Akka 分片集群的实现

- 以 spray-socktio 为例

QCon 北京 2014

@ 邓草原

豌豆荚

大数据 / 消息的实时流式处理

- 证券行业
  - 实时报价和成交回报驱动的分布式并行金融计算平台
  - 2006年：Erlang
  - 2008年-2012年：Scala
- 关注Process/Actor
- 豌豆荚
  - 实时消息流驱动的移动互联服务
  - 2013年：Scala/Akka
  - 首先要解决的问题
    - 事件、消息流的接入
    - 状态的保持
    - 业务逻辑的流式接口
  - 然后
    - 实时流式计算

spray-socketio

Actor Model

- 一种计算颗粒（粒度），本身包含：
  - 处理（行为）
  - 存贮（状态）
  - 通讯（消息）
- 三定则—当 Actor 接收到一条消息时，它可以：

  1.创建另外一些 Actors

  2.向已知的 Actors 发送消息

  3.指定接收下一条消息时的行为

Actor - 适合并行计算的最小颗粒

- 单个 Actor 的状态和行为只由接收到的消息驱动
- 单个 Actor 串行地处理接收到的消息
- → 单个 Actor 总是线程安全的

- 大量 Actors 同时处在活跃状态，其行为是并行的
- → 并行是多个 Actors 的行为

Entity(带状态) 应该是 Actor

- 可以按需即时加载到内存
- 可以设定 ReceiveTimeout 自动或主动从内存卸载
- 应该持久化所有对状态产生影响的事件（消息）
- 可以持久化状态快照
  - 按固定间隔持久化快照（分钟线、日线）
  - 从最近的快照恢复状态
- Entity = 状态快照 + 事件重演
- 不提倡 In-place Update 或者说修改持久化后的状态

Akka 的 Actor 实现

- Actor 是非常轻量的计算单元
  - 5000 万 / 秒消息转发能力（单机、单核、本地）
  - 250 万 Actors / GB 内存（每个空 Actor 约 400 多字节）
- Actor 位置透明，本身即具分布能力
  - 按地址创建和查找 - 本地或远程节点
  - 访问本地或远程节点仅在于地址 (Path) 不同
  - 可以跨节点迁移
- Actor 是按层级实现督导 (supervision) 的
  - Actor 按树状组织成层级
  - 父 Actor 监控子 Actor 的状态，可以在出状况时停止、重启、恢复它。

Akka 2.3.X (3 月 5 日发布)

- 分片集群 (Sharding Cluster) – Entity Actors
  - 按 Entity 的 ID 分片，按需自动在相应的节点创建
  - 消息按 ID 发送，由 Resolver 根据 ID 自动定位到 Actor 所在的 region(节点中) 并由 region 发送给 Actor
- 持久化 (Persistence) – 状态快照或事件历史
  - LevelDB （开发、测试）
  - HBase

分片 - IdExtractor / ShardResolver

```
type EntryId = String
type ShardId = String
type Msg = Any

type IdExtractor = PartialFunction[Msg, (EntryId, Msg)]
type ShardResolver = Msg => ShardId
```

分片 - IdExtractor / shardResolver

```
sealed trait Command extends Msg with Serializable {
  def sessionId: String
}

// cluster 按 sessionId 与 actor 一一对应，按需即时创建或定位转发
lazy val idExtractor: ShardRegion.IdExtractor = {
  case cmd: Command => (cmd.sessionId, cmd)
}

// cluster 依据 sessionId ，按一定规则，将 actor 分片到 Region
// 比如 100 个 regions ，cluster 会在每个节点分配若干个 Regions
lazy val shardResolver: ShardRegion.ShardResolver = {
  case cmd: Command =>
    (math.abs(cmd.sessionId.hashCode) % 100).toString
}
```

分片 – 带 EntryId 的消息

```
sealed trait Command extends Msg with Serializable {
  def sessionId: String
}

case class CreateSession(sessionId: String) extends Command
case class Connecting(sessionId: String, query: Uri.Query, origins: Seq[HttpOrigin],
                      transportConn: ActorRef, transport: Transport) extends Command

// called by connection
case class OnFrame(sessionId: String, frame: TextFrame) extends Command

// called by business logic
case class SendMessage(sessionId: String, endpoint: String, msg: String) extends Command
case class SendJson(sessionId: String, endpoint: String, json: String) extends Command
```

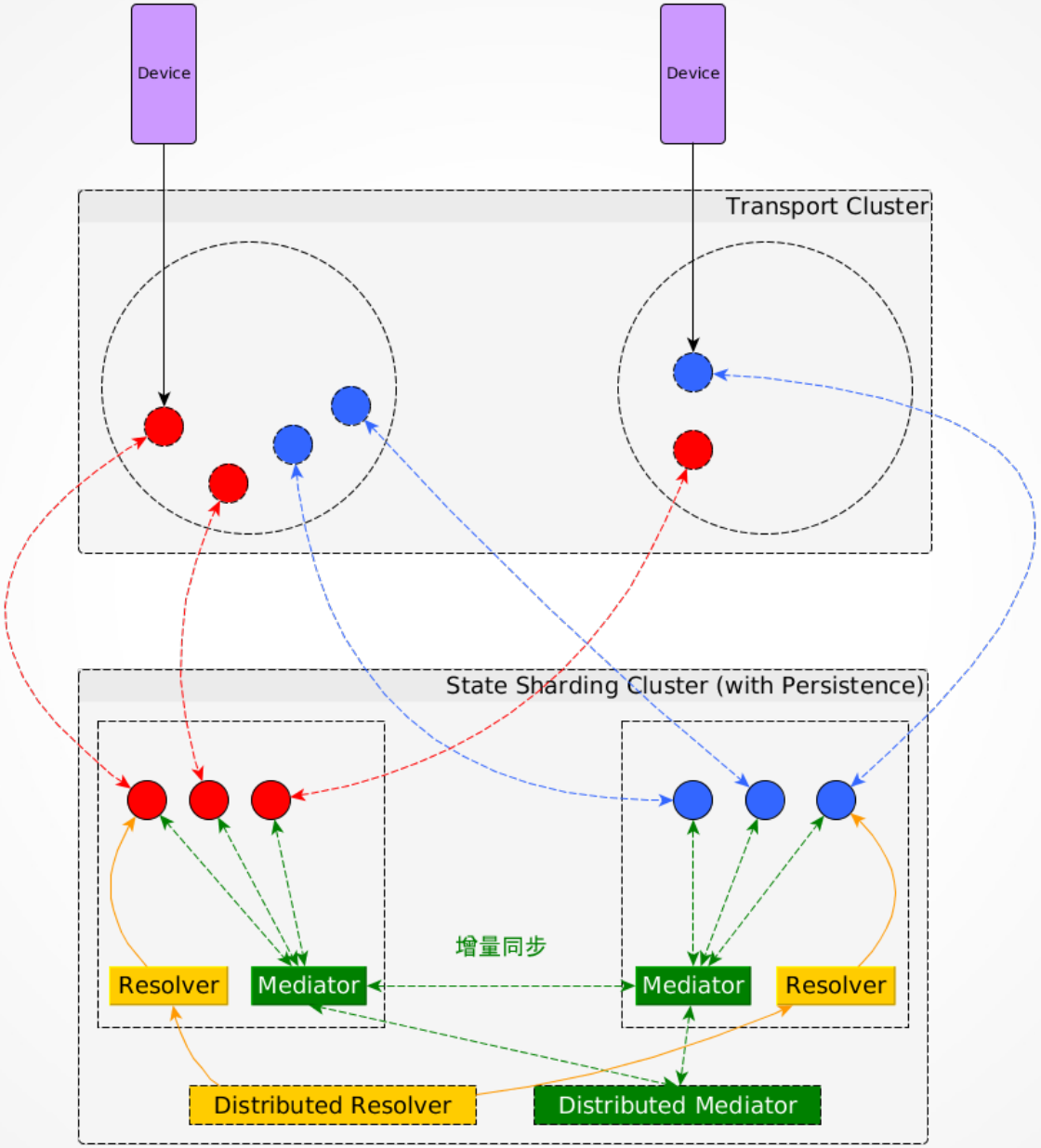
持久化 – 会改变状态的消息

```
sealed trait Event extends Serializable

case class Connected(sessionId: String, query: Uri.Query,
            origins: Seq[HttpOrigin],
            transportConn: ActorRef,
            transport: Transport) extends Event

case class UpdatePackets(packets: Seq[Packet]) extends Event
```

