通过 LoRe 与 PRDTs 实现应用特定的协调与收敛

Julian Haas · 2026-09-28 · 中文译文

演讲与问答全文

演讲与问答可读版逐字稿 · 中文译文。2026 年 9 月 29 日二次校订：对照 Soniox 转录、Zoom 字幕和演示材料，整理重复起句、断句及术语，保留发言顺序、例子、限定条件与讲者观点。中文逐段对照英文校订。时间从原录像 00:05:04 起计；省略等候及设备调试。仍无法确认的表述已标注；未逐句人工回听。

开场与研究背景



活动标题 · 00:00:06

00:00:00 · Julian Haas

那我再自我介绍一下。大家好，我是 Julian，在德国达姆施塔特工业大学攻读博士，研究本地优先软件的编程语言。今天我会简要介绍我的两篇研究论文：LoRe 和 PRDTs。它们都关注本地优先应用的数据一致性如何与应用特定的逻辑相互作用。这场演讲有点像“导演剪辑版”，因为我把两场演讲合到了一起。希望不会让大家困惑；最后应该还有提问时间，如果听完后有疑问，我很乐意回答。

00:01:09 · Julian Haas

首先介绍 LoRe，一种用于构建可验证地满足安全性质的本地优先软件的编程语言。我们三年前在 ECOOP 发表了这项工作，之后还有扩展的期刊版本。现在进入正题。

LoRe: A Programming Model for Verifiably Safe Local-First Software

Presented at ECOOP '23
Published in TOPLAS '24


LoRe: A Programming Model for Verifiably Safe Local-first Software

2

https://doi.org/10.1145/3633769

1 / 47

LoRe：可验证安全的本地优先编程模型 · 00:01:16

从云端到本地优先

00:01:37 · Julian Haas

这项工作的动机，是从集中式云端转向本地优先软件。我想这些对大家并不陌生，因为你们已经听过不少相关演讲。如今大多数应用的结构大致如此：应用实际运行在某个云端、某个数据中心里，你的设备主要连接那里，相当于访问服务器上应用的一扇窗口。众所周知，这有几个缺点。首先，离线时你基本没法使用这些应用；连接一断，你就无法再访问应用。

00:02:23 · Julian Haas

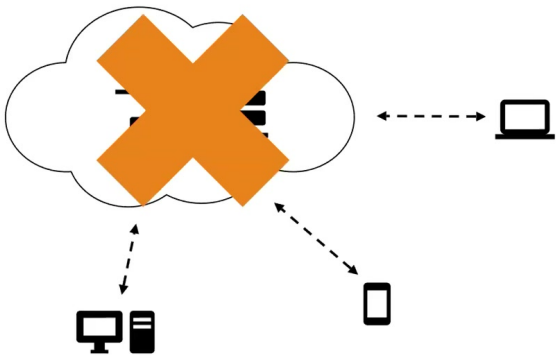
这也带来数据隐私和所有权问题。很多时候，你并不知道服务器究竟在哪里、属于谁、谁能访问你的数据，以及谁能阅读你发到服务器上的内容。这也是本地优先软件的一个动机。最后，我认为还有一个重要问题：用户的能动性。

00:02:53 · Julian Haas

我的意思是，云端应用没有永远存在的保证。我们已经见过很多公司关闭服务器的例子。如果你的所有数据都在那里，即使你想继续使用应用，也可能无能为力。在我看来，本地优先就是要改变这种结构：设备首先彼此通信，每个设备都有重要应用数据的本地副本。你仍然可以使用云端、向它发送数据，但它不应是应用不可或缺的部分；没有云端，应用也应该能继续运行。

Motivation: From Centralized Clouds to Local-First Software

The Problem With Modern Cloud Software



Problems:

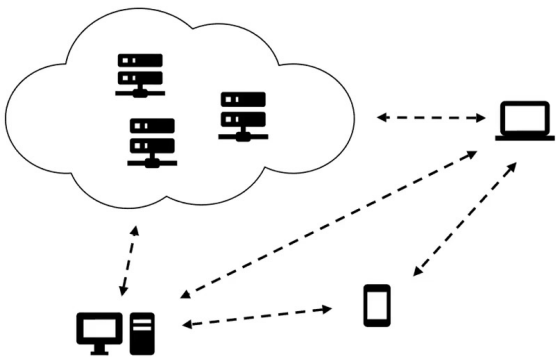
- Offline Availability
- Data Privacy & Ownership
- User Agency

3 / 47

现代云软件的问题 · 00:03:11

Motivation: From Centralized Clouds to Local-First Software

The Local-First Approach



M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, 'Local-first software: you own your data, in spite of the cloud', in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, in Onward! 2019, Oct. 2019, doi: 10.1145/3359591.3359737.

