Les secrets de l'éclair

Recipe for the perfect Lightning Node

Fabrice Drouin <fabrice.drouin@acinq.fr> - https://acinq.co/

Timeline

Lightning Network
(J.Poon, T.Dryja)

Feb. 2015

Nov. 2015

Reaching the ground with LN
(R.Russell)

LN specs
workshop in Milan

Oct 2016

May 2017

First “real” payment on LTC

BOLTs V1.0 RC
Interop tests
1st tx on mainnet

Dec 2017

April 2018

All 3 major impls on mainnet

LN summit Adelaide

Nov 2018

Feb 2018

BOLTs V1.0

- LN is an open source protocol
- At least 5 different, open-source implementations (c-lightning, eclair, lnd, lit, ptarnigan)
- Common specification is now at 1.0

Basis Of Lightning Technology

- BOLT #1: **Base Protocol**
- BOLT #2: **Peer Protocol for Channel Management**
- BOLT #3: **Bitcoin Transaction and Script Formats**
- BOLT #4: **Onion Routing Protocol**
- BOLT #5: **Recommendations for On-chain Transaction Handling**
- BOLT #7: **P2P Node and Channel Discovery**
- BOLT #8: **Encrypted and Authenticated Transport**
- BOLT #9: **Assigned Feature Flags**
- BOLT #10: **DNS Bootstrap and Assisted Node Location**
- BOLT #11: **Invoice Protocol for Lightning Payments**

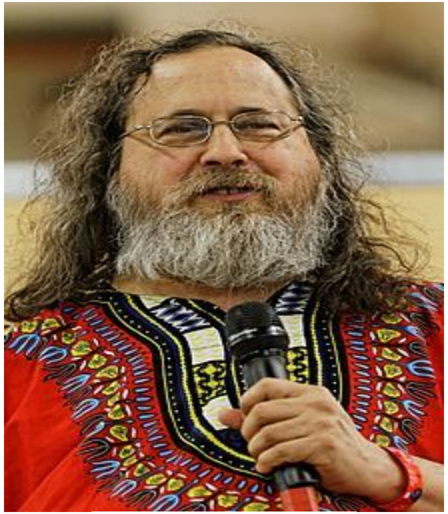
See https://github.com/lightningnetwork/lightning-rfc

The Lighting Network

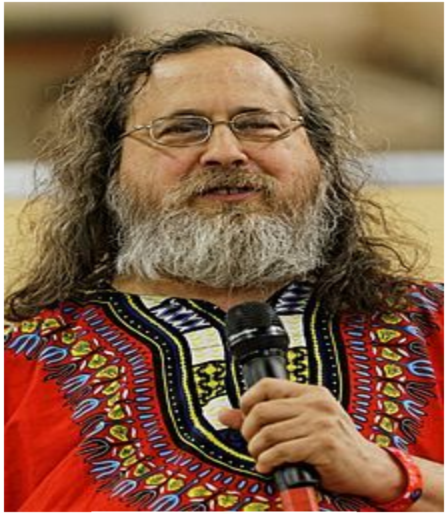
Several independent, open source implementations

- C-Lighting (C)
- Eclair (Scala)
- Lnd (Go)
- And Ptarmigan (C++), Rust-Lighting (Rust), LIT (Python), Electrum(Python)...