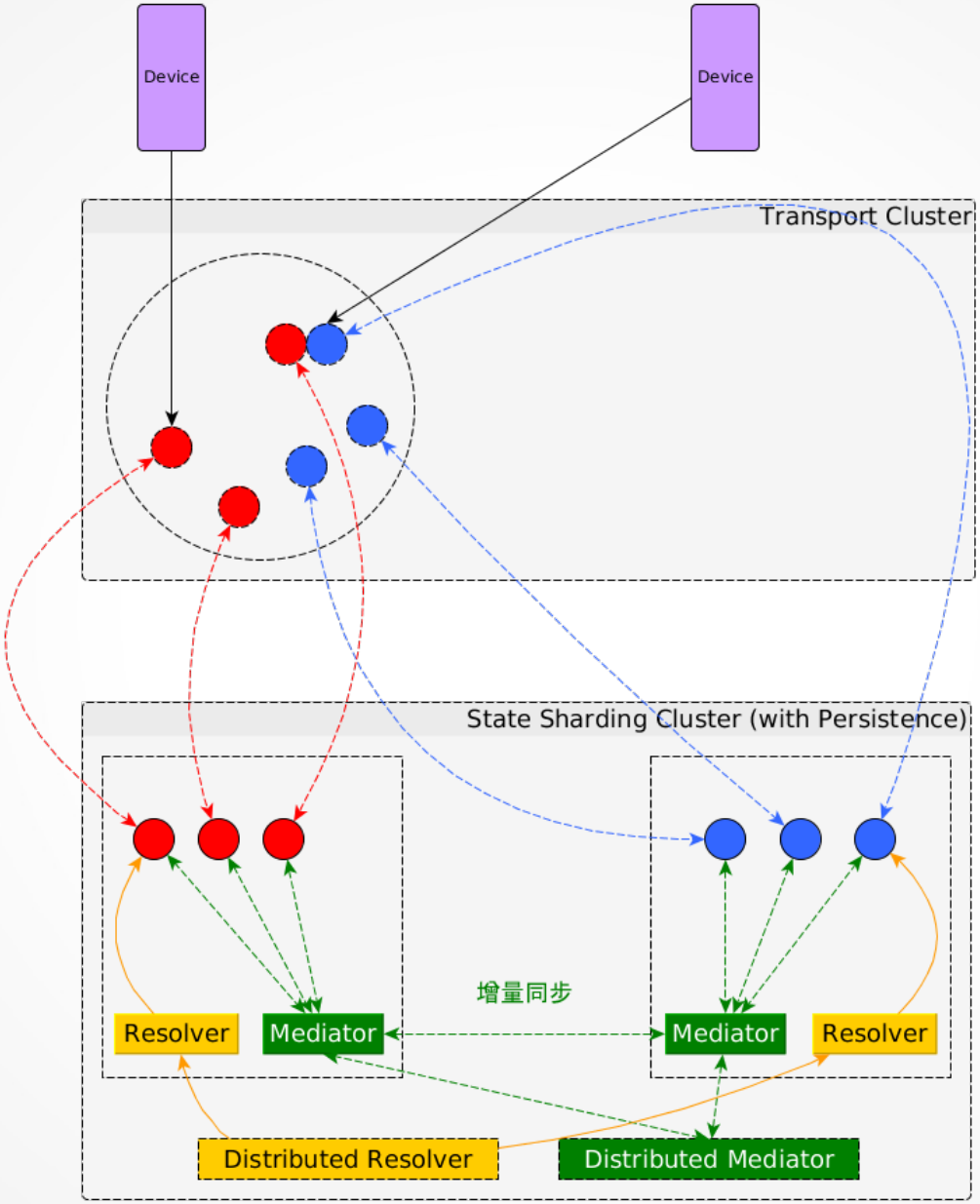
持久化 – persist / recover

```
class ClusterConnectionActive(val namespaceMediator: ActorRef,
                              val broadcastMediator: ActorRef) extends
      ConnectionActive with EventsourcedProcessor {

override def receiveRecover: Receive = {
  case event: Event => updateState(event) //  重演持久化的消息历史以恢复状态
}

//  只持久化会改变状态的消息
override def receiveCommand: Receive = {
  case connected: Connected =>
    persist(connected)(updateState(_))

  case packets: UpdatePackets =>
    persist(packets)(updateState(_))

  case _ => //  处理其它消息
}

def updateState(event: Event) = {
  event match {
    case x: Connected =>
      connectionContext.foreach(_.bindTransport(x.transport))
    case x: UpdatePackets =>
      pendingPackets = immutable.Queue(x.packets: _*)
  }
}
```

spray-socketio 集群 (2+ 层)



