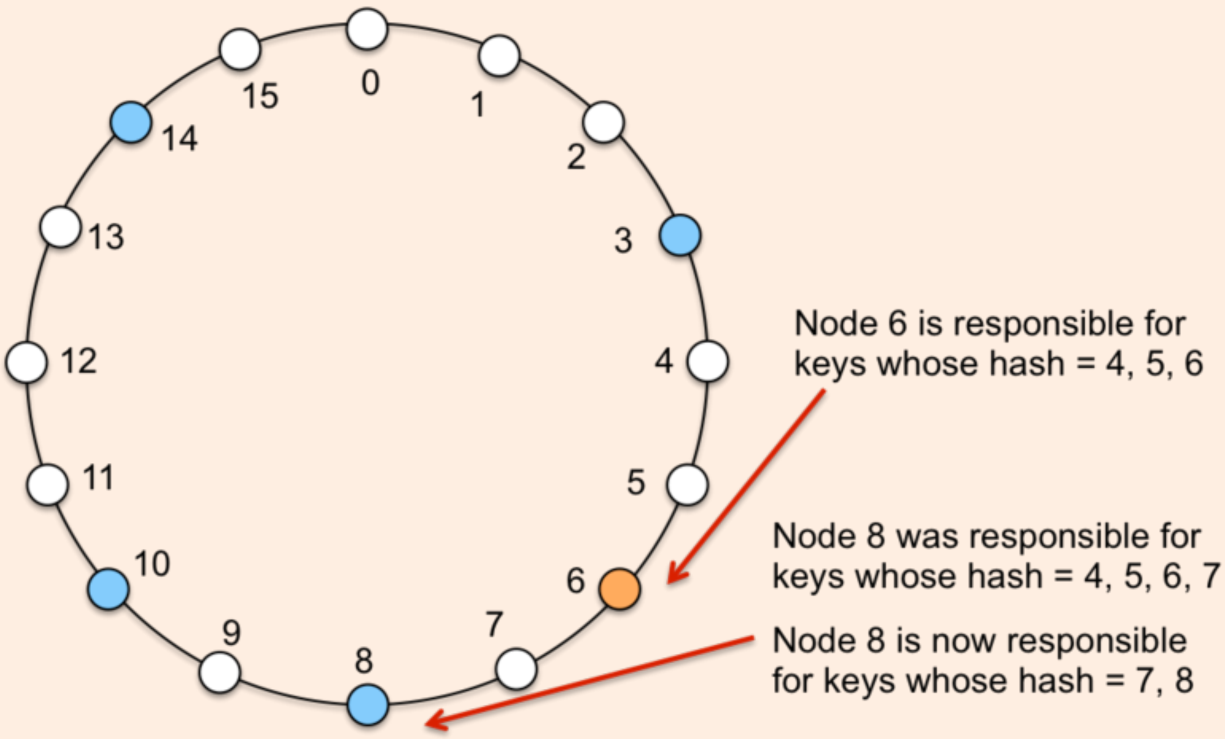


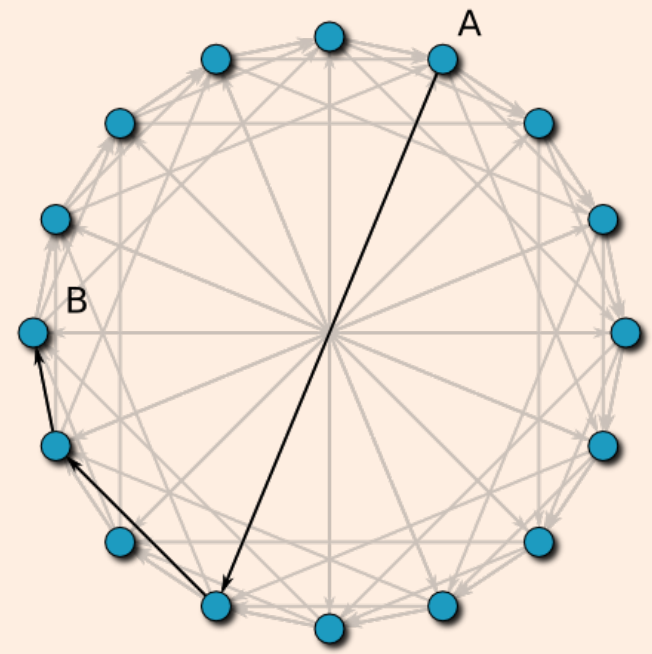
Diagram from: https://www.cs.rutgers.edu/~pxk/417/notes/23-lookup.html