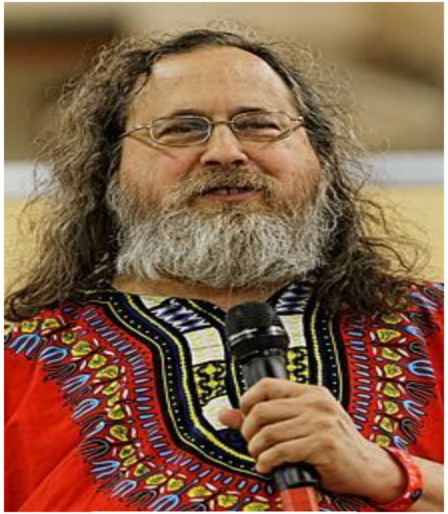
Multiple Lightning Implementations



Multiple Lightning Implementations



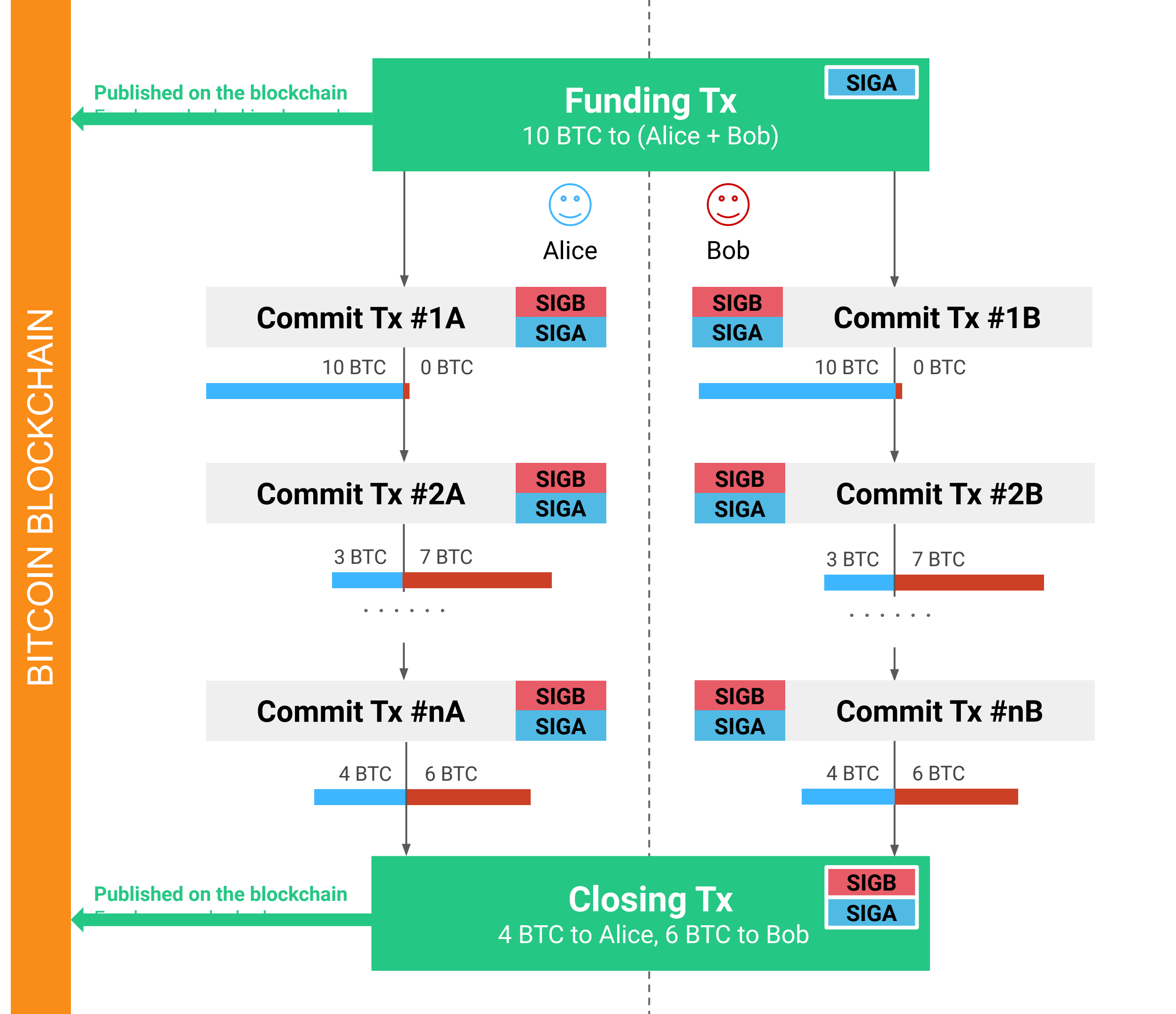
Multiple Lightning Implementations



Not really….