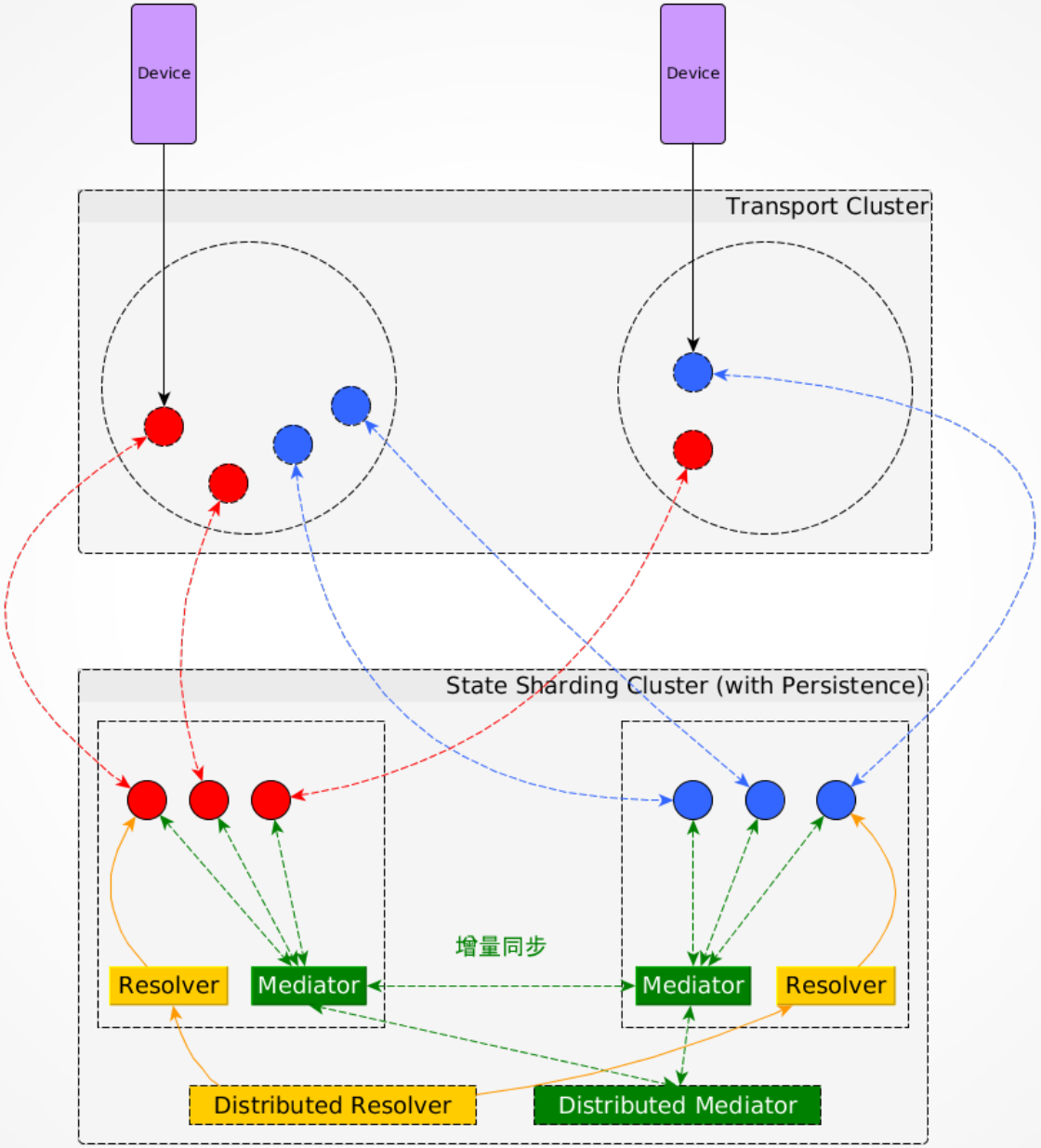
连接层节点挂掉


Device
Device
Transport Cluster
State Sharding Cluster (with Persistence)
增量同步
Resolver
Mediator
Mediator
Resolver
Distributed Resolver
Distributed Mediator

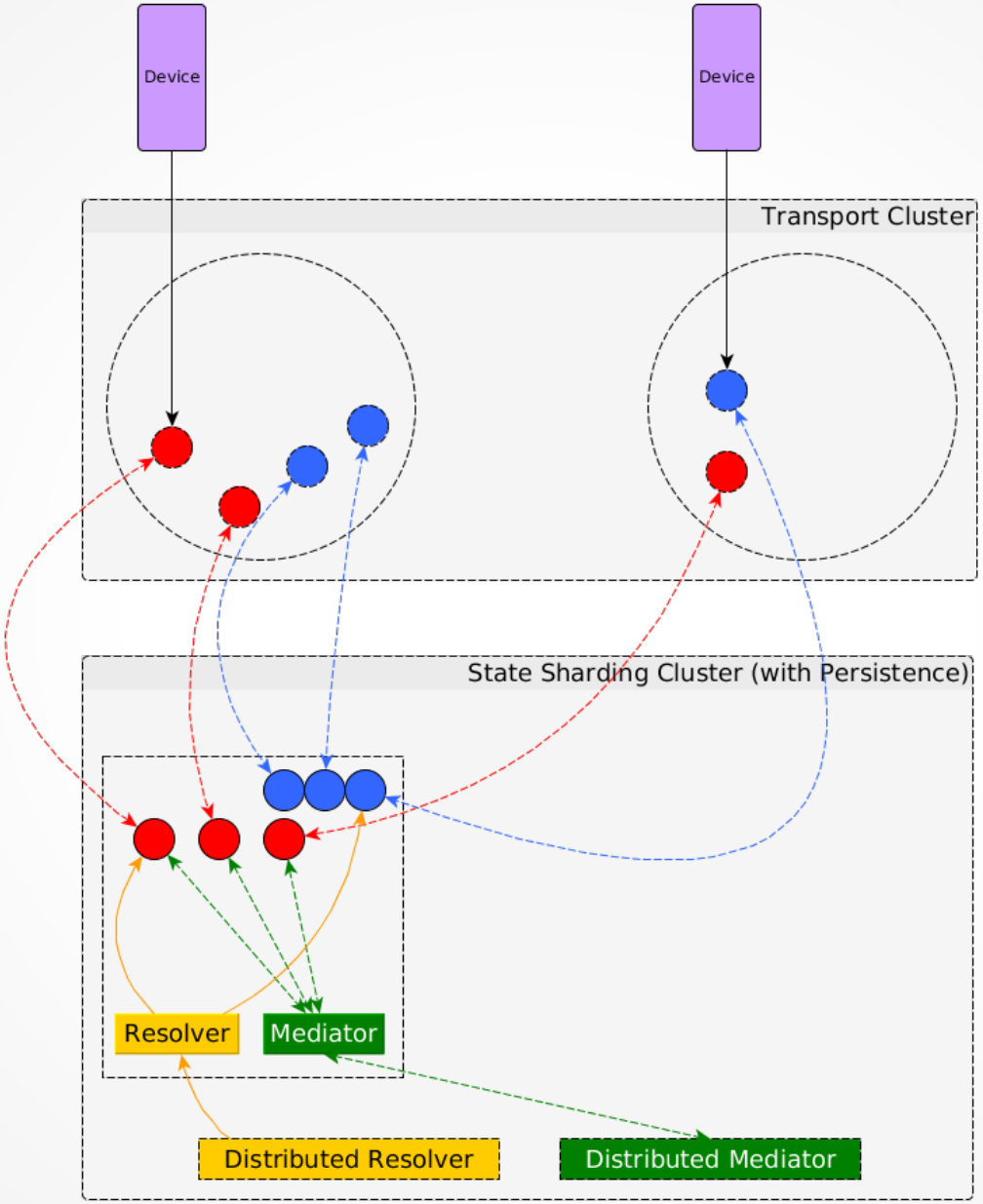
连接层节点挂掉

- 终端重新连接
- 在连接层正常的节点重建 Transport Actor
- 根据 SessionId 从状态层集群重置 Transport Actor 与 State Actor 的关联
- 所有场景恢复

spray-socketio 集群 (2+ 层)



状态层节点挂掉


Device
Device
Transport Cluster
State Sharding Cluster (with Persistence)
Resolver
Mediator
Distributed Resolver
Distributed Mediator