4 / 47

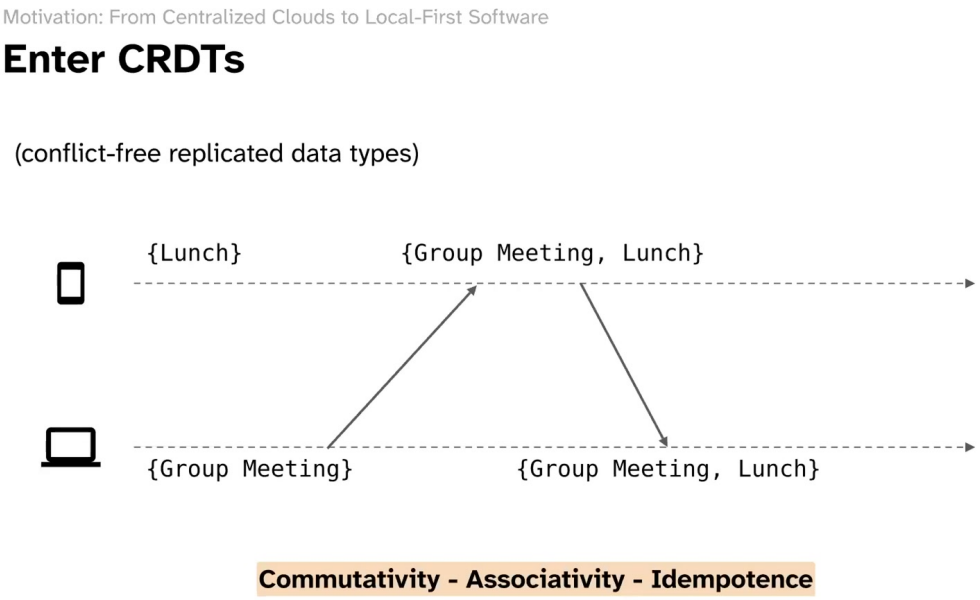
本地优先架构 · 00:03:45

00:03:45 · Julian Haas

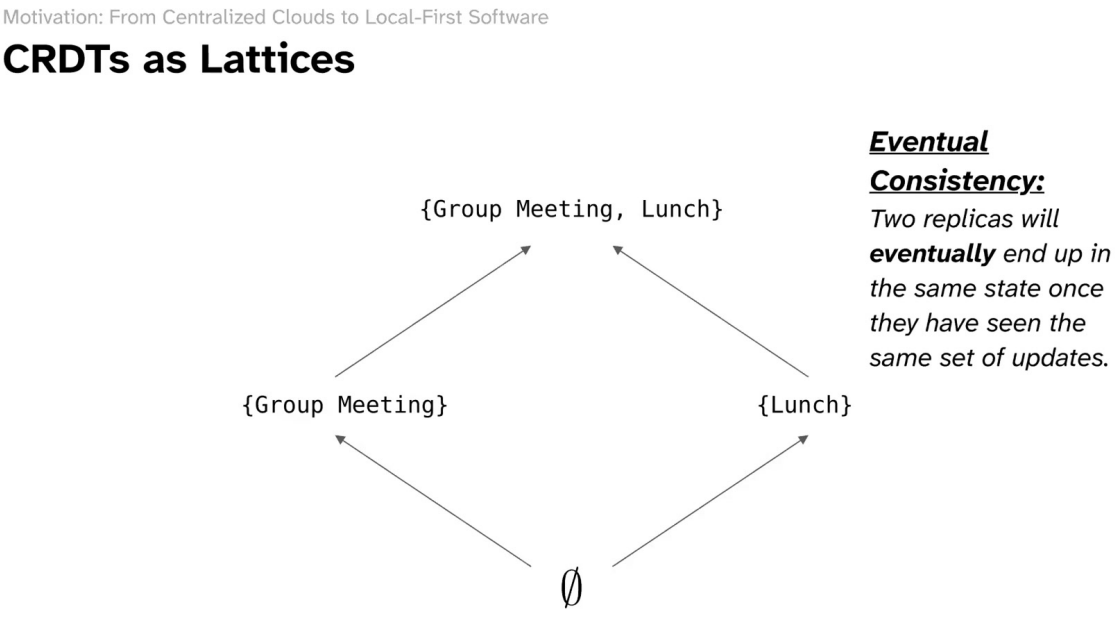
结束这段介绍前，再说一下如今最常见的实现方式：CRDT。CRDT 通过限制应用可执行的操作，并以特定方式组织数据，来避免合并冲突。以分布式日历应用为例：我在手机上添加一个午餐事件，之后在笔记本电脑上添加一个小组会议。设备重新连接后，便可以合并各自的状态，得到包含两个事件的日历。我们希望 CRDT 具备的关键性质是交换律、结合律和幂等性。

00:04:43 · Julian Haas

我不详细展开这些性质，换一种方式来看：系统的状态构成一个并半格（join-semilattice）。图中展示了应用数据可能处于的状态，任意两个状态都能合并。设备可以暂时分歧，但之后总能重新合并状态。这就是最终一致性：两个副本接收到同一组更新后，最终到达相同状态。但这也是今天的切入点：对某些应用，仅有这个保证还不够。假设用日历记录休假，公司规定员工最多只能休 30 天。



CRDT 的合并性质 · 00:04:46



CRDT 与格 · 00:05:17

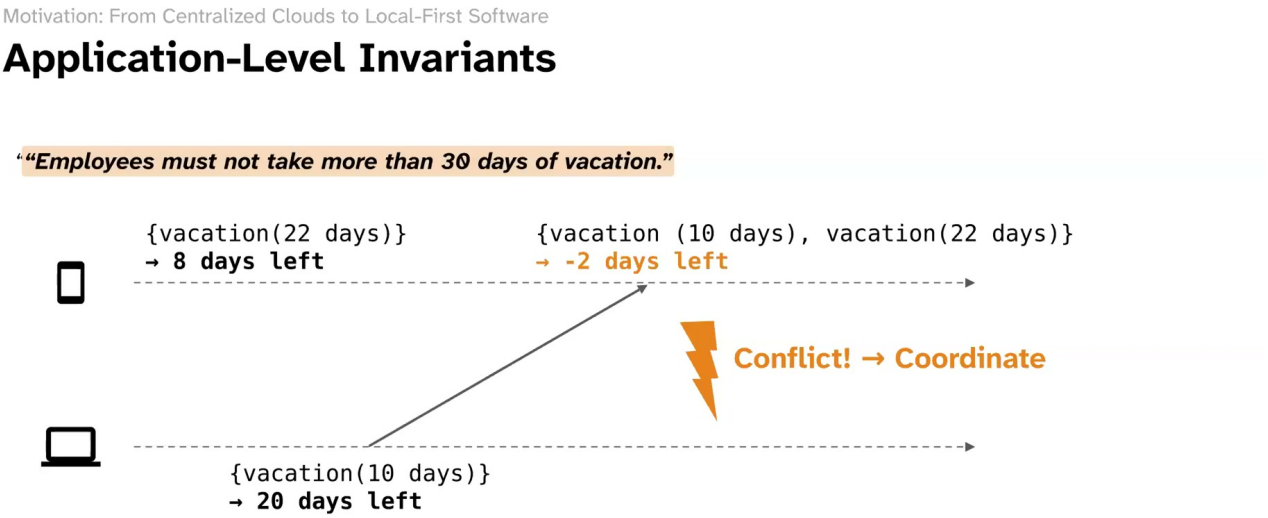
最终一致性的边界

00:05:45 · Julian Haas

我知道，这是一个很德国式的休假视角，但我们通常确实有 30 天。现在，我在手机上添加 22 天假期，手机认为还剩 8 天；随后在笔记本上添加另一个 10 天假期，笔记本认为还剩 20 天。你应该能看出问题：合并后，两个假期都在日历里，但我使用的假期总数已经超额。

00:06:19 · Julian Haas

这只是可能出现的冲突之一。只要应用里存在某种有限资源，就可能遇到类似问题。比如售票，或者寄送包裹时确保同一件东西不会寄出两次。这些地方就需要协调：设备必须协商由谁做什么，才能避免这类冲突。



应用层不变式与冲突 · 00:06:43

LoRe 的编程模型

00:06:52 · Julian Haas

LoRe 的想法是：能不能设计一种编程语言，让我们自动推理这些冲突？我们提出的答案是 LoRe，一种带有验证编译器的本地优先响应式编程语言。这些词听起来有点复杂，下面简要看看它如何工作。设想的架构是：用户提供输入程序，编译器自动分析它及其数据流，然后生成验证代码，交给自动程序验证器。在这项工作里，我们使用 Viper 验证工具包。

Motivation: From Centralized Clouds to Local-First Software

LoRe to the Rescue

High-level idea:
Can we design a programming model that **detects** and **fixes** these kinds of conflicts statically (and automatically)?

Approach:
LoRe: A **local-first reactive programming** language with a **verifying compiler**.

8 / 47

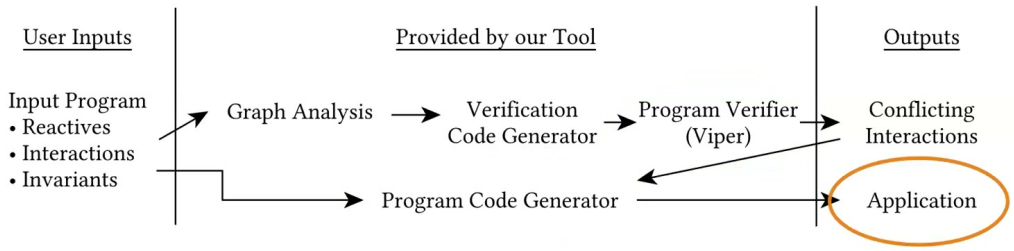
LoRe 的方法 · 00:07:13

00:07:51 · Julian Haas

Viper 的任务，是分析程序中可能发生哪些交互，以及这些交互在什么地方可能产生冲突。结果会反馈给编译器；理想情况下，编译器可以在生成的应用中修复这些冲突。

LoRe in a Nutshell

Architecture Overview



10 / 47

LoRe 架构 · 00:08:08

00:08:11 · Julian Haas

如何把它放进编程语言？LoRe 的模型结合了三个基本组成部分：响应式状态（reactives）、交互（interactions）和不变式（invariants）。我用一个小代码示例说明，希望不会太难理解。这是 Scala 语法；我们把 LoRe 实现为 Scala 中的领域特定语言。例子还是刚才的日历应用。首先定义两个响应式状态，也就是工作日历和休假日历。在 LoRe 中，响应式状态基本上就是 CRDT，数据存放在这里。此处二者都是添加优先集合（add-wins set）。



语言示例 · 00:08:41

00:09:07 · Julian Haas

添加进去的事件会合并到集合中。由于这是响应式编程语言，这些状态还可以连接起来，右侧的数据流图展示了这一点。除源响应式状态外，我们还能定义派生响应式状态：根据输入计算出的应用状态。例如，all_appointments 合并两个日历的所有日程，而 remaining_vacation 根据休假日历持续计算还剩多少天假期。