What is Lightning ?

- A peer-to-peer network of payment channels
- What is a payment channel ?
  - A 2-party entity that maintains how the output of an onchain transaction is spent
  - A opens a channel to B
  - A publishes a **funding transaction** that sends to (A + B)
  - A and B maintain an **unpublished commit transaction** that spends the funding transaction
  - A and B exchange messages to **update their commit transaction**
    - Payment model: HTLC
    - A pays for the preimage R of a hash H
    - After a delay A get their money back



OPEN

First tx is published on the blockchain.
Funds are “locked” in the channel

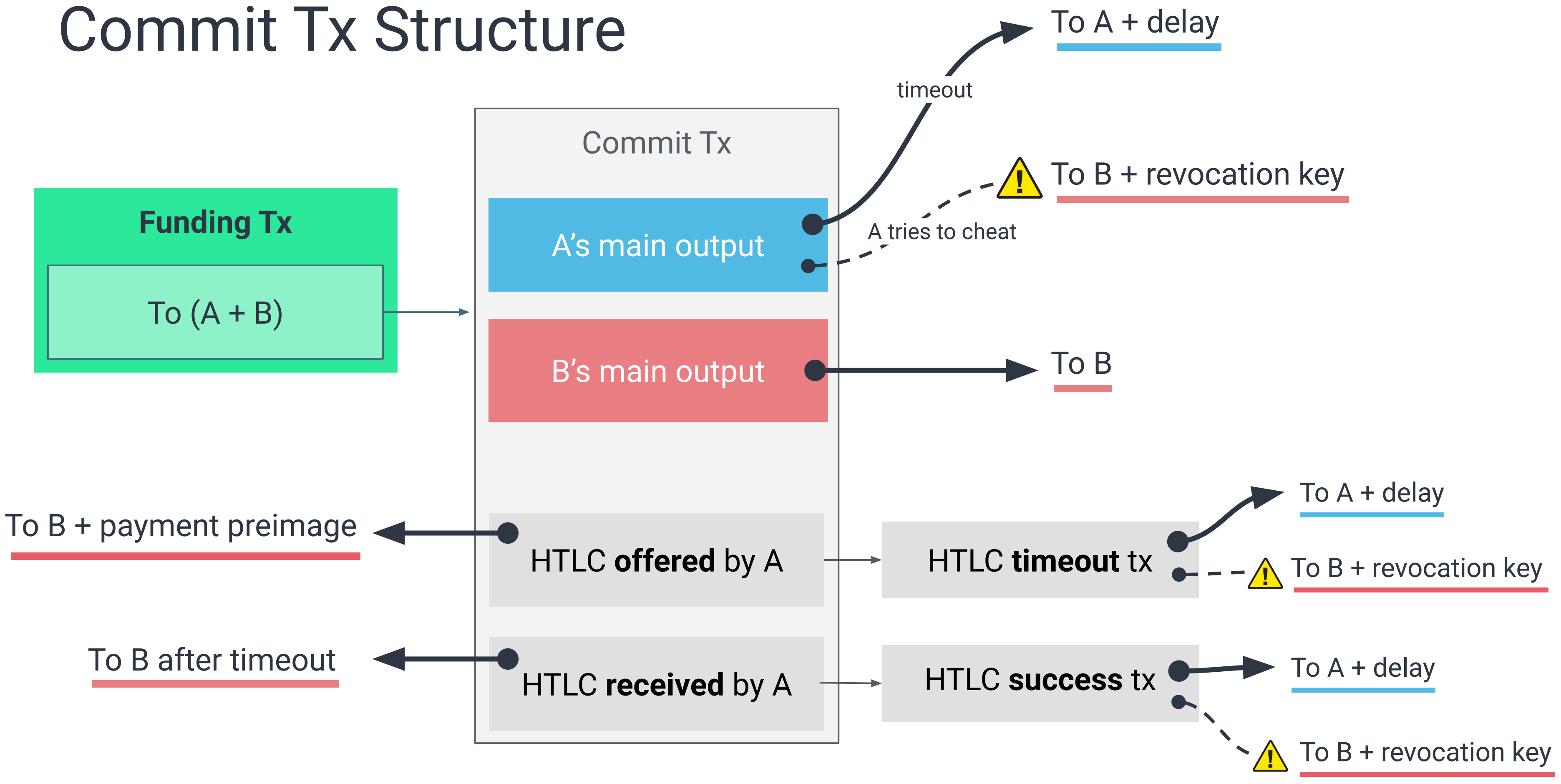
UPDATE

Publishable (signed by both parties) but **not published**!

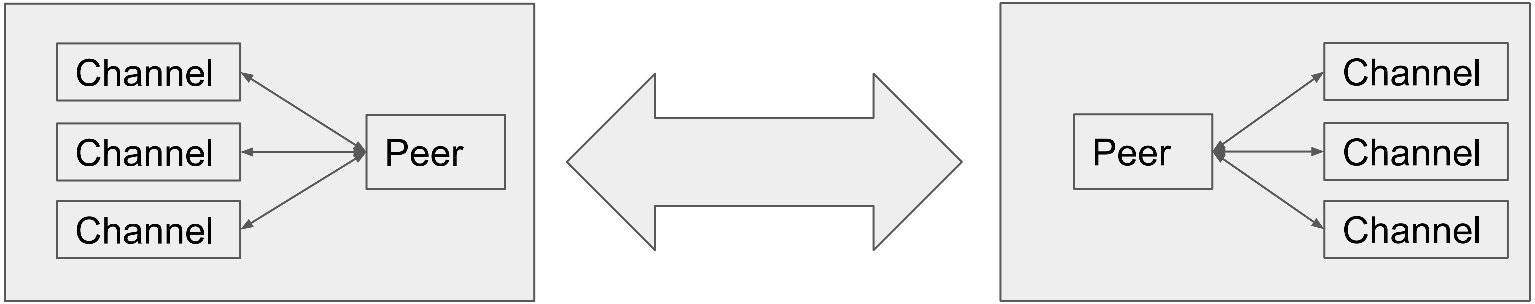
CLOSE

Last tx is published on the blockchain.
Funds are “unlocked”

Commit Tx Structure



P2P network of LN Channels



- Each node has a (public key, private key) pair, public key is used as **node id**
- Connections between Peers are **encrypted** and **authenticated**
- Channels are **multiplexed over a single connection** per peer

The Lightning Protocol

```
+-------+                               +-------+
|       |--(1)---- update_add_htlc ---->|       |
|       |--(2)---- update_add_htlc ---->|       |
|       |<-(3)---- update_add_htlc -----|       |
|       |                               |       |
|       |--(4)--- commitment_signed --->|       |
|   A   |<-(5)---- revoke_and_ack ------|   B   |
|       |                               |       |
|       |<-(6)--- commitment_signed ----|       |
|       |--(7)---- revoke_and_ack ----->|       |
|       |                               |       |
|       |--(8)--- commitment_signed --->|       |
|       |<-(9)---- revoke_and_ack ------|       |
+-------+                               +-------+
```

The Lightning Protocol

- Nodes exchange message over an encrypted/authenticated transport
- Messages are ordered (but can get lost)
- Update protocol works in batch mode
    - Send many **updates**
    - **Sign** all your pending updates
    - Wait for a **revocation** which also serves as an Acknowledgement message
- Nodes maintain 2 commit transactions
    - local : their commit tx. This is what they revoke
    - remote: their view of their peer’s commit tx. This is what they sign

Channels as State Machines

- From a functional point of view, a channel is a state machine
- Everytime it receives a message, it
  - transitions to a new updated state
  - optionally sends back a message
- They also need to maintain origin/destination information about relayed payment
  - What is the incoming upstream channel ?
  - What is the outgoing downstream channel ?