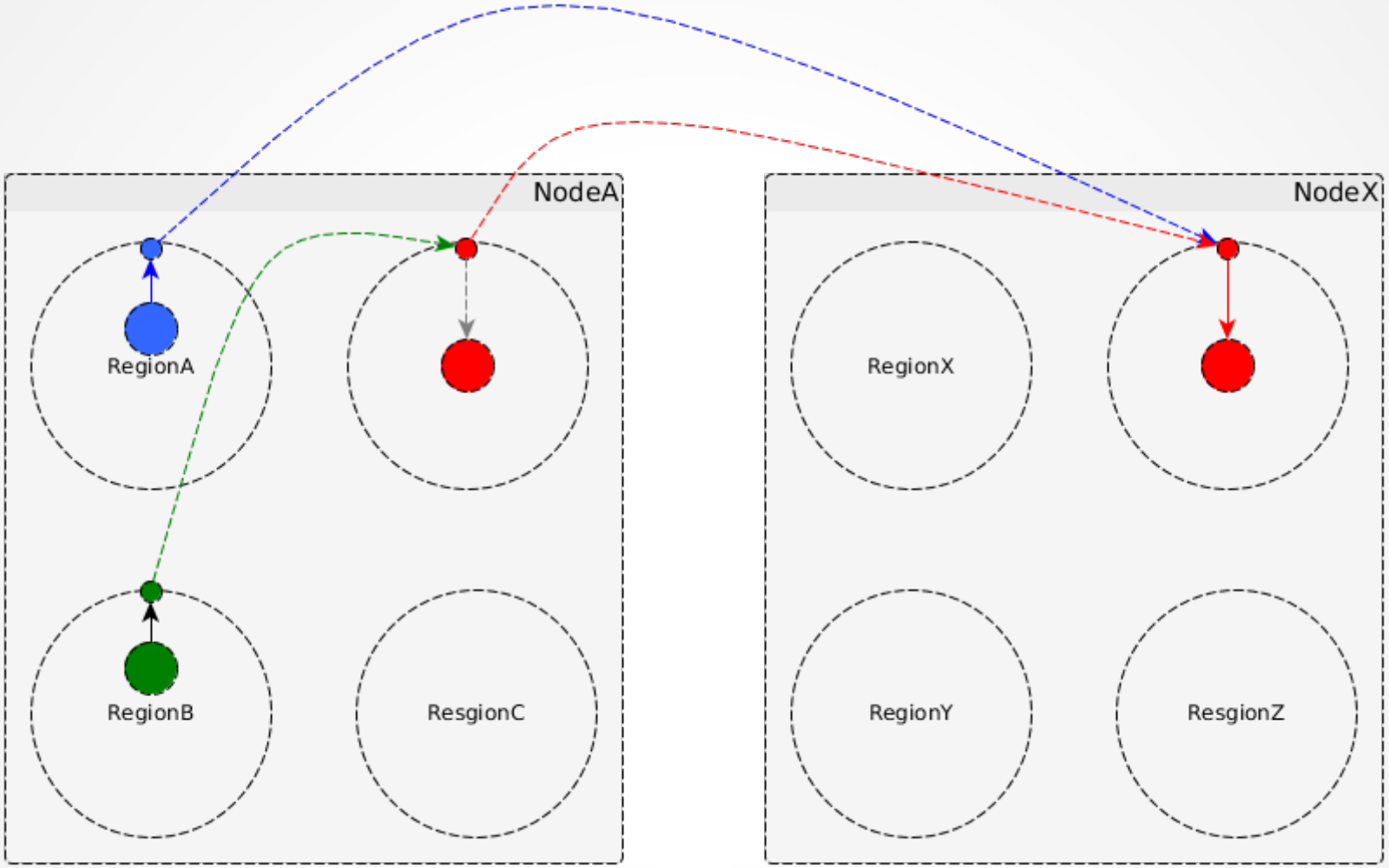
状态层节点挂掉

- 如果没有操作，状态层可以暂不加载相应 Actor
- 有操作，状态层选择正常的节点即时加载相应 Actor （首先建立的是 mailbox ），并暂存 (stash) 所有接收到消息。
- 从 Persistence 恢复状态（快照 + 事件重演）
- unstash ，处理暂存 (stash) 的消息和后续消息
- 进入正常状态

增加节点

- 各节点上的 actors 会根据分片规则迁移
- 所有发到 actors 的消息是通过所在的 shardRegion 转发的
  - 本 region 的直接转到目的 actor
  - 非本 region 的将转到目的 actor 所在的 region
- 要迁移的 actor 的迁移过程：
  - 在新节点的 region 新建一个 actor ，行为变为暂存 (stash) ，并从 persist 中开始 recover
  - 原 actor 所在的 region ：
    1. 确知新 actor 的创建 (在某节点的 region 上)
    2. 停掉原 actor
    3. 所有发送给原 actor 的消息转发到新 actor 所在 region （这是日常工作， region 的常态）
  - 已确知新 region 加入的 region ，直接发送到新的 region
  - 尚未确知而仍然发送到原 region 的消息，会由原 region 转发给新的 region
  - 新 actor 在从 persist 中 recover 后，行为转换为 unstash
- actor 位置透明 + 通过 ShardRegion 转发，使一切变得简单

新增节点



RegionA 确知新 Region ，直接访问
RegionB 未知新 Region ，通过原 Region 间接访问

可能丢失的消息

- 连接层节点或者状态层节点挂掉时
  - 业务逻辑通过状态层正在向终端发送的消息可能丢失
  - 终端正在发送的消息失败
- 规模在 1500 个节点的 Akka 集群，节点挂掉需要 40 秒通知到全局 (gossip)
- 总之：
  - Akka 只保证在 Actor 或节点挂掉时将迅速恢复该 Actor 或者节点上的所有 Actors
  - Akka 不保证在挂掉到恢复过程中正在发送的消息不丢失