Lookup and Insert Operations

- Lookup and insert operations are based on key ownership
  - When we want to find a key, we use the hash function to find its location on the ring
  - Given the location of a key on the ring, Chord allows us to efficiently identify its successor
  - For lookups, we ask the successor whether it knows about the key that we're interested in
    - Application-layer is responsible for further logic
  - For insert operations, we tell the successor to store the given key

Network Stabilization

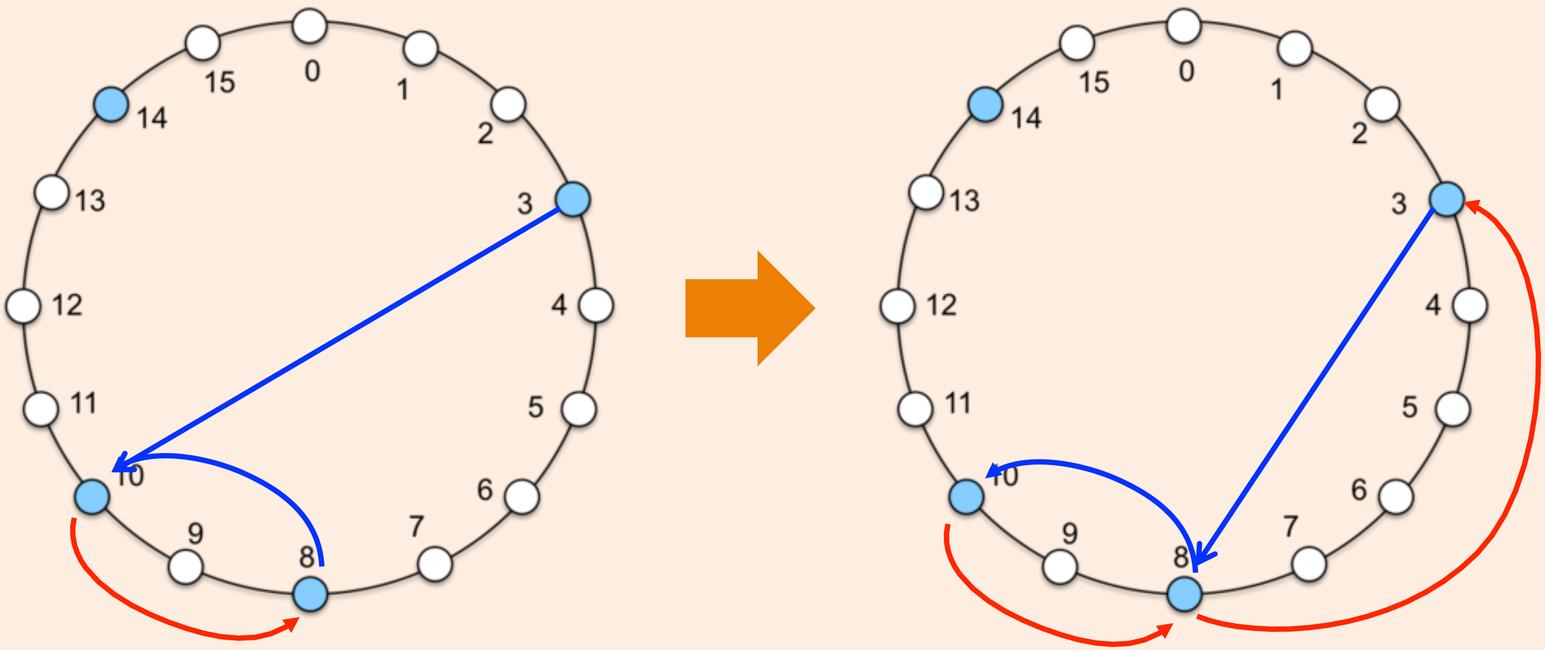
- Each node needs to maintain pointers to other nodes
  - Successor(s)
  - Predecessor
  - Finger table
- Finger table is a list of nodes at increasing distances from the current node
  - E.g. n, n+1, n+2, n+4, …
  - Allows for shortcuts across segments of the network, hence the name Chord