LoRe in a Nutshell

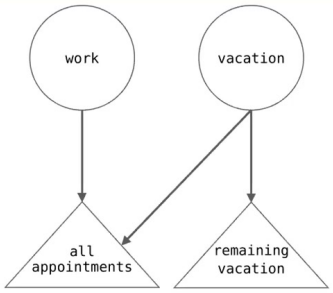
Syntax: Reactives

```
val work: Source[Calendar]     = Source(AWSet())
val vacation: Source[Calendar] = Source(AWSet())

val all_appointments: Derived[Set[Appointment]] = Derived {
 work.value.union(vacation.value)
}
val remaining_vacation: Derived[Int] = Derived {
 30 - sumDays(vacation.value)
}
val add_appointment = Interaction[Calendar, Appointment]
   .requires { (_, a) => a.start < a.end }
   .requires { (cal: Calendar, a: Appointment) => !cal.contains(a) }
   .executes { (cal: Calendar, a: Appointment) => cal + a }
   .ensures { (cal: Calendar, a: Appointment) => cal.contains(a) }
val add_vacation = add_appointment.modifies(vacation)
 .requires { (cal, a) => remaining_vacation.value - a.days >= 0 }

val i1 = Invariant {
 all_appointments.value.forall { a => a.start < a.end }
}
val i2 = Invariant {
 remaining_vacation.value >= 0
}
```

1. **Reactives**
2. Interactions
3. Invariants



13 / 47

响应式状态与数据流 · 00:09:37

00:09:58 · Julian Haas

接下来是交互，它们用来更新响应式状态。这里定义了 add_appointment 交互，向日历添加日程。requires、executes 和 ensures 子句向验证器描述这个交互的行为。每个交互都可以有前置条件和后置条件，供验证使用。add_vacation 示例还使用 modifies 子句，限定它作用于哪个响应式状态：只向休假日历添加假期，并增加一个仅适用于休假的前置条件。

LoRe in a Nutshell

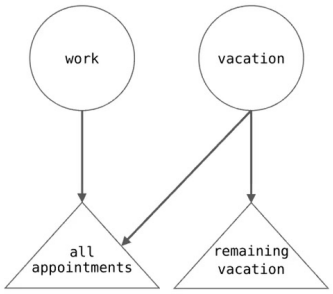
Syntax: Interactions

```
val work: Source[Calendar]     = Source(AWSet())
val vacation: Source[Calendar] = Source(AWSet())

val all_appointments: Derived[Set[Appointment]] = Derived {
 work.value.union(vacation.value)
}
val remaining_vacation: Derived[Int] = Derived {
 30 - sumDays(vacation.value)
}
val add_appointment = Interaction[Calendar, Appointment]
   .requires { (_, a) => a.start < a.end }
   .requires { (cal: Calendar, a: Appointment) => !cal.contains(a) }
   .executes { (cal: Calendar, a: Appointment) => cal + a }
   .ensures { (cal: Calendar, a: Appointment) => cal.contains(a) }
val add_vacation = add_appointment.modifies(vacation)
 .requires { (cal, a) => remaining_vacation.value - a.days >= 0 }

val i1 = Invariant {
 all_appointments.value.forall { a => a.start < a.end }
}
val i2 = Invariant {
 remaining_vacation.value >= 0
}
```

1. Reactives
2. **Interactions**
3. Invariants



14 / 47

交互及前后置条件 · 00:10:39

00:10:52 · Julian Haas

也许这已经有点细了。第三部分可能最重要：不变式。它描述应用应该满足的行为，也就是我们希望得到哪些保证。这里定义了两个不变式：所有日程的开始时间必须早于结束时间；剩余假期不能低于零。也就是说，剩余假期应保持非负。现在看起来可能有道理，但究竟怎么验证？怎么知道哪些部分会出问题、哪些交互可能产生冲突？

LoRe in a Nutshell

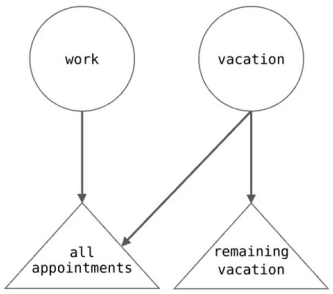
Syntax: Invariants

```
val work: Source[Calendar]     = Source(AWSet())
val vacation: Source[Calendar] = Source(AWSet())

val all_appointments: Derived[Set[Appointment]] = Derived {
 work.value.union(vacation.value)
}
val remaining_vacation: Derived[Int] = Derived {
 30 - sumDays(vacation.value)
}
val add_appointment = Interaction[Calendar, Appointment]
   .requires { (_, a) => a.start < a.end }
   .requires { (cal: Calendar, a: Appointment) => !cal.contains(a) }
   .executes { (cal: Calendar, a: Appointment) => cal + a }
   .ensures { (cal: Calendar, a: Appointment) => cal.contains(a) }
val add_vacation = add_appointment.modifies(vacation)
 .requires { (cal, a) => remaining_vacation.value - a.days >= 0 }

val i1 = Invariant {
 all_appointments.value.forall { a => a.start < a.end }
}
val i2 = Invariant {
 remaining_vacation.value >= 0
}
```

1. Reactives
2. Interactions
3. **Invariants**



15 / 47

应用不变式 · 00:11:17

不变式保持与合流性

00:11:46 · Julian Haas

这就是接下来要研究的。我们问的是：给定的交互会不会违反给定的不变式？右侧的响应式数据流可以帮助我们。假如要验证第二个不变式，也就是剩余假期，那么查看数据流图就能发现：它唯一的变化来源，是上方的休假状态。完整程序可能复杂得多，但我们只需验证下面这个交互，因为它会改变休假状态，而这又是影响剩余假期的唯一变化来源。对于这些交互，我们需要验证两件事。

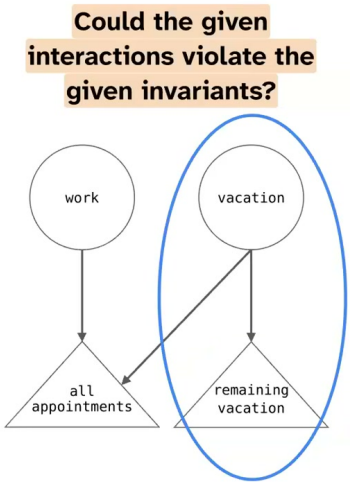
Semantics and Verification

High-Level Idea

```
val work: Source[Calendar]     = Source(AWSet())
val vacation: Source[Calendar] = Source(AWSet())

val all_appointments: Derived[Set[Appointment]] = Derived {
 work.value.union(vacation.value)
}
val remaining_vacation: Derived[Int] = Derived {
 30 - sumDays(vacation.value)
}
val add_appointment = Interaction[Calendar, Appointment]
   .requires { (_, a) => a.start < a.end }
   .requires { (cal: Calendar, a: Appointment) => !cal.contains(a) }
   .executes { (cal: Calendar, a: Appointment) => cal + a }
   .ensures { (cal: Calendar, a: Appointment) => cal.contains(a) }
val add_vacation = add_appointment.modifies(vacation)
 .requires { (cal, a) => remaining_vacation.value - a.days >= 0 }

val i1 = Invariant {
 all_appointments.value.forall { a => a.start < a.end }
}
val i2 = Invariant {
 remaining_vacation.value >= 0
}
```



18 / 47

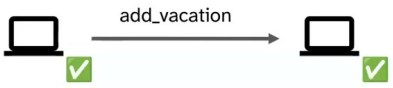
依据数据流缩小验证范围 · 00:12:37

00:12:47 · Julian Haas

我们把它们定义为两个验证条件：不变式保持性和合流性（confluence）。不变式保持性比较直观：如果不变式在某个状态下成立，设备执行一次交互后，它仍应成立。但其他设备同时执行操作时，会不会出问题？这正是第二个条件——合流性——要处理的。我们希望两个并发交互无论在什么时刻与合并操作交错执行，都得到相同结果。