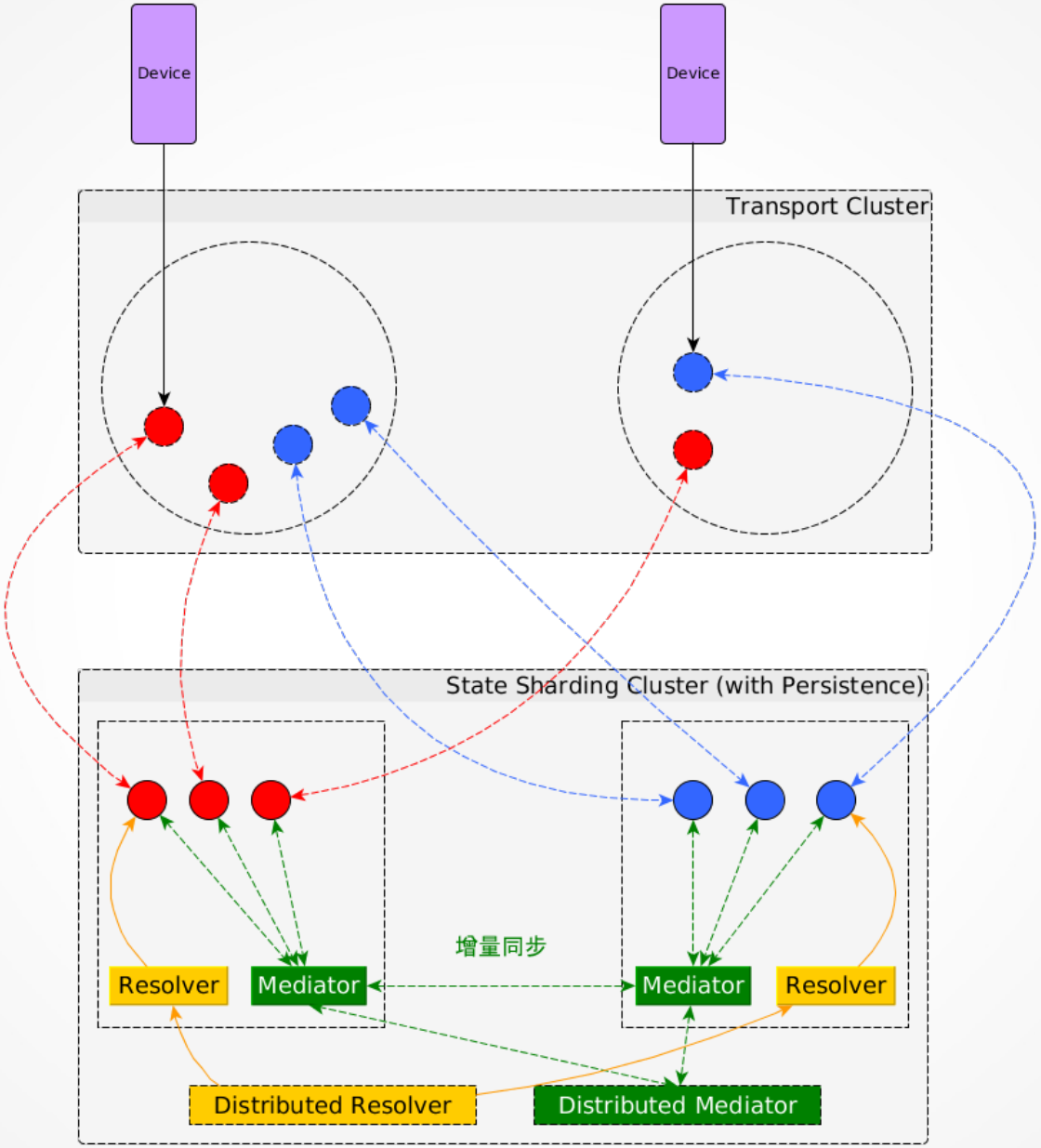
确保消息可靠传递？

- 有很多种场景，而且与业务逻辑和客户端的配合有关
  - 确保送达？
  - 确保按顺序送达？
  - 确保有且只有一次送达？
  - 确保有且只有一次按顺序送达？
  - 在指定时间内送达，超时重发？
  - 在指定时间内送达，超时丢弃？
- Akka 原则：
  - Supervised
  - let it crash
  - resume/recreate when necessary
- 由上层协议和逻辑负责实现(Ack，Transaction 等)
  - AMQP
  - 业务逻辑层的事务

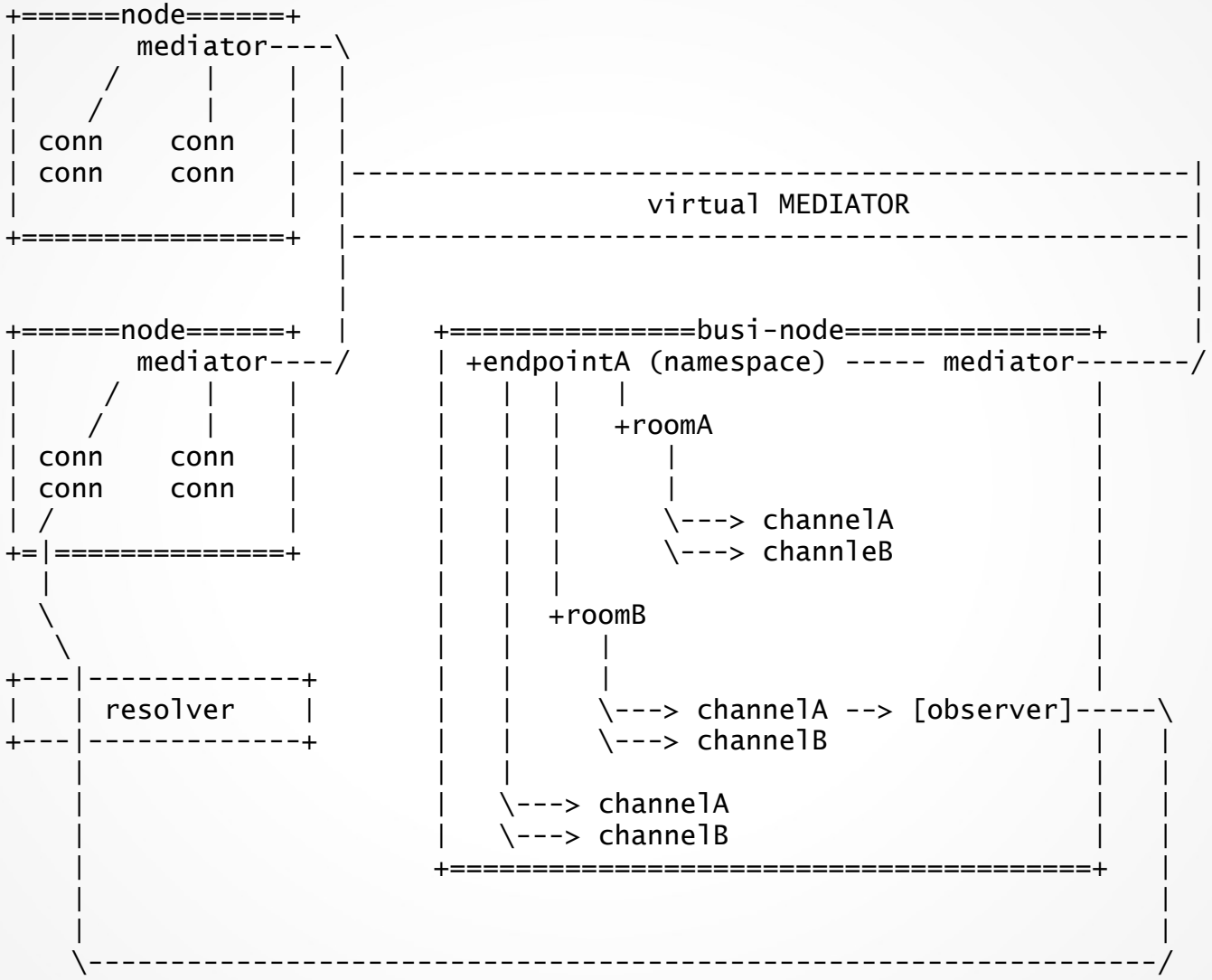
实时流式计算

- Reactive Programming
  - Data Flow（将数据看作有序传递的流）
  - Propagate of Change （变化按流传播）
  - Incremental Computation（各个计算单元依据变化作增量计算）
  - Querying on Flow （对流操作，map, filter, folder, combine etc）
- RxJava
  - 将消息/事件，错误，回调等抽象成Observable流
  - 可以对Observable流完成各种Querying操作，这些操作像SQL一样是可以进行依赖分析和优化的
    - select, select from a and b, group_by, map ...
    - 一种是针对已有数据的Pull形式的批量处理，一种是针对将至数据的Push形式的实时处理
  - Observable是Push流，可以是异步流
- Akka Reactive Stream
  - Actor是接受、处理、发送消息的单元
  - 但怎么表达“消息流”？
  - 受Rx启发的akka stream是回答

spray-socketio 集群 (2+ 层)