Finger tables allow lookup operations to take shortcuts across the key-space

Diagram from: https://en.wikipedia.org/wiki/Chord_(peer-to-peer)

Visualizing Stabilization



- Node 3 joins network, with node 10 as its initial successor
- Node 3 begins stabilization; asks node 10 for its predecessor
- Node 10 tells node 3 that its predecessor is node 8
- Node 8 is closer to node 3, and becomes its new successor
- Node 3 notifies node 8 of the change, so node 3 updates its predecessor pointer

Diagram adapted from: https://www.cs.rutgers.edu/~pxk/417/notes/23-lookup.html

Chord algorithms

- A Chord network can be thought of as a *dynamic* distributed data structure
- The Chord *protocol* defines eight algorithms that are used to navigate and maintain this data structure:
  - CheckPredecessor
  - ClosestPrecedingNode
  - FindPredecessor
  - FindSuccessor
  - FixFingers
  - Join
  - Notify
  - Stabilize

We're going to focus on **Stabilization**

A quick demo

Code available at:
https://github.com/tristanpenman/chordial

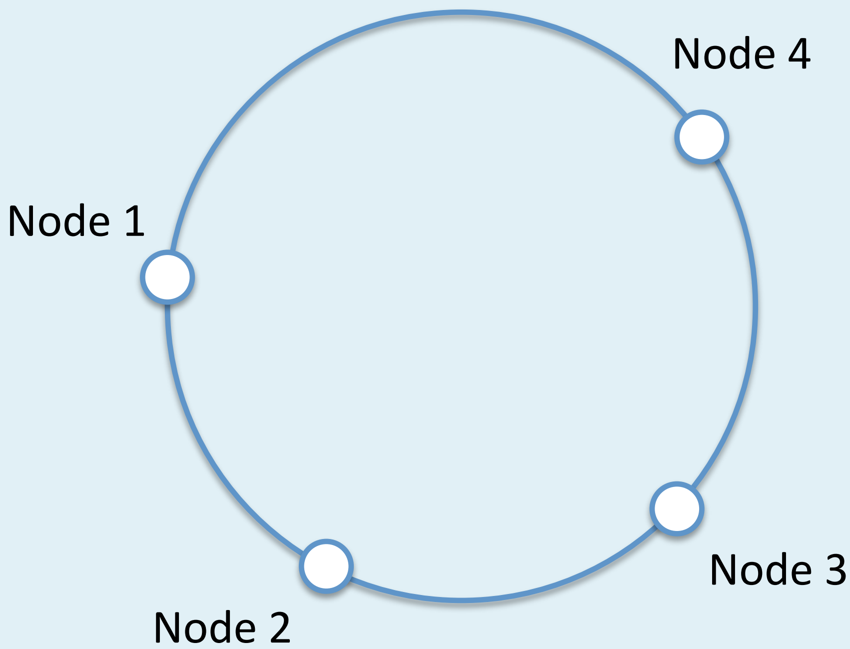
Modeling Chord using Actors

(while following Akka best practices)

Actor Model

- Computations defined in terms of individual components that respond to a set of well-formatted messages
- Group of actors is an Actor System
- Message Passing is asynchronous
- But nodes process messages sequentially

What (who) are our actors?



- Nodes are an obvious starting place…
  - Stores network pointers
  - Supports message nine types, along with the appropriate response messages
  - Timing logic for stabilization…
  - **This is starting to sound really complicated!**

Decomposition via Best Practices

- By identifying some Best Practices, we can take a more principled approach to decomposing our Actor System
- We want to preserve three key properties:
  - Determinism
  - Immutability and referential transparency
  - Thread-safety

Case study: Stabilization

- Chord requires that a node should periodically perform a stabilization operation
  - Stabilization ensures that the node’s successor is still the next closest node on the ring
  - If a closer node is found, then the successor pointer needs to be updated (involves a state change)
- Let’s look at how we might implement periodic stabilization using Scala and Akka…
  - Useful exercise to explore some Akka best practices

Stabilization… The Wrong Way

```
class Node(val initialSuccessor: ActorRef) extends Actor {
  var successor: ActorRef = initialSuccessor

  def doStabilization: Future[ActorRef] = ???

  override def receive: Receive = {
    case Stabilize() =>
      val newSuccessorFuture = doStabilization()
      newSuccessorFuture.onSuccess { newSuccessor =>
        successor = newSuccessor
      }
  }

  context.system.scheduler.schedule(3000.milliseconds,
    3000.milliseconds, self, Stabilize())
}
```