Semantics and Verification

Invariant-Preservation



20 / 47

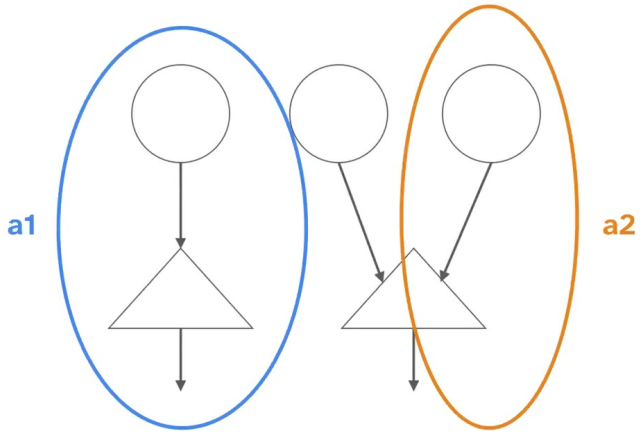
不变式保持 · 00:13:20

00:13:49 · Julian Haas

也就是说，两个设备先添加事件再合并、先合并再添加，或者在中途合并，都不应该影响结果。数据流图也让这个检查更容易：如果两个交互 A1 和 A2 影响的是图中完全分离的部分，我们就知道它们是合流的，彼此不会干扰。但我们已经看到不合流的例子：合并时机确实会影响结果。如果那两个设备更早合并，就会知道不能加入所有这些假期。怎么办？我们强制要求这些交互不能并发发生，而必须协调。

Semantics and Verification

Invariant-Confluence

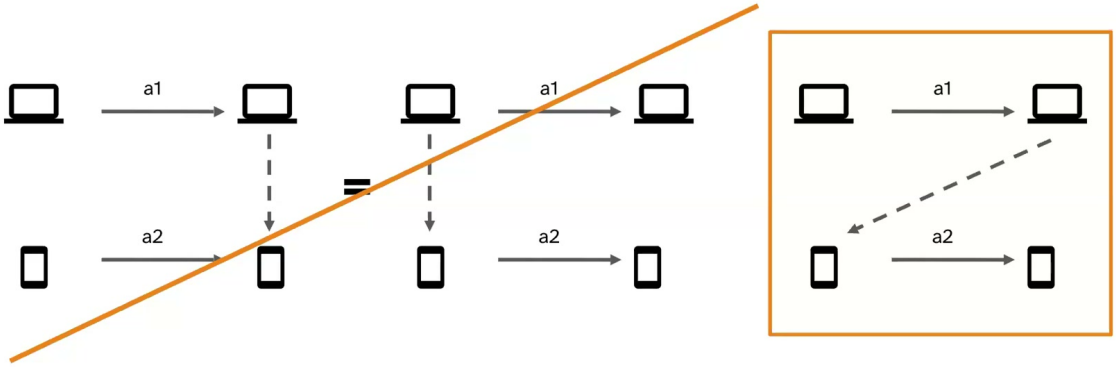


22 / 47

不变式合流性 · 00:14:12

Semantics and Verification

Non-Confluence



24 / 47

非合流操作需要协调 · 00:14:46

00:14:49 · Julian Haas

这里，笔记本必须与手机协调，决定谁先执行交互。总结来说，每个设备在本地检查不变式保持，确保自己要做的操作符合不变式；对于不合流的操作，设备还需要彼此协调，确保不会并发执行。

混合一致性与协调

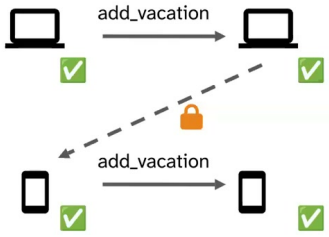
00:15:24 · Julian Haas

退一步看，这是一种混合一致性模型。大多数交互只更新 CRDT，无需协调，提供最终一致性；另一些交互则需要协调。这自然带来问题：如何把协调放进本地优先应用，确保两个设备真正达成一致？很多论文和系统都研究过混合一致性编程。但遗憾的是，最常见的协调办法，仍然是使用集中式服务器或数据库，于是就出现了图中这种“美好”的情况。这大概不是本地优先应用想要的，因为为了协调，我们又把服务器塞了回来。接下来进入演讲的第二部分：能否不依赖中央服务器，为本地优先应用提供协调？这正是开头提到的 PRDTs 所研究的问题：仅通过复制数据类型，如何把协调加入本地优先应用？

Semantics and Verification

Verification Summary

Safety through coordination + invariant preservation

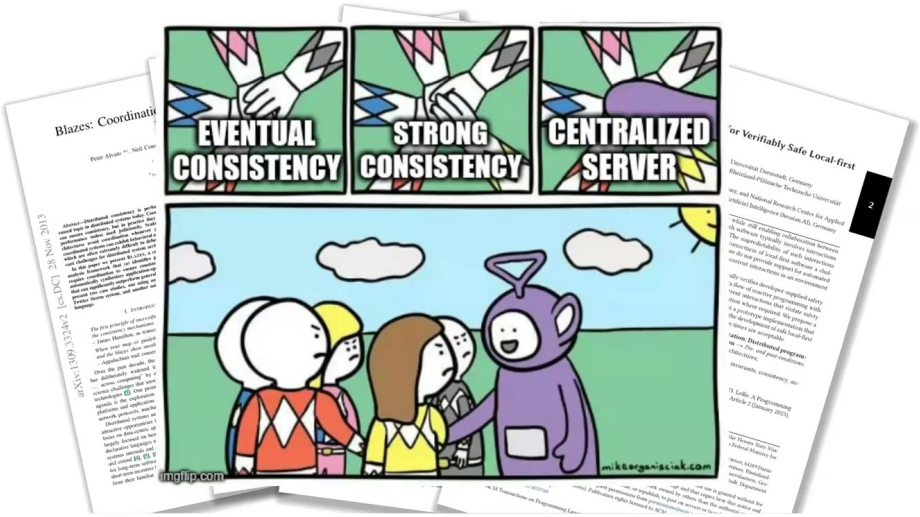


25 / 47

LoRe 验证小结 · 00:15:26

Semantics and Verification

The Problem With Mixed-Consistency



27 / 47

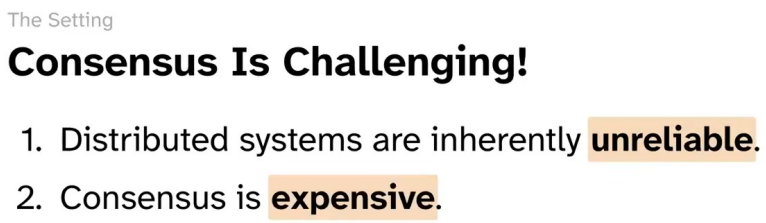
混合一致性的协调问题 · 00:16:33

00:17:02 · Julian Haas

先简要说明场景。我们想实现的本质上是共识协议：多个设备对某个值、某个结果达成一致。前面的例子是决定谁先行动。再简化一点，假设三个设备在猫和狗之间做选择，两个投猫、一个投狗，那么结果应该是猫。

00:17:38 · Julian Haas

问题看起来简单，实践中却很难做对。主要有两个原因：首先，分布式系统不可靠，投票消息可能丢失；其次，共识代价很高。如果要正确完成协议，让设备确认彼此已经看到投票、知道决定，就需要大家在线并交换大量消息。因此，不同场景有很多不同的共识算法。设备是在数据中心、分布于不同大陆，还是直接通信的移动设备，都会显著影响哪种协议更适合。



31 / 47

共识的困难 • 00:18:11

用 PRDTs 构建共识

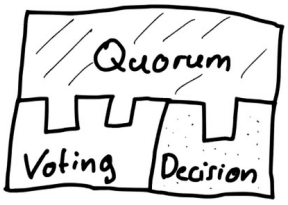
00:18:38 • Julian Haas

PRDTs 在这里发挥作用。PRDTs 是 Protocol Replicated Data Types，即协议复制数据类型：一种使用复制数据类型，以组合方式实现共识协议的编程模型。你可以堆叠、组合不同的基本组件，构建新协议；我们还希望提供自动验证框架，检查组合之后是否仍然是安全的共识协议。怎么做？这项工作的一条灵感线索来自 1990 年 Halpern 和 Moses 的论文：可以把分布式系统中的一致同意或共识，视为一个知识问题。

The Setting

Our Approach

Protocol Replicated Data Types (PRDTs): a programming model for the **compositional** implementation of **consensus protocols** using **replicated data types**.





