App-Specific Coordination and Convergence With LoRe and PRDTs

Julian Haas • 2026-09-28 • English

Full presentation and discussion

Edited transcript of the full presentation and Q&A. Revised on 29 September 2026 against the Soniox transcript, Zoom captions, and presentation material; the Chinese version was checked paragraph by paragraph against the English. Repetitions and broken sentences have been edited for reading, while examples, qualifications, and the speaker's claims are retained. Timestamps begin at 00:05:04 in the source recording. Setup and waiting are omitted. Unresolved wording is marked. This is not a word-by-word manual listening audit.

Introduction



Presentation frame • 00:00:06

00:00:00 • Julian Haas

Then I'll introduce myself again. Hi everyone. I'm Julian, and I'm a PhD student at TU Darmstadt in Germany, where I research programming languages for local-first software. And today I'm going to briefly talk about two of my research papers, which are LoRe and PRDTs. Both of them are concerned with how the data consistency of our local-first application interacts with the application-specific logic. And this talk specifically is going to be a bit of a director's cut, so to speak, because I combined two of my presentations. So I hope it's not too confusing, but in the end we probably have time for questions, so I'm happy to answer anything that might leave question marks after the presentation.

00:01:09 • Julian Haas

What I was about to say is that the first work I'm going to talk about is LoRe, which is a programming language for verifiably safe local-first software, which we published at ECOOP three years ago, and there's also an extended journal version of that article. But yeah, let's get right into it.

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

Presented at ECOOP '23
Published in TOPLAS '24

https://doi.org/10.1145/3633769


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

1 / 47

Presentation frame • 00:01:16

From clouds to local-first software

00:01:37 • Julian Haas

The motivation for this is moving from centralized clouds to local-first software. And I'm sure I'm not telling you anything new here because you already had a bunch of talks on this topic, but pretty much most of our applications nowadays look something like this. So your application actually lives in some cloud, in some data center, and then your devices mainly connect to that and serve as like a window to that application on the server. And as we all know, this has several downsides. First of all, you can't really use these kinds of applications when they're offline. So if your connection here breaks, then you can't access the application anymore.

00:02:23 • Julian Haas

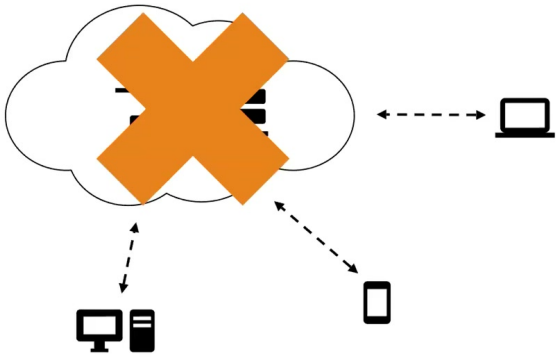
But also it raises questions of data privacy and ownership, because many times you don't really know where those servers actually stand, who owns them, who has access to my data, who can actually read the stuff that I'm sending to the server. So this is also a motivation for local-first software. And finally, another important point that I find there is also user agency.

00:02:53 • Julian Haas

And what I mean by that is, with cloud applications, there's no guarantee that they will be there forever. And we have had many examples where companies just shut down their servers, and if all your data lives with them, then you don't really have an option to actually continue using the application, even if you wanted to. So in my view, the idea of local-first is to change that picture to something like this, where first and foremost, the devices communicate with each other, and every device has a local copy of the important application data. And if you want to, you can still use clouds, and you can still send them data, but they should not be this integral part of the application. Without them, the application should still continue to work.

Motivation: From Centralized Clouds to Local-First Software

The Problem With Modern Cloud Software



Problems:

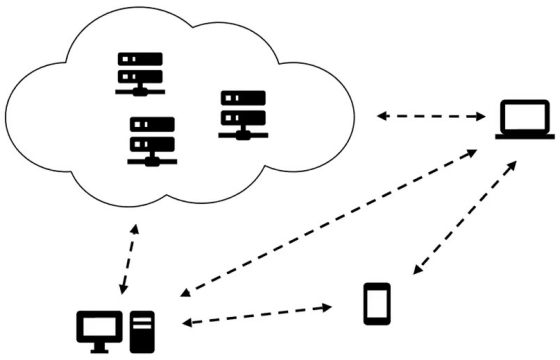
- Offline Availability
- Data Privacy & Ownership
- User Agency

3 / 47

Presentation frame • 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

Presentation frame • 00:03:45

00:03:45 • Julian Haas

And to finish this introduction part, the most common way to implement these applications nowadays is by using CRDTs. And CRDTs avoid merge conflicts by restricting the operations that you could do in your application and by structuring your data in a certain way. So to illustrate this a bit, think of a distributed calendar application, for example. So what I could do here is I could add an event on my phone, for example, a lunch meeting, and later on the laptop, I add a group meeting to the calendar. And then once these devices connect again, they can simply merge their states, and we end up with a calendar that has both events inside of them. And the key properties that you want from CRDTs are commutativity, associativity, and idempotence.