- Actor state should only ever evolve in response to messages received from the outside
  - Internal scheduling makes an actor's state non-deterministic, which is particularly bad for testing
  - Scheduling should take place *outside* the actor

```
class MyActor extends Actor {
  var counter = 0

  override def receive: Receive = {
    case IncrementCounter() =>
      counter += 1
  }

  context.system.scheduler.schedule(2.seconds, 3.seconds,
    self, IncrementCounter())
}
```

```scala
class MyActor extends Actor {
  var counter = 0
  override def receive: Receive = {
    case IncrementCounter() =>
      counter += 1
  }
}

object MyActor {
  case class IncrementCounter()
}

class MyApp extends App {
  val myActor =
    context.actorOf(Props(classOf[MyActor]))

  system.scheduler.schedule(2.seconds, 3.seconds,
    myActor, IncrementCounter())
}
```

- Actor state should only ever be mutated with a call to `context.become`
  - Using vars (or vals for *mutable objects*) allows unintended states to be introduced
  - Prefer immutability and referential transparency

```
class MyActor extends Actor {
  val isInSet = mutable.Set.empty[String]

  override def receive: Receive = {
    case AddToSet(key) =>
      isInSet += key

    case Contains(key) =>
      sender() ! isInSet(key)
  }
}
```

```
class MyActor extends Actor {

  def activeSet(isInSet: Set[String]): Receive = {
    case Add(key) =>
      context.become(activeSet(isInSet + key))

    case Contains(key) =>
      sender() ! isInSet(key)
  }

  override def receive: Receive = active(Set.empty)
}
```

- Actor state should not be allowed to leak into asynchronous closures
  - A Closure may be executed on another thread
  - Akka’s Context class is not thread-safe, which means no more calls to ‘context.become’:

```
class MyActor extends Actor {
  def withConfig(config: String): Receive = ???

  override def receive: Receive = {
    case Initialise() =>
      val myFuture = loadConfig()
      myFuture.onSuccess { config =>
        context.become(withConfig(config))
      }
  }
}
```

```scala
class MyActor extends Actor {
  def withConfig(config: String): Receive = ???

  override def receive: Receive = {
    case Initialise() =>
      val myFuture = loadConfig()
      myFuture.onSuccess { config =>
        self ! ConfigLoaded(config)
      }

    case ConfigLoaded(config) =>
      context.become(withConfig(config))
  }
}
```

Stabilization… Improved

```
class Node(val initialSuccessor: ActorRef) extends Actor {
  def doStabilization: Future[ActorRef] = ???

  def active(successor: ActorRef): Receive = {
    case Stabilize() =>
      val newSuccessorFuture = doStabilization()
      newSuccessorFuture.onSuccess { newSuccessor =>
        self ! Stabilized(newSuccessor)
      }
    case Stabilized(newSuccessor) =>
      context.become(active(newSuccessor))
  }

  override def receive: Receive = active(initialSuccessor)
}
```

Best Practices in Summary

- Here are the three Best Practices that we'll come back to while designing our Actor System:
    1. Actor state should only ever evolve in response to messages received from the outside
    2. Actor state should only ever be mutated with a call to `context.become`
    3. Actor state should not be allowed to leak into asynchronous closures

Based on best practices documented at: https://github.com/alexandru/scala-best-practices