32 / 47

PRDT 的方法 · 00:19:11

00:19:36 · Julian Haas

基本想法是，把只有部分设备知道的分布式知识，提升为共同知识（common knowledge）。这里我把共同知识描述为：我们知道每个设备最终都会知道的东西。起初，每个设备只知道自己的投票，是猫还是狗；交换足够信息后，就知道结果应该是什么。关键在于，这是一个单调过程。像 CRDT 一样，设备持续交换消息、交换知识片段，直到能够安全地得出结论。PRDTs 就用复制数据类型储存这些知识：不断加入知识片段，累积到足以做出决定。

Core Idea

Agreement Is a Question of Knowledge

Halpern & Moses, 1990:

distributed knowledge
known by some devices

→

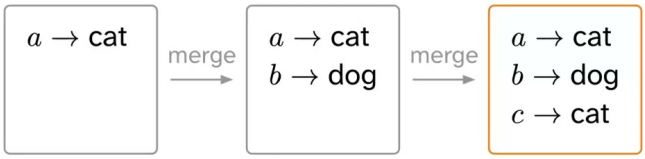
common knowledge
eventually known by every device

34 / 47

共识与知识 · 00:19:52

Core Idea

Knowledge Grows Monotonically



35 / 47

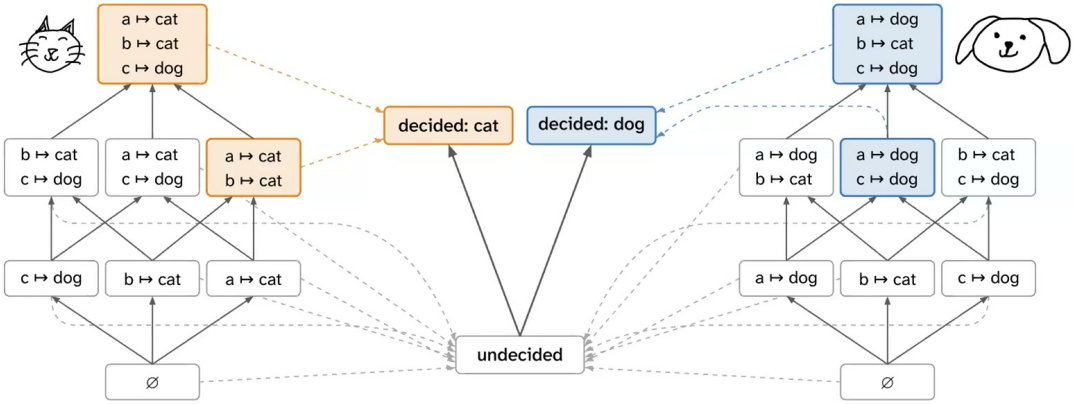
知识单调增长 · 00:20:31

00:20:40 · Julian Haas

看这张图，可以理解这个过程。左右两边展示了知识在两种情况下如何演变，中间展示了我们能得出的结论：左边最后是猫获胜，右边最后是狗获胜。正确的共识协议，也就是正确的 PRDT，必须保证最终只会进入左边或者右边，而不能混在中间。所有设备必须处于同一个“左侧世界”或“右侧世界”，不能混用。

Core Idea

Knowledge Grows Monotonically



36 / 47

知识状态与决策 · 00:21:06

知识格、动作与决策

00:21:24 · Julian Haas

为此，我们再次提出一个包含三个部分的编程模型，稍后会展示具体语法。第一部分是知识格：就是前面看到的格结构，用一种数据类型保存系统知识，并且单调增长。第二部分是带前置条件的动作，用来限制向知识格中添加什么知识。通过这些条件，可以约束知识的增加，保证协议安全。例如，每个设备只能投一次票。

00:22:22 · Julian Haas

第三部分是决策函数，也就是前面图中的箭头。它根据知识状态做出不可撤回的决定。“不可撤回”是关键：一旦我们把某个结果定义为决定，就必须保证它不会被撤销。接下来又是 Scala 代码，实现的是刚才那个简单投票协议：每个设备只能投一次票，得票最多的值获胜。

The Programming Model

The Three PRDT Components

1. **Knowledge lattice**
   structures monotonically growing protocol knowledge
2. **Protocol actions** with preconditions
   define how new knowledge is produced
3. **Decision function**
   makes irrevocable decisions based on the knowledge state

38 / 47

PRDT 的三个组成部分 · 00:22:42

00:23:06 · Julian Haas

先看知识格。这里唯一需要保存的相关知识，是每个参与者投了什么。复合数据类型只有一个 votes 字段，它是一个映射：从投票者的 ID 映射到其选择的值。第二部分是协议动作，规定如何向知识状态添加知识。这里的 voteFor 动作允许为某个值投票：返回一个映射，把本地副本 ID 映射到我投票选择的值。

The Programming Model

Voting as a PRDT

```
val participants: Set[Uid] = ...       // fixed for all processes

case class Voting[A](votes: Map[Uid, A]) {       // knowledge state

  def hasNotVoted(using LocalUid): Boolean =     // precondition
    !votes.contains(replicaId)

  def voteFor(value: A)(using LocalUid): Voting[A] =   // protocol action
    precondition(hasNotVoted) {
      Voting(Map(replicaId -> value))
    }

  def decision: Decision[A] = getLeadingValue() match  // decision function
    case Some(value, count) if count > participants.size / 2 => Decided(value)
    case _ => Undecided
}
```

39 / 47

投票协议的知识格 · 00:23:10

The Programming Model

Voting as a PRDT

```
val participants: Set[Uid] = ...       // fixed for all processes

case class Voting[A](votes: Map[Uid, A]) {       // knowledge state

  def hasNotVoted(using LocalUid): Boolean =     // precondition
    !votes.contains(replicaId)

  def voteFor(value: A)(using LocalUid): Voting[A] =   // protocol action
    precondition(hasNotVoted) {
      Voting(Map(replicaId -> value))
    }

  def decision: Decision[A] = getLeadingValue() match  // decision function
    case Some(value, count) if count > participants.size / 2 => Decided(value)
    case _ => Undecided
}
```

39 / 47

投票动作 · 00:23:51

00:23:56 · Julian Haas

这个动作有一个前置条件 hasNotVoted，即“尚未投票”。我们检查映射里是否已有本地副本 ID 的投票；如果有，就不能再投。这保证每个设备只投一次。最后是决策函数：统计票数，检查是否超过半数参与者选择同一个值。如果是，就返回决定；否则返回 Undecided，即尚未决定。

The Programming Model

Voting as a PRDT

```
val participants: Set[Uid] = ...       // fixed for all processes

case class Voting[A](votes: Map[Uid, A]) {       // knowledge state

  def hasNotVoted(using LocalUid): Boolean =     // precondition
    !votes.contains(replicaId)

  def voteFor(value: A)(using LocalUid): Voting[A] =   // protocol action
    precondition(hasNotVoted) {
      Voting(Map(replicaId -> value))
    }

  def decision: Decision[A] = getLeadingValue() match  // decision function
    case Some(value, count) if count > participants.size / 2 => Decided(value)
    case _ => Undecided
}
```