Design Constraints TL;DR

A LN node is a server application that

- manages incoming and outgoing tcp connection
- pipes incoming messages to channel state machines
- has good cryptographic libraries
- has good bitcoin libraries

Bonus points:

- performance (CPU, memory, …)
- tooling (build, deploy, debug, …)
- monitoring
- portability

LN Dev Ultimate Toolbox

- A good runtime
  - Performance, monitoring, portability, …
- A good programming language
  - Expressive, readable
  - Built-in “good practices”
  - Libraries: networking, cryptographic primitives, logging, …
- A good concurrency model
  - Pipe messages from/to lots of channels

LN Dev Ultimate Toolbox

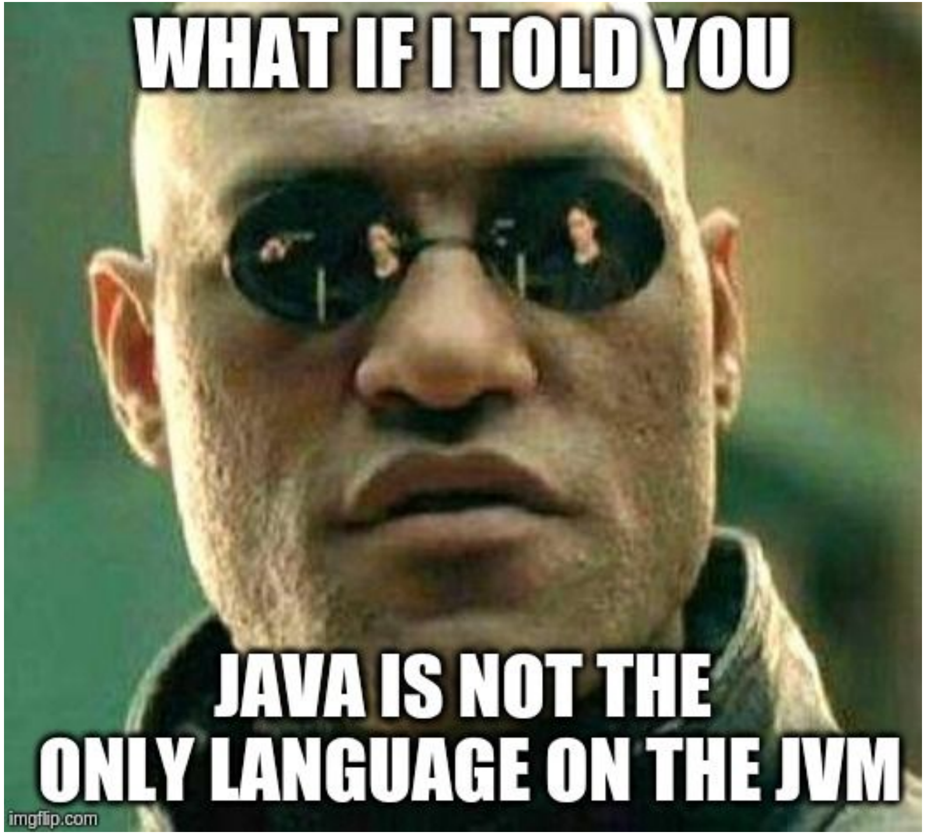
- A good runtime
  - Performance, monitoring, portability, …
- A good programming language
  - Expressive, readable
  - Built-in “good practices”
  - Libraries: networking, cryptographic primitives, logging, …
- A good concurrency model
  - Pipe messages from/to lots of channels

JVM

Scala +
Akka

The Java Virtual Machine

- One of the most **performant, reliable, and underrated** runtimes available today
- Runs almost everywhere
  - That includes phones, this how we build our mobile app
- Memory Management
  - Garbage collection (not your dad’s “stop the world” GC)
- Performance
  - JIT compiler: performance on par with non-optimized native code
  - Still too slow ? use native libraries (secp256k1 for example, which we include in our server and Android applications)
- Out-of-the box monitoring, profiling, debugging…
  - You monitor the JVM that runs your code, not your code


WHAT IF I TOLD YOU
JAVA IS NOT THE
ONLY LANGUAGE ON THE JVM
imgflip.com

The JVM Development Ecosystem

- JVM used on several billion devices
- Many different programming languages
    - Java, Scala, Clojure, Groovy, Kotlin… but also Python and Ruby
- Stable, battle-tested libraries for about everything
    - Usable from all JVM languages
    - Popular Java library = **millions** of users
- Great tooling
    - Build, dependency management, IDE, monitoring, debugging, deployment, ...