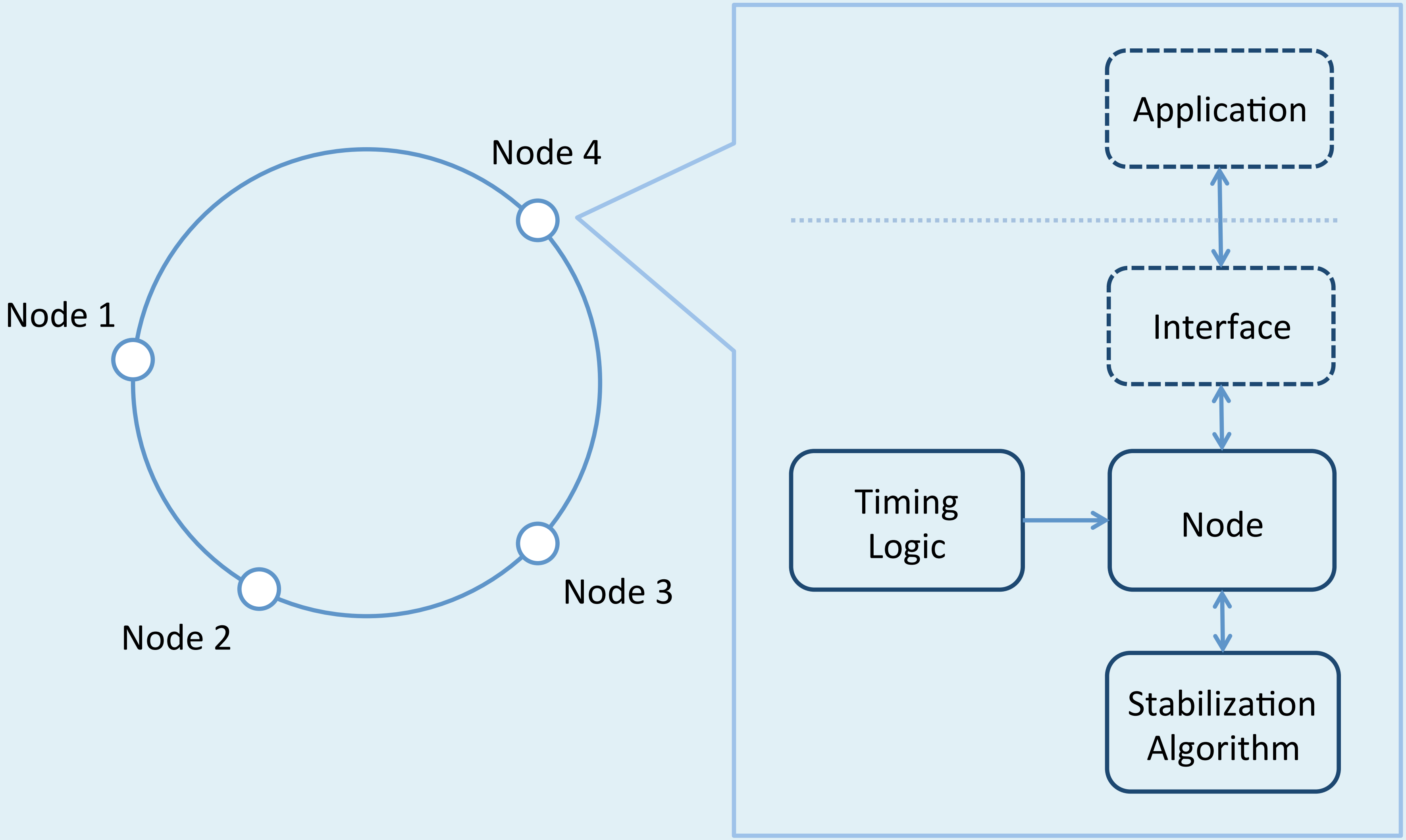
How can we decompose this actor?

- **<u>Lift the stabilization algorithm into its own actor</u>**
- Allows us to achieve our three best practices
  - “Actor state should only ever evolve in response to messages received from the outside” (determinism)
  - “Actor state should only ever be mutated with a call to context.become” (immutability and referential transparency)
  - “Actor state should not be allowed to leak into asynchronous closures” (thread-safety)