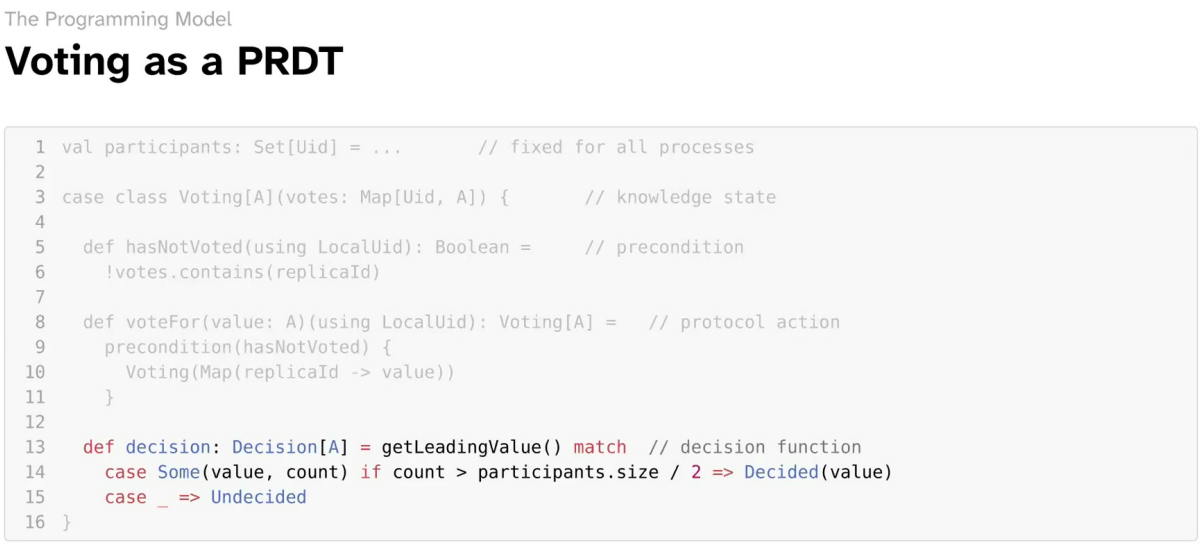
39 / 47

投票前置条件 · 00:24:07

组合协议

00:24:44 • Julian Haas

希望这样比较清楚，如果有疑问，之后还可以回来讨论。PRDTs 的一个关键优势是可组合性。它们本质上是 Scala 的代数数据类型，因此也能按照代数数据类型的方式组合。比如，定义一个让两次投票并行发生的协议，只需要组合两个前面介绍的 Voting 实例。这里不增加新的协议动作，仍然可以分别访问组件、分别投票。但我们定义新的决策函数：只有两个组件都已经做出决定，ParallelVoting 才返回决定。



决策函数 • 00:24:45

The Programming Model

PRDT Composition

```scala
case class ParallelVoting[A, B](voting1: Voting[A], voting2: Voting[B]) {

  def decision: Decision[(A, B)] =
    (voting1.decision, voting2.decision) match
      case (Decided(a), Decided(b)) => Decided((a, b))
      case _                        => Undecided
}
```



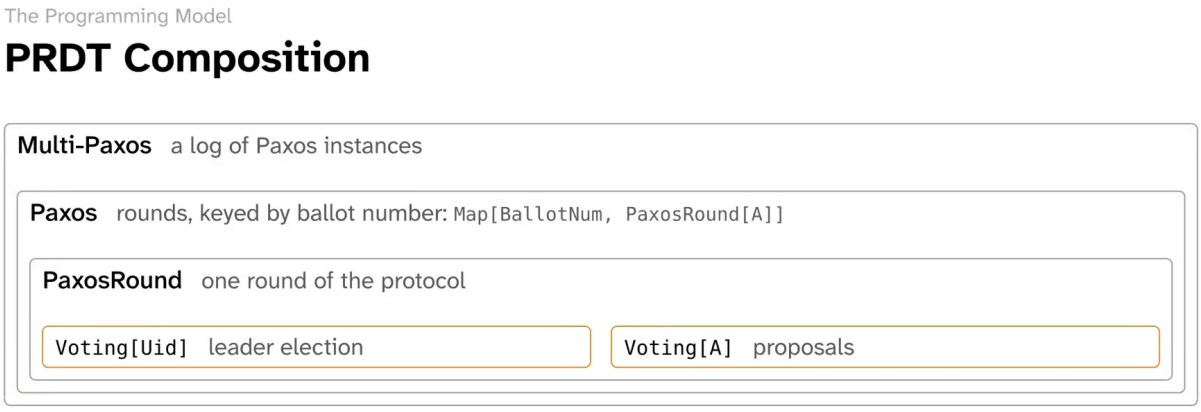
并行投票的组合 · 00:25:41

00:25:46 · Julian Haas

否则，它仍处于未决定状态。关键是，我们完全可以采用其他定义，取决于应用需要怎样的设计。这里等到二者都决定后才返回；也可以更早返回部分决定，例如第一项投票已有结果，而第二项还不知道。本例选择了前一种方式。

00:26:24 · Julian Haas

论文中还有更复杂的组合例子：用多个 Paxos 实例组合出 Multi-Paxos；每个 Paxos 实例又由多个 Paxos 轮次组成，内部还使用多次投票。这里没有足够时间，也没有足够幻灯片空间展示全部代码，但这种组合方式可以扩展到更复杂的情况。感兴趣的话，可以阅读论文。



组合出更复杂的协议 · 00:26:43

验证协议安全性

00:26:54 · Julian Haas

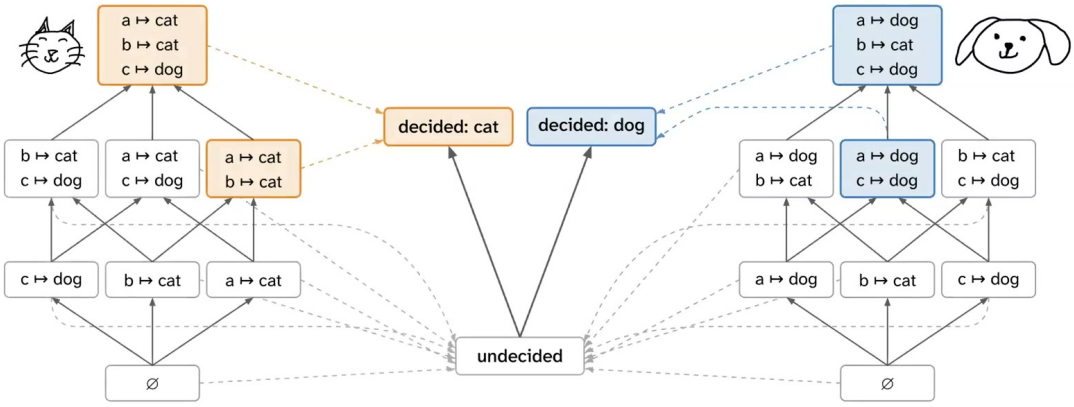
到目前为止，我们讨论的是 PRDTs 的设计部分：从较小的组件组装协议，方便地创建已有实现的变体。但和 LoRe 一样，我也强调过，共识算法设计中最困难的是正确性，我们希望自动验证它。怎么知道组合出来的协议确实正确？

00:27:25 · Julian Haas

回到前面的图，就能得到一些直觉。在中间的决策序里，决定只能向前发展：可以从“未决定”到“决定为猫”，但不能倒退。一旦进入左边的“猫王国”或右边的“狗领域”，就不能反悔。因此，不仅要保证决定单调推进，还要保证所有设备朝着相同方向推进。

Verification

When Is a PRDT Safe?