The Scala Programming Language

- Hybrid Object-Oriented/Functional Programming language
- Can be used as a Better Java or as a Worse Haskell :)
- Lots of goodies
  - Pattern Matching
  - Case classes
  - Immutability
  - Option type
  - Futures
- Very concise and expressive language
  - Java -> Scala: 5 to 10x less code !
- But still readable (*)
  - What is the code doing ?
  - What is the code **supposed** to be doing ?

The Actor Model

- Traditional Concurrency Model: Shared State + Locks
  - Contention issues
  - Deadlock issues
  - Very hard to debug
- Actor Model
  - Communication through **asynchronous message passing**
  - Messages are processed **one at a time, and sequentially**
  - State accessed through messages
  - **State never shared !**
  - Very easy to reason with
    - But not a generic solution to all problems

The Akka Actor Library

- Started as a port of Erlang OTP
- Includes Actor, FSM, tcp & http client & servers, logging, configuration, remoting, ...
- Akka alone is a good enough reason to give Scala s try

Show me the codz

```scala
case class Set(x: Int)
case class Add(x: Int)
case class Subtract(x: Int)
case object Get

class Calculator extends Actor {
 var state: Int = 0

 def receive = {
   case Set(x) =>  state = x
   case Add(x) => state = state + x
   case Subtract(x) => state = state - x
   case Get => sender ! state
 }
}
```

```scala
object Calculator extends App {

 val system = ActorSystem("mySystem")

 class Boot extends Actor {
   val calculator = context.actorOf(Props[Calculator])
   calculator ! Get
   calculator ! Set(42)
   calculator ! Add(1)
   calculator ! Get
   calculator ! Subtract(1)
   calculator ! Get

   def receive = {
     case state: Int => println(s"calculator state is $state")
   }
 }

 system.actorOf(Props[Boot])
}
```