Timing Logic

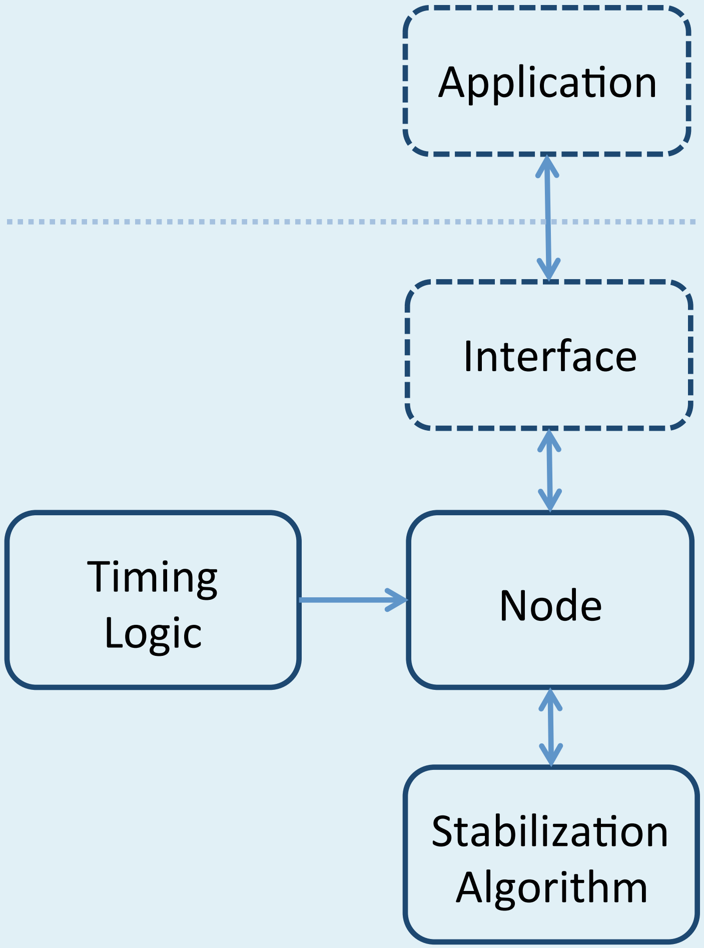
- **<u>We can also lift the timing logic into its own actor</u>**
- Once again, achieves our three best practices
  - In particular, ensures that the Node state only evolves in response to messages from the outside!

Our additional actors



Division of responsibilities

- Node handles requests
- Node stores network pointers:
  - Successor
  - Predecessor (optional)
  - Finger table (see Chord paper)
- TimingLogic triggers stabilization
- StabilizationAlgorithm runs asynchronous



Akka patterns
(Ask and Pipe)

Stabilization constraints

- Only one stabilization request should be in progress at any given time
- Stabilization should fail if the algorithm exceeds a given timeout
- When stabilization finishes, a notification should be sent to the TimingLogic actor
  - so that it can adjust the frequency of stabilization requests based on time-to-complete or failure rate

Ask pattern

- Ask pattern (import akka.pattern.ask)
  - Ask (?) instead of tell (!)
  - Returns a Future that will complete with the first response from the target actor
  - Takes a Timeout value, which specifies how long to wait until the Future should fail
  - This can be nicer than setReceiveTimeout

- *"Stabilization should fail if the algorithm exceeds a given timeout"*

Ask pattern example

- Asking a node for its ID:

```
val nodeIdFuture = nodeRef.ask(GetId())(requestTimeout)
  .mapTo[GetIdResponse]
  .map {
    case GetIdOk(nodeId) =>
      Some(NodeInfo(nodeId, nodeRef))
  }
  .recover {
    case ex =>
      log.error(s"GetId failed: ${ex.getMessage}")
      None
  }
```

Pipe pattern

- Pipe pattern (import akka.pattern.pipe)
  - Complements the Ask pattern by allowing the result of a Future to be piped to an actor
  - Augments Futures with the pipeTo method
    - **onSuccess** -> sends result to actor
    - **onFailure** -> sends exception to actor, as an akka.actor.Status.Failure

- *"When stabilization finishes, a notification should be sent to the TimingLogic actor"*

Combining Ask and Pipe patterns

- Requests that depend on the output of an asynchronous algorithm will create (or reuse) an algorithm actor
  - Using the Ask pattern gives us a Future that will expire after a fixed amount of time
  - Transform and ‘pipe’ result to client that originally made the request
  - Allows async request handling to take place outside of the main thread

Stabilization Using Ask and Pipe

```
class Node(val initialSuccessor: ActorRef) extends Actor {
  def running(nodeRef: ActorRef, algorithm: ActorRef): Receive = {
    case Stabilize() =>
      algorithm.ask(StabilizationStart(nodeRef))(timeout)
        .mapTo[StabilizationStartResponse]
        .map {
          case StabilizationComplete() => StabilizeOk()
          case StabilizationAlreadyRunning => StabilizeInProgress()
          case StabilizationFailed(m) => StabilizeFailed(m)
        }
        .recover { case ex => StabilizeFailed(ex.getMessage) }
        .pipeTo(sender())
  }

  override def receive: Receive = running(
    context.actorOf(Props(classOf[Node], initialSuccessor)),
    context.actorOf(Props(classOf[StabilizationAlgorithm])))
}
```