消息流


node
mediator
conn
conn
conn
conn
virtual MEDIATOR
node
mediator
conn
conn
conn
busi-node
endpointA (namespace)
mediator
roomA
channelA
channleB
roomB
resolver
channelA --> [observer]
channelB
channelA
channelB

与业务逻辑的接口 - RxJava

```
class Namespace(endpoint: String, mediator: ActorRef) extends Actor with ActorLogging {
  var channels = Set[Subject[OnData]]()

  def subscribeMediator(action: () => Unit) = {
    mediator.ask(Subscribe(endpoint, self))(timeout).mapTo[SubscribeAck].onComplete {
      case Success(ack) => action()
      case Failure(ex)  => log.warning("Failed to subscribe to mediator {}", ex)
    }
  }

  def receive: Receive = {
    case Subscribe(channel)   => subscribeMediator { () => channels += channel }
    case Unsubscribe(channel) => channels -= channel

    case OnPacket(packet: MessagePacket, connContext) =>
      channels foreach (_.onNext(OnMessage(packet.data, connContext)(packet)))
    case OnPacket(packet: JsonPacket, connContext) =>
      channels foreach (_.onNext(OnJson(packet.json, connContext)(packet)))
  }
}
```

业务逻辑示例

```
val observer = new Observer[OnEvent] {
  override def onNext(value: OnEvent) {
    case OnEvent("Hi!", args, context) =>
      if (value.packet.hasAckData) {
        value.ack("[]")
      }
      value.reply("welcome", List(Msg("Greeting")).toJson.toString)
  }
}

val channel = Subject[OnData]()
channel.ofType[OnEvent].filter(_.name == "Hi!").subscribe(observer)

namespaceExt.namespace("test") ! Namespace.Subscribe(channel)
```

性能分析 - 单服务器

- i7 4xcore 2.0GHz
- 消息大小 100 字节
- 每秒发送消息 4+ 万条
- 每秒推送消息 8+ 万条

性能分析 - 集群

- 集群需要将数据序列化和反序列化传输，这需要额外的处理时间，对性能有较大的影响
- 集群下，每个消息多了四次序列化的encode/decode：
  - 接收：Transport – en/decode -> ConnectionState - en/decode -> Business
  - 发送：Transport <- de/encode - ConnectionState <- de/encode - Business

- Encode/Decode消息是系统中最耗时的，单机下约占65~70%
- 单机时，这些消息分别各也经过了一次Encode/Decoder，将这个开销记为1，则在集群下开销变成1+4=5。因此，集群下，性能可能只到
  - 1 / ((1 + 4) x 70% + (1 - 70%)) = 26.3%
  - 单机能处理4万msg/s的话，集群下能达到的上限应在4 x 26% = 1万/s左右

- 集群内还会有额外的数据流量和处理，也会影响到性能
  - gossip的消耗，mediator的同步等
  - 1500个节点时，集群本身的心跳、状态等传输流量约100M bits/s，节点平均CPU占用率10%

性能分析 - 集群扩展能力

- 扩展有额外的消耗
- 随着节点的增加，扩展能力是否为线性是我们最关心的
- 设法通过各种调优手段将其尽量调整为接近线性
- 扩展能力采用线性回归做个简单预估
  - 1 -> 10k, 2 -> 15.8k, 3 -> 22.2k
  - Y=3.8+6.1n

性能分析 - 千万级连接集群

- 1 x 节点为 1 万 msg/s ， 100 万长连接
- 10 x 节点可以达到 10 x 100 万 = 1000 万长连接
- 10 x 节点消息处理能力：

  ➔3.8 + 6.1 x 10 = 64.8k
- 10 x 节点为
  - 6 万 / 秒 ,
  - 360 万 / 分
  - 52 亿 / 天

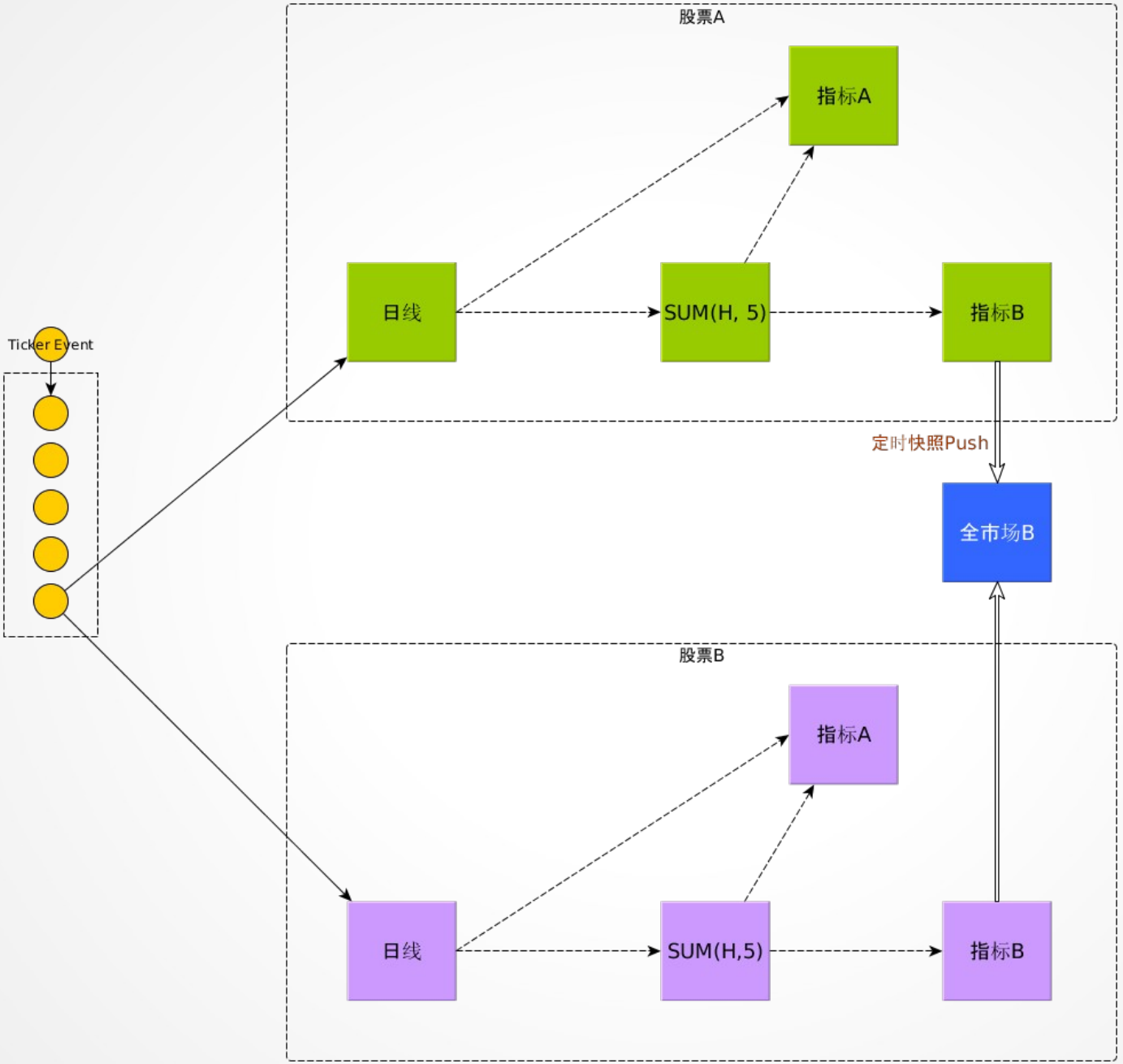
Reactive 分布式并行金融计算平台

- 实时响应式平台实现要点
  - 数据 In-Memory （可以按需载入）
  - 异步事件驱动
  - 变化依路径依赖传播
  - 增量计算
    - 只依每刻接收到的 ' 变化事件 ' 计算
    - 计算粒度 (单元) 尽量最小化，各算各的 - 也是并行的要求
  - 事件流 + 快照