PRDT 的安全性 • 00:28:01

00:28:09 • Julian Haas

PRDTs 能把这个全局条件归约为局部验证条件。图中本来需要考虑网络中每个副本的状态，但我们把它归约成两个条件：动作单调性（action monotonicity）和稳定前置条件（stable preconditions）。动作单调性要求副本的本地动作只能让决定向前发展，不能倒退。稳定前置条件则要求：一个在本地已经可以执行的动作，不能因为其他设备的操作而变得不可执行。这与 LoRe 的思路很相似。也就是说，决定动作是否合法的前置条件，不能被其他设备的并发操作破坏。这两个条件相对容易检查，也可以使用自动工具。再看投票例子。首先验证动作单调性：这里只有一个协议动作，我们要确保它只能让决定增长。voteFor 做的是向本地映射添加一张票。

Verification

Two Local Verification Conditions

1. Action monotonicity:

An action enabled on a replica can only increase the decision.

2. Stable preconditions: Enabled preconditions stay enabled in the presence of concurrent actions.

44 / 47

两个局部验证条件 · 00:28:56

00:29:57 · Julian Haas

再看决策函数：只有超过半数参与者支持同一个值时，它才返回决定。假设已经达到这个多数条件，再增加一张票也不会推翻原来的多数，因为参与者集合是固定的。如果尚未决定，增加投票也只可能让决定向前发展。第二个条件是前置条件稳定。hasNotVoted 检查本地 ID 还没有投票；只要其他设备不能以这个 ID 添加投票，这个条件就是稳定的。

Verification

Verifying Voting

```
case class Voting[A](votes: Map[Uid, A]) {       // knowledge state
  def hasNotVoted(using LocalUid): Boolean =     // precondition
    !votes.contains(replicaId)

  def voteFor(value: A)(using LocalUid): Voting[A] =   // protocol action
    precondition(hasNotVoted) {
      Voting(Map(replicaId -> value))
    }

  def decision: Decision[A] = getLeadingValue() match  // decision function
    case Some(value, count) if count > participants.size / 2 => Decided(value)
    case _ => Undecided
}
```

1. Action monotonicity:

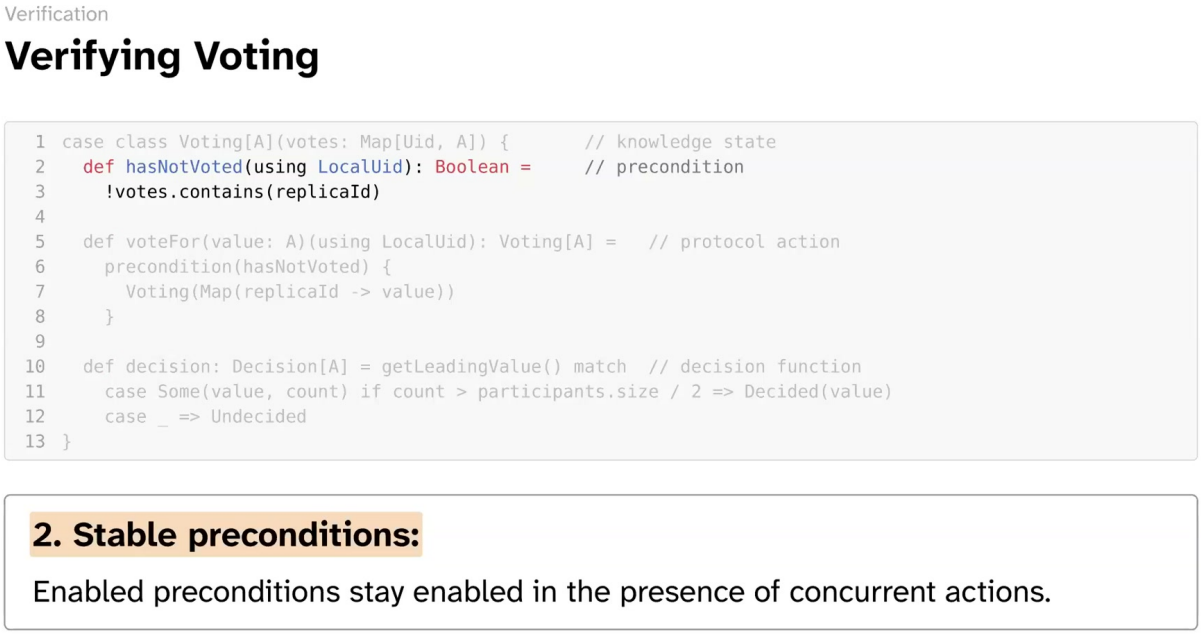
An action enabled on a replica can only increase the decision.

45 / 47

验证动作单调性 · 00:30:14

00:30:54 · Julian Haas

再强调一下假设：副本集合固定；每个副本有固定 ID，而且只有它自己能够使用该 ID 添加投票。



验证稳定前置条件 · 00:31:06

演讲小结

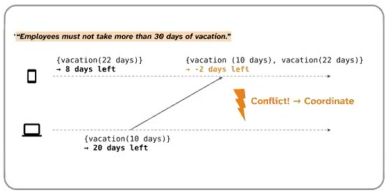
00:31:10 · Julian Haas

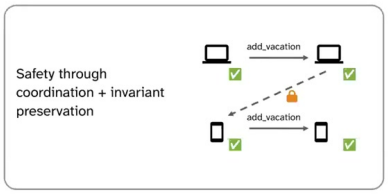
内容不少，我快速回顾一下，把时间留给问答。第一部分讨论了本地优先应用有时会遇到最终一致性的边界，比如日历例子。接着介绍 LoRe 和验证框架，它能分析应用哪些地方确实需要协调。但协调成本很高，不同场景需要不同的方案和算法。PRDTs 则允许我们在复制数据类型之上实现协调，不依赖中央服务器，因而容易集成到已有的本地优先应用中。通过堆叠和嵌套，还可以实现新的协议变体。

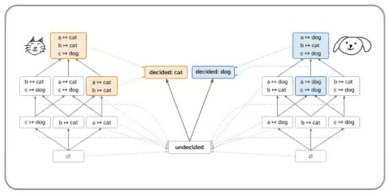
00:32:14 · Julian Haas

以上就是我的演讲。这里是两篇论文的二维码，感兴趣的话欢迎阅读，里面还有更多内容。我很乐意回答大家的问题。

Summary







julian-haas.de
types.pl/@haaase

LoRe: doi.org/10.1145/3633769

PRDTs: doi.org/10.48550/arXiv.2504.05173

47 / 47

演讲小结与论文链接 · 00:32:36

现场交流与问答

00:32:57 · Julian Haas

聊天里有一条评论：“这项工作很优雅，把共识表达为复制知识状态上的单调决策，是个很好的想法。”谢谢。可惜我读不懂第二条评论，如果有人能翻译一下就太好了。我不确定它是不是问题，也可能是第一条评论的翻译。

00:33:33 · 观众

那条评论只是说，如果有人需要帮助来提问，可以提出来，我们也会帮忙翻译。

研究思路如何形成

00:34:16 · 观众

我有一个问题。

00:34:19 · Julian Haas

当然，请讲。

00:34:21 · 观众

我更想了解你的研究经历：你是怎么遇到这些想法的，又是怎么开始把一致性看成共同知识的问题？我很想知道你是怎样走到这一步的。

00:34:49 · Julian Haas

这是个好问题。我觉得它更像一个迭代过程。我们在达姆施塔特的团队开发了一个用于代数复制数据类型的 Scala 库，其中已经有组合能力，可以把组件堆叠起来构造新组件。后来我们开始想：如果整个程序都这么构建呢？把应用逻辑也放进这种组件组合模型，是不是也能实现 Raft 或 Paxos？我看不出有什么理由不行，于是我们就开始尝试。说实话，关于共同知识的洞见是在这之后才出现的。我们当时正在实验，有人推荐了一篇 90 年代的论文。读完后我们发现："这正是我们在做的事情，也解释了为什么它可行。"这很有意思。但我并不是先找到那篇论文、再产生想法；更像是先不断尝试，看看什么能行。

00:36:17 · Julian Haas

然后有人指出："你们做的事情，30 年前已经有人研究过了。"当然，我指的是理论上，当时并没有对应的语言或实现。那篇论文读起来很有趣，我很推荐，甚至有些哲学意味，因为它讨论的是：根据你对"别人知道什么"的了解，你自己能够知道什么？这样说起来可能有点绕。