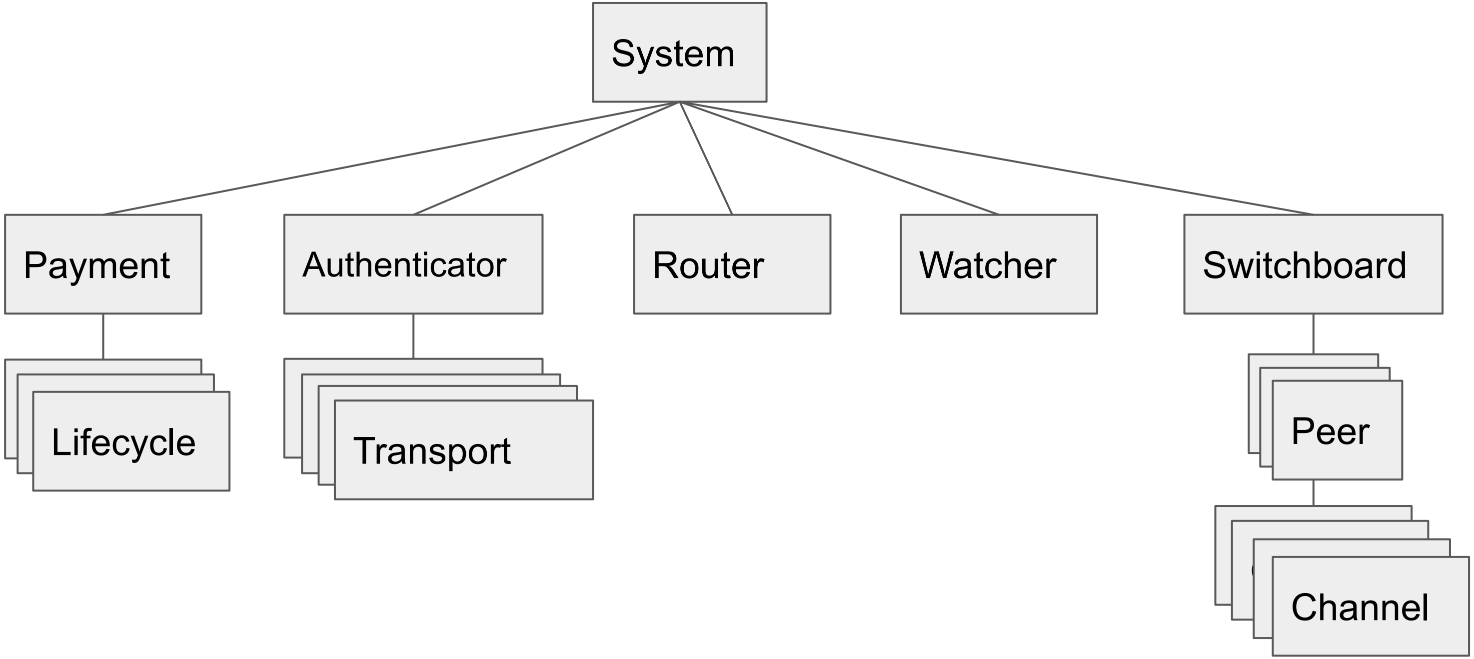
Show me the codz

```
case class Set(x: Int)
case class Add(x: Int)
case class Subtract(x: Int)
case object Get

class Calculator extends Actor {
 var state: Int = 0

 def receive = {
   case Set(x) =>  state = x
   case Add(x) => state = state + x
   case Subtract(x) => state = state - x
   case Get => sender ! state
 }
}
```

Look Ma No Locks !

```
object Calculator extends App {

 val system = ActorSystem("me")

 class Boot extends Actor {
   val calculator = context.actorOf(Props[Calculator])
   calculator ! Get
   calculator ! Set(42)
   calculator ! Add(1)
   calculator ! Get
   calculator ! Subtract(1)
   calculator ! Get

   def receive = {
     case state: Int => println(s"calculator state is $state")
   }
 }

 system.actorOf(Props[Boot])
}
```

Design Overview



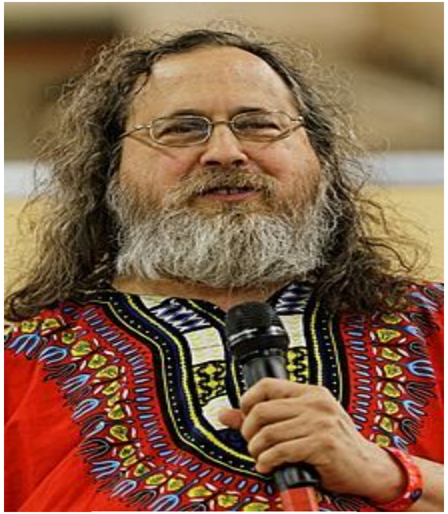
Eclair in Production

- ACINQ’s node is one of the largest and busiest nodes on Mainnet
  - 100s of peers
  - 1000s of channels
  - A lot of short-lived connection from mobile apps
- Runs on a single machine
- Reliable. It just runs
  - No db corruptions ever
  - Crashed once (in 18 months!) because of an (unreleased) “optimization”
  - Very easy to monitor
- Developed and maintained by a very small team

Recipe for the Perfect Lightning Node TL;DR:)

- A good runtime: the JVM!
  - It's * much * better than you think
- An expressive, precise, powerful language: Scala
  - Eclair is 22K LOC (+23K LOC of tests!) vs 150K for our friends c-lightning and lnd
  - OOP and functional
- An awesome Actor library: Akka
  - We've * never * had a concurrency bug/performance/memory issue
  - Very easy to reason with

Multiple Lightning Implementations



Go

Scala