Reactive 分布式并行金融计算平台

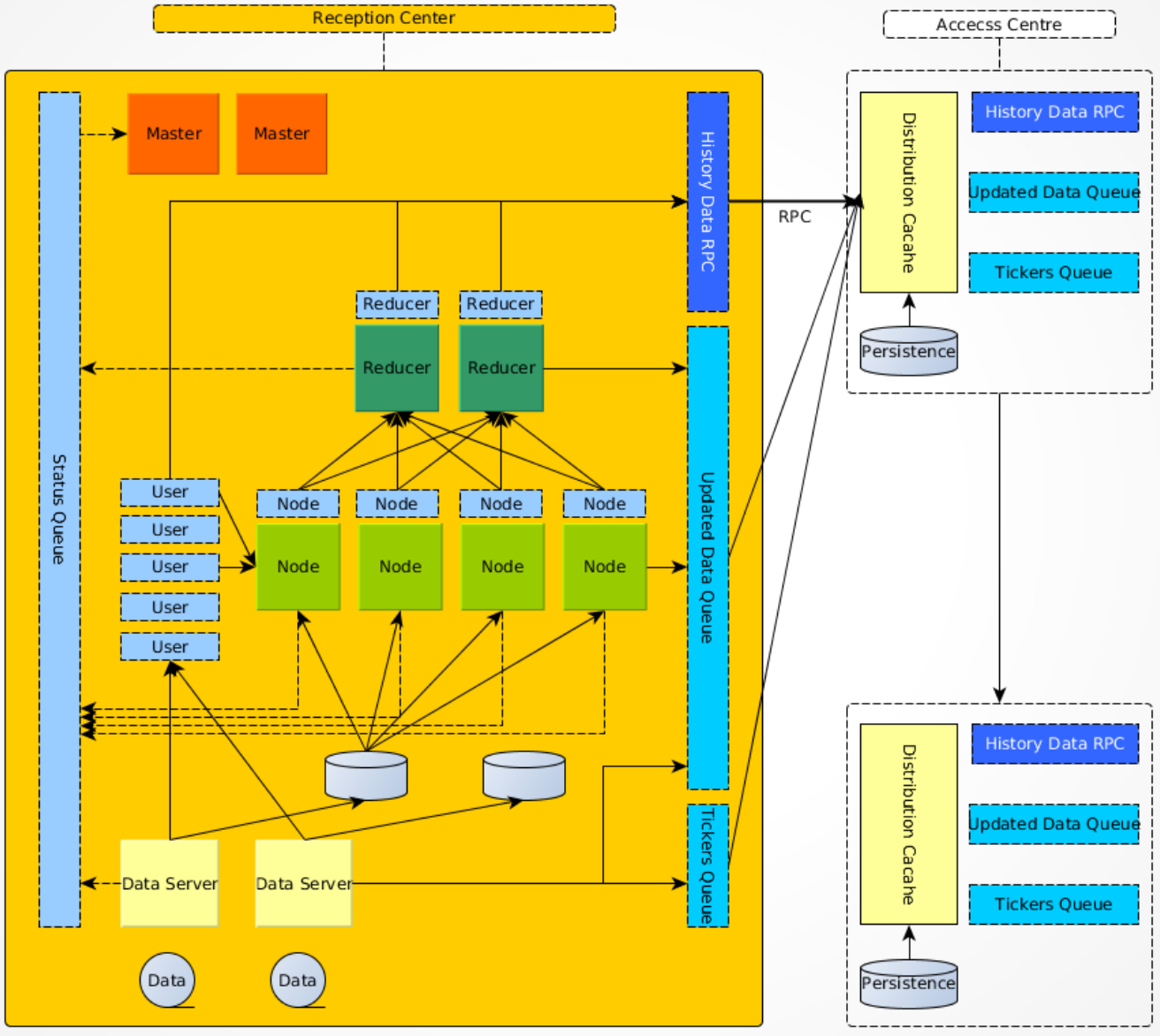
- 实现 - 2010到2012年
  - 数据In-Memory
    - 每只证券的每项行情和指标都各是一个Actor。上百万个计算Actors
  - 异步事件驱动
    - Ticker和QuoteUpdated事件流
  - 变化按传递路径扩散
    - 变化是事件Event：DailyQuoteUpdated，MinuteQuoteUpdated
    - 指标间的依赖关系是Flow(流图)
    - 指标订阅相关的变化Event
    - 包括UI常用的传统Listener模式，全部改用Actor异步实现，订阅变化Event
  - 增量计算，比如MA(High, 6)
    - 按优化算法只计算最新变化的影响
    - 所有引用该指标的其它指标均收到该变化，并做自己相应的增量计算
    - UI上的指标曲线（也是Actor）异步刷新
  - 事件流＋快照
    - 对事件流可以组合、过滤、查询
    - 对于同时依赖或组合了成千上万变化源的指标，比如聚合指标，难以对每个/每次变化都响应，用定时快照驱动更新