Bonus tip: Use Sealed Traits

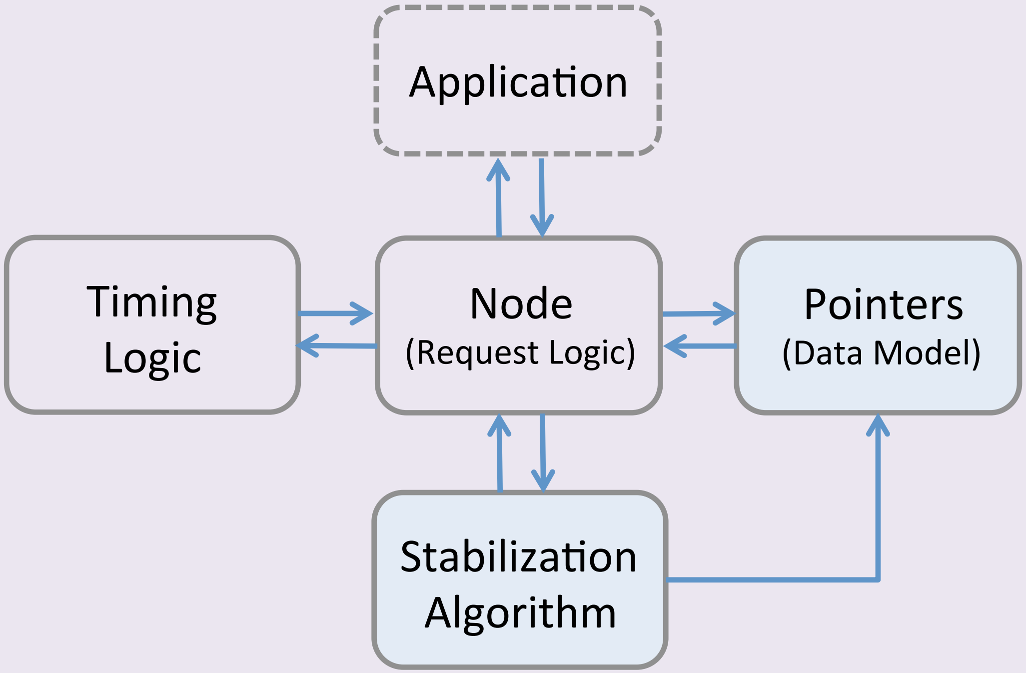
- Model your message types using Sealed Traits
  - Allows the Scala compiler to validate the exhaustiveness of pattern matching

```
algorithm.ask(StabilizationStart(nodeRef))(timeout)
    .mapTo[StabilizationStartResponse]
    .map {
        case StabilizationComplete() => StabilizeOk()
        case StabilizationAlreadyRunning => StabilizeInProgress()
        case StabilizationFailed(m) => StabilizeFailed(m)
    }
    .recover { case ex => StabilizeFailed(ex.getMessage) }
    .pipeTo(sender())
```

An interesting edge case

- Allowing concurrent stabilization requests could lead to sub-optimal network behaviour
  - Maintain one instance of the StabilizationAlgorithm actor
  - While running, it will respond with an AlreadyInProgress message for any attempt to restart it
- When joining a new network, algorithm actors are terminated and replaced with new actors. However, messages from old actors may still be queued!
- So once we join a new network, we want need a way to ignore any queued messages that may alter network pointers
  - Accepting these messages could lead to the Node being split across two networks