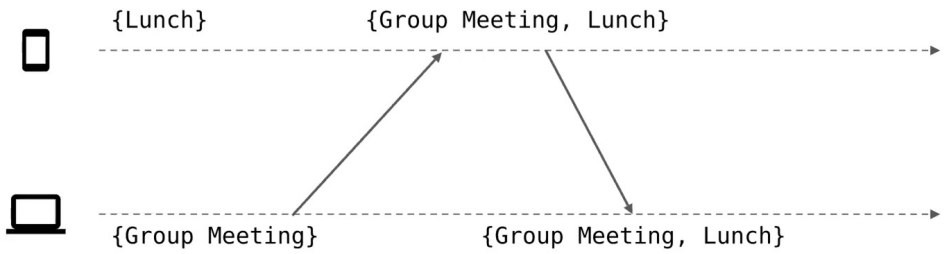
00:04:43 • Julian Haas

But I won't go into them in much detail, but just to visualize this a bit differently, essentially what you get is that these states in your system, they form what is called a join-semilattice. So in the end, you can merge every point of data. So this kind of represents the spectrum of your application data, and devices can diverge here, but they can always reconcile their application states again. And this is what we call eventual consistency. So two replicas will eventually end up in the same state once they've seen the same set of updates. Now, the problem with this—and this is like the main hook for my talk—is that for some applications, this is just not enough, right? So imagine, for example, that we now wanted to use the calendar to track vacation days, and maybe the company enforces that employees can only take 30 days of vacation.

Motivation: From Centralized Clouds to Local-First Software

Enter CRDTs

(conflict-free replicated data types)



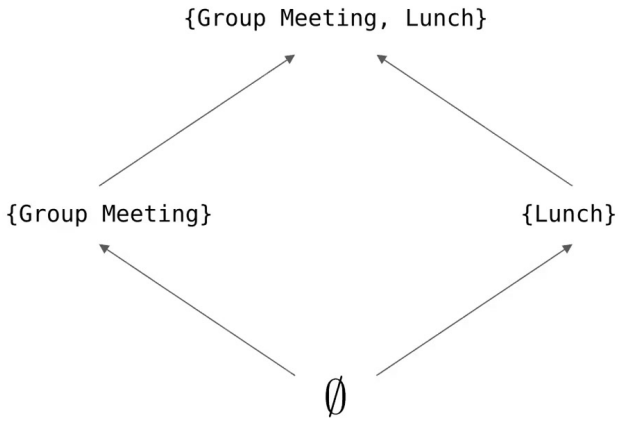
Commutativity - Associativity - Idempotence

5 / 47

Presentation frame • 00:04:46

Motivation: From Centralized Clouds to Local-First Software

CRDTs as Lattices



Eventual Consistency:
*Two replicas will **eventually** end up in the same state once they have seen the same set of updates.*