Reactive 分布式并行金融计算平台



处处是 Actor
处处是异步无阻塞
包括 UI 要素

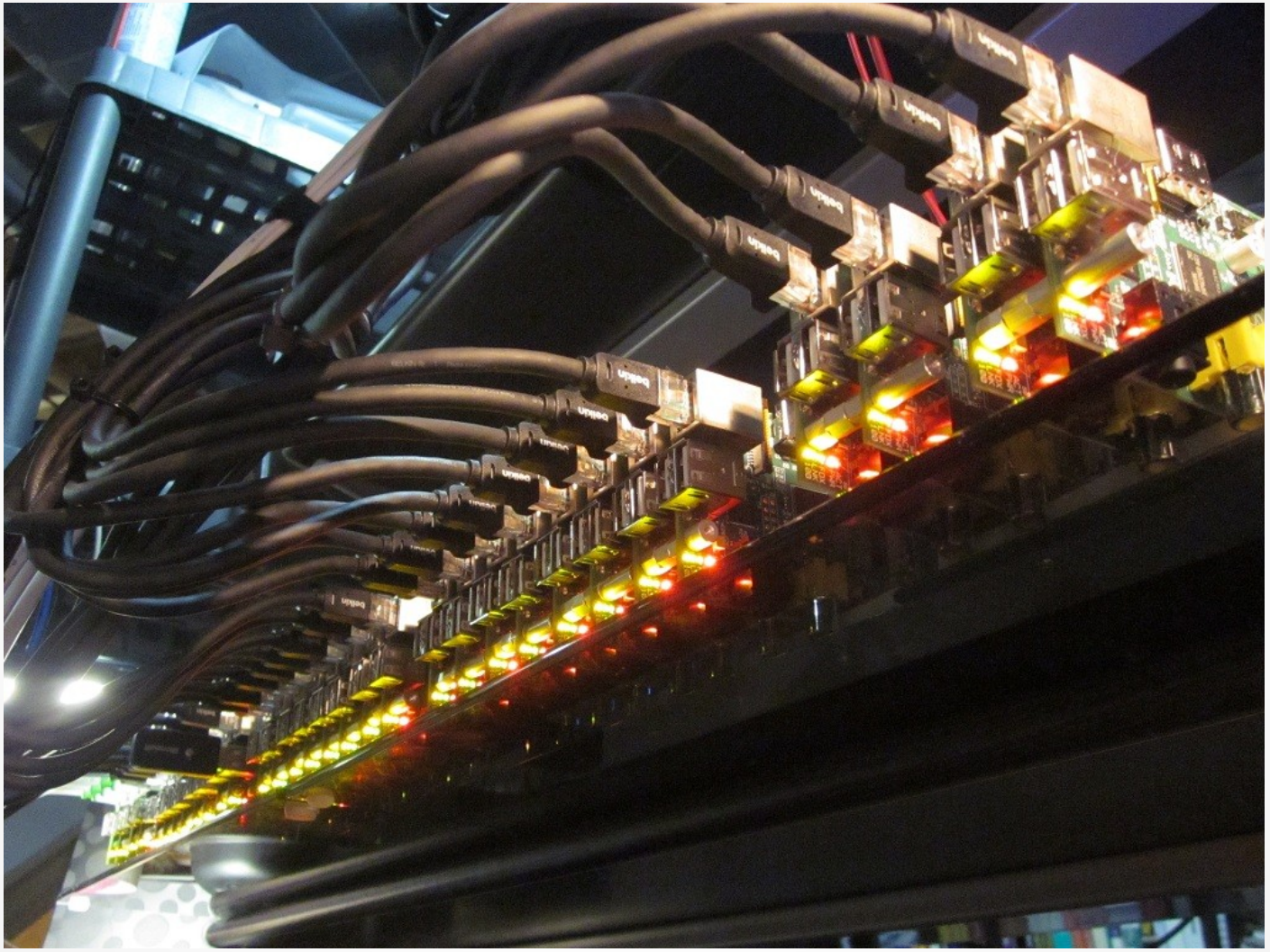
Reactive 分布式并行金融计算平台



Reactive 分布式并行金融计算平台

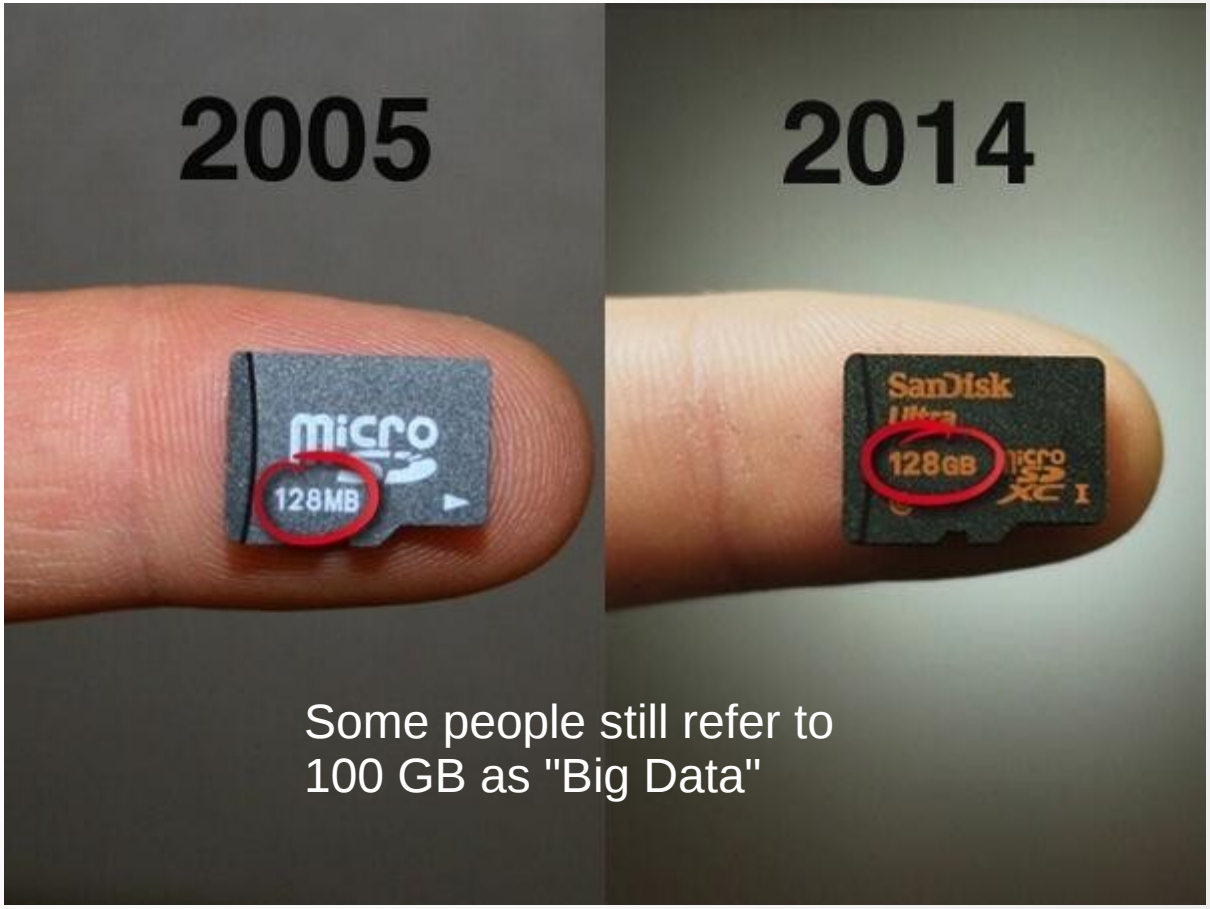
- 201? 年
  - Actor 是地址透明的，本身具备分布能力
  - N 只股票及其对应的 Actors 可以在一个节点
  - 也可以：股票 / 节点，比如一块 Raspberry Pi
  - 每只股票的数据持久化在本地，加载和计算在本地
  - 几千个 Raspberry Pi 的成本可以接受，而且
    - 新增一只证券 ＝ 加一个 Pi 节点 (200 多元)
    - 挂掉一个节点恢复快（一只证券）
    - 分布式下主要瓶颈“网络 IO” 和“存储 IO” 端口大大增加

Reactive 分布式并行金融计算平台



未来展望（5 年内）

- 750 亿长连终端
- 每时每刻产生巨量的消息
- 需要实时响应
- 我们准备好了吗？



Q & A

移动互联网络应该是

一个异步的由事件流驱动的巨大的状态机

谢谢!

https://github.com/wandoulabs/spray-websocket
https://github.com/wandoulabs/spray-socketio