00:36:53 · 观众

谢谢。

00:36:55 · Julian Haas

不客气。

区块链能否作为协调器

00:36:58 · 观众

你觉得我们能否把区块链作为协调器？它是一个公开的共同知识数据库，所有人都能以去中心化的方式访问。当然，访问区块链仍然需要互联网。

00:37:18 · Julian Haas

谢谢提问。当然可以，但要看具体应用是否适合。区块链也使用共识算法，对吧？我觉得，当你需要开放网络时，通常会考虑区块链。这里 PRDTs 假设的是固定且已知的副本集合。可以改变集合、添加设备，但不能让成员随意流动；系统需要决定添加谁、移除谁。区块链使用基于工作量证明或权益证明等不同的共识算法，允许成员随时加入或退出。如果应用需要这种特性，可以使用区块链。

00:38:18 · Julian Haas

不过我觉得，对很多应用来说，这可能有些过度；正如你提到的，也会再次遇到集中化问题。当然，有些应用可能适合：如果你没有可信的服务器，但希望有一个在线、人人都能查看的东西，区块链可能是可行的办法。

成员变更与权限撤销

00:38:52 · 观众

谢谢。还有一个问题：我觉得在异步环境里，最难处理的是成员变更，尤其是权限撤销。PRDT 的想法很好，但我能想象成员动态变化不太容易建模。添加成员时，要确保大家同意；移除成员时，还要确保其他人不会继续向被移除的节点发送私密消息。

00:39:54 · Julian Haas

论文里有一些例子说明如何处理。基本做法是把成员列表也纳入数据类型，作为共识的一部分。例如，每一轮共识做出决定时，同时决定下一轮的成员集合。当然，对集合做任何修改，都需要当前集合的多数派参与。共识算法通常都是如此。例如 etcd，它基于 Raft 共识协议，是一种分布式键值存储，可以运行在多台服务器上，由这些服务器进行协调。在这种情况下，安全地添加或移除成员，也需要能够联系到大多数服务器。PRDTs 同样可以这样做：把下一组成员的决定纳入共识过程。这回答了你的问题吗？

是否必须有多数派参与

00:41:19 · 观众（聊天提问，由讲者代读）

对于有冲突的操作，也就是不合流的操作，例如休假例子，协调总是需要大多数设备参与吗？PRDT 能否只让更少的设备参与，例如仅涉及相关的设备？

00:41:39 · Julian Haas

我觉得这在一定程度上取决于应用：你如何组织和实现它。例如，应用中可以有不同的设备子集，有些设备只能访问休假日历，另一些访问工作日历。这样可以使用多个 PRDT，让各自的子集只协调相关部分。但一般来说，没有多数派就更难确定操作是否安全。当然，也可以把共识算法替换成其他集中式机制，比如区块链、集中式键值存储之类的方案。

00:42:49 · Julian Haas

总体来说，我认为需要多数派，除非使用其他技巧。举个可能有点深入的例子：有些共识算法采用网格结构，不要求全部节点的多数，而是把副本组织成矩阵，通过完整的行或列来组织同意决定的集合。关键是每次决定所用的成员集合，都要与其他决定所用的集合相交。共识里称之为 quorum intersection，也就是法定人数集合相交。没有这种相交保证，就无法保证安全。

使用 TLA+ 验证

00:43:55 · 观众（聊天提问，由讲者代读）

如何用 TLA+ 来证明？

00:44:07 · Julian Haas

我不确定你具体指哪一部分、想证明什么。我相当确定，为 PRDT 模型定义的那两个条件，也可以在 TLA+ 中验证；它们应该不难写出来。TLA+ 通常有一个问题：验证的是实现的模型，你还需要保证模型与最终运行的代码对应。例如我用 Scala 编程、用 TLA+ 验证，就必须确保 TLA+ 中的描述与 Scala、Rust 或其他语言的实现一致。如果只是讨论理论框架与定理，我认为可以用 TLA+ 验证。

00:45:20 · Julian Haas

还有其他问题吗？暂时没有看到。

应用特定的条件与决策

00:45:45 · 主持人

我有一个问题。我们可以写不同的协议条件或阈值条件，规定在什么情况下做出决定，对吗？能否再谈谈，PRDTs 在这方面能支持哪些可能性？

00:46:31 · Julian Haas

谢谢提问。我们理论的核心有两点：首先，动作的前置条件必须足够强，以保证这些确定的决策；其次，条件必须稳定，不会被其他设备的操作破坏。常见做法是依赖多数派：一旦形成支持某个值的多数，就不能再形成支持另一个值的多数。但如果适合应用，而且仍能证明决定的单调性，我相信也可以采用其他定义。

00:47:45 · Julian Haas

举例来说，我可以不直接实现共识算法，而是写入集中式数据库或区块链，再读取结果。这样的结果是稳定的，其他人不能再影响它。不过，我猜你更想问的是能否定义更细粒度的决策，让它们并发推进，而且不需要每个设备都参与。这与前面说的有关：如果应用包含不同且相互独立的决策，只有某些设备影响特定决策，对其他决策并不重要，就可以尝试把它拆成不同的 PRDT，各自推进，甚至根据应用中发生的情况作出适应。

00:48:52 · Julian Haas

论文里有一个购物应用的例子：结账、付款、发货彼此相关，需要按顺序进行，但这只针对某一位顾客。多个顾客同时购物时，不同顾客的决定可以独立推进。关键是在 PRDT 中明确这些依赖关系。我不确定解释得是否足够清楚；论文里有一个例子，将 PRDT 的一部分组织成前驱映射：每个决定对应一组先前的决定，等它们完成后，才能做出新决定。如果没有前驱，就可以独立推进。也就是说，为每个决定说明它依赖什么，就能更细粒度地建模。

00:50:16 · 主持人

非常感谢。还有其他问题吗？

并行执行

00:50:29 · 观众

我觉得这种语言也适合让程序并行运行。可以把复杂程序分成许多子过程，只在这些条件要求的地方进行协调，其余部分则自动并行运行。

00:51:12 · Julian Haas

确实如此。这里都是不可变数据类型，决策过程或者 PRDT 会明确指出哪些地方必须协调，其他地方则不需要。这一点很好。

CORDA 与整体研究方向

00:51:55 · 主持人

我也想问一下另一个正在进行的项目 CORDA。它的标题很有意思：“Think Locally, Act Globally”，也就是“本地思考，全局行动”，对吗？

00:52:23 · Julian Haas

对，是这个标题。CORDA 与其说是另一个独立项目，不如说是统合这些组件的整体框架。LoRe 是其中一部分，PRDTs 也是。基本想法是：如何设计一个完全基于本地推理的编程模型？首先写一个在本地运行、做本地决策的应用，同时它也是本地优先应用，本地操作最终会传播到其他设备，从而产生全局结果。CORDA 想解决的是，怎样安全地做到这一点？

00:53:24 · Julian Haas

如何保证决策正确？我的一位同事正在研究访问控制，这也是很重要的一部分。在去中心化环境里，如何确保只有部分对等节点能读取某些数据，而其他节点不能？这是 CORDA 中我今天没有展开的一部分。我们正在研究，希望不久就能有一个解决方案。

00:54:05 · 主持人

所以，这是涵盖这些工作的更大项目。

00:54:14 · Julian Haas

对，完全正确。

结束交流

00:54:31 · 主持人

如果没有其他问题，我们今天就到这里。非常感谢你，Julian。

00:54:41 · Julian Haas

谢谢大家的关注，这次交流很有意思。关于博客文章和录像，我们继续保持联系。也祝后续演讲以及本地优先周的活动顺利。

00:54:58 · 主持人

非常感谢。我们会把资料发给你，保持联系。

00:55:04 · Julian Haas

好的，谢谢。大家再见，谢谢你们的提问。