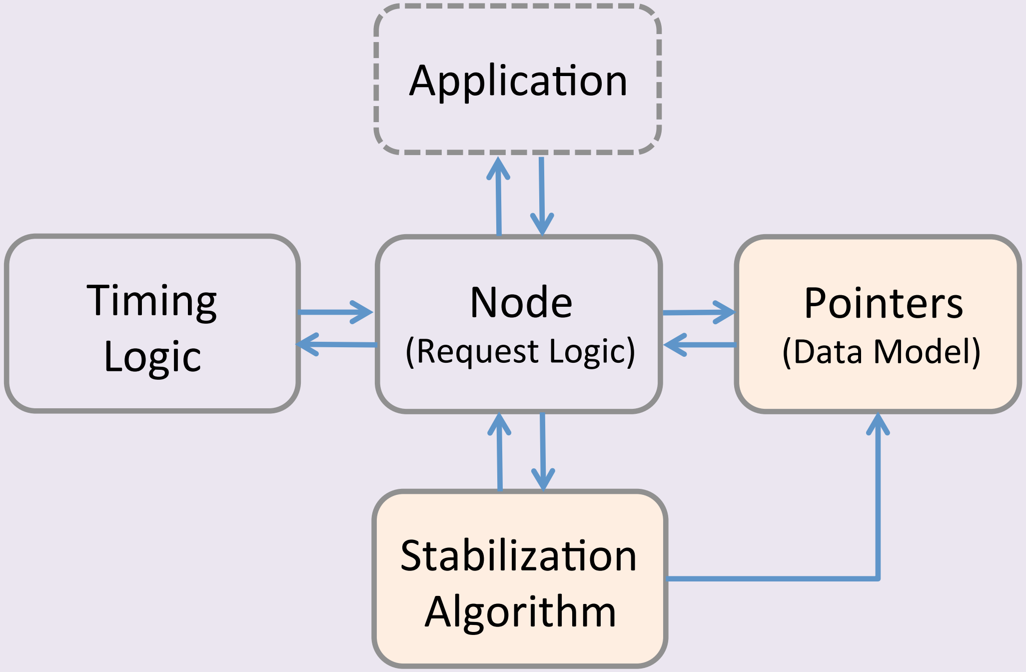
Possible solution (before Join)



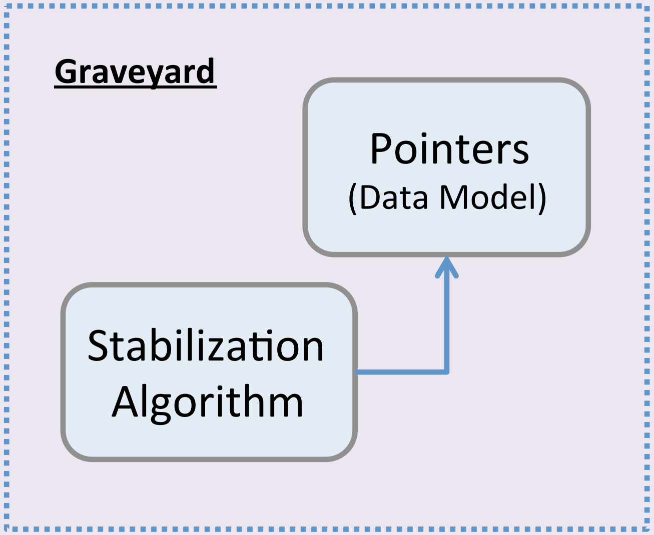
- Join request causes Pointers and StabilizationAlgorithm actors to be stopped
- context.stop achieves this using a message send, so other messages may be in their queues
- New Pointers and StabilizationAlgorithm actors are created
- Old actors are effectively detached and cannot alter state of Node

Not sure that this is the best approach…

Possible solution (*after* Join)



- Node becomes a proxy / controller for Pointers actor
  - Maybe even the 'Interface'?
  - Pointers actor becomes model
- StabilizationAlgorithm sends update messages to Pointers actor



Not sure that this is the best approach…

Scala resources

- Useful resources for Scala and Akka:
  - *Principles of Reactive Programming* course on Coursera
    https://www.coursera.org/course/reactive
  - Functional Programming in Scala
    (book by Chiusano and Runar)
  - Scala Best Practices (includes some Akka best practices)
    https://github.com/alexandru/scala-best-practices

Papers

- "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications" (IEEE Transactions on Networking version) https://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf

- "Consistent Hashing and Random Trees- Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web", http://www.ccs.neu.edu/home/cbw/4700/papers/akamai.pdf

Libraries used

- Backend:
  - spray (library for building Akka-based web services)
    https://github.com/spray/spray
  - spray-json (JSON de-/serialisation, spun off from spray)
    https://github.com/spray/spray-json
  - spray-websocket (Stream data over HTTP - this used to be really hard!)
    https://github.com/wandoulabs/spray-websocket
  - scalastyle (detect code smells, formatting errors, etc)
    http://www.scalastyle.org

- Frontend:
  - d3.js (bring data to life using HTML, SVG and CSS)
    https://github.com/mbostock/d3

Thanks for listening

Time for questions!

Email:
tristan@tristanpenman.com

Demo code available at
https://github.com/tristanpenman/chordial