6 / 47

Presentation frame • 00:05:17

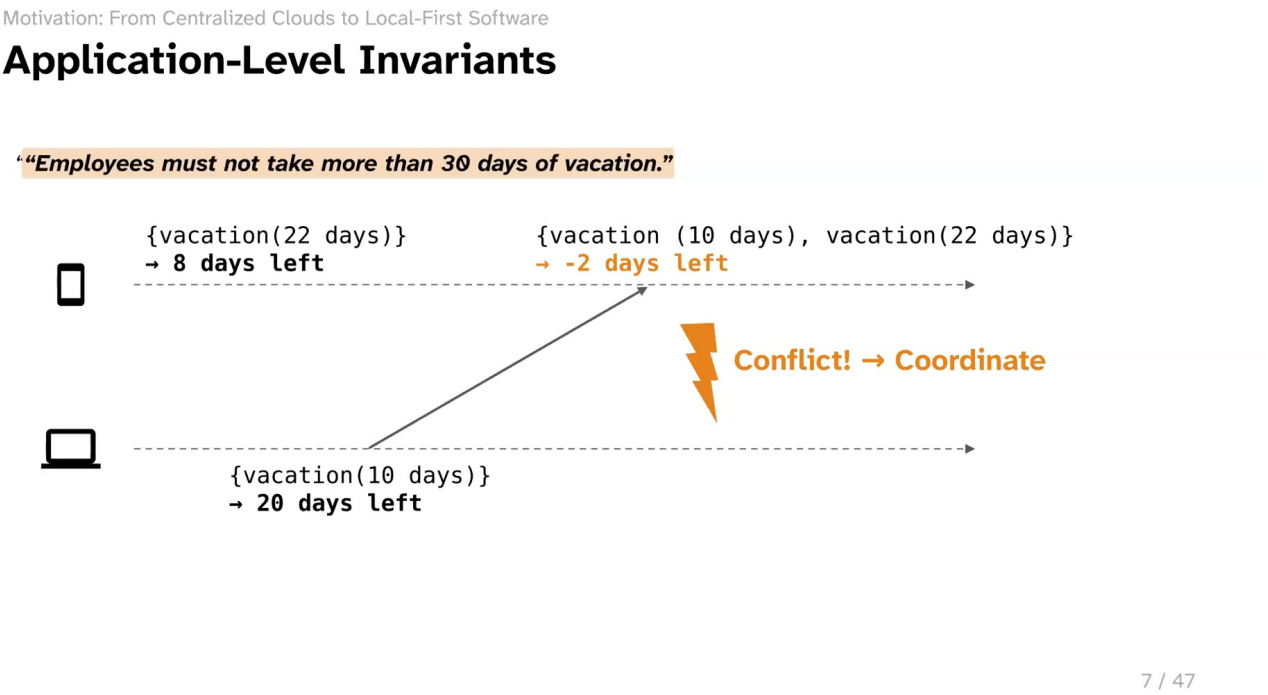
The limits of *eventual consistency*

00:05:45 • Julian Haas

I know this is a very German perspective on vacation, but yeah, we usually have 30 days. Yeah, and what you could do here is that I could, on my smartphone, add a vacation of 22 days. So the smartphone thinks I have 8 days left. Then later on my laptop, I add another vacation of 10 days, so the laptop thinks there are 20 days left. And you probably see where this is going, because once I merge those two, I have both dates in my calendar, but I actually use too much vacation.

00:06:19 • Julian Haas

And this is just one example of such a conflict that could arise. And you have these conflicts whenever you have some kind of a limited resource in your application. For example, maybe you want to sell tickets, or maybe you want to send—you want to ship parcels, and you want to make sure that you don't ship the same thing twice. Then these are the kinds of points in your application where you need coordination. So the devices actually have to coordinate with each other who does what in order to prevent these kinds of conflicts.



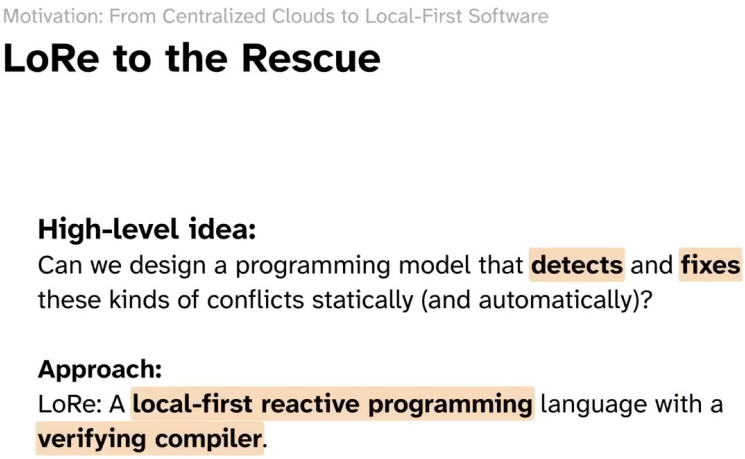
7 / 47

Presentation frame • 00:06:43

The LoRe programming model

00:06:52 • Julian Haas

The idea behind our LoRe work was, can we design a programming language that enables us to automatically reason about these kinds of conflicts? And the answer that we were proposing is this LoRe language, which is a local-first reactive programming language with a verifying compiler. Okay. I know that's a lot of complicated words, but let's look into that and see how this works in a nutshell. So basically, the architecture that we envisioned looked something like this. So we want the user to provide the input program, then we automatically, in the compiler, analyze that input program to analyze its data flow. Then from that, we generate verification code, which we pass to an automatic program verifier. In this case, we used the Viper verification toolkit.



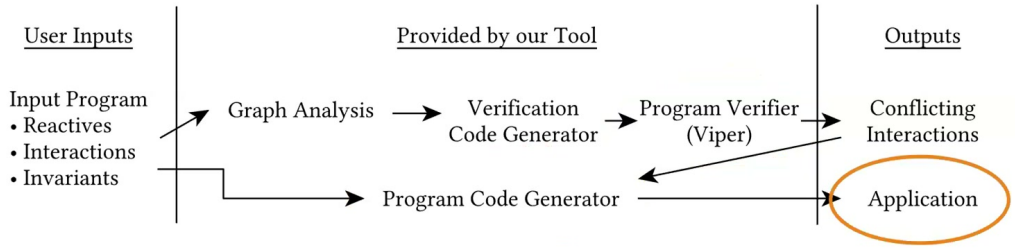
Presentation frame • 00:07:13

00:07:51 • Julian Haas

And the job of Viper is then to analyze which kinds of interactions could happen in the program and where those interactions could lead to conflicts. This is then fed back into the compiler, who can ideally fix these conflicts in the resulting application.

LoRe in a Nutshell

Architecture Overview



Presentation frame • 00:08:08

00:08:11 • Julian Haas

Now, how do we fit this into a programming language? For LoRe, we came up with a model that combines three main building blocks: reactives, interactions, and invariants. And I'm going to show you a small code example to see what this looks like. Again, I hope it's not too confusing. So this is Scala syntax. So we implemented this as a domain-specific language in Scala. And yeah, I'll just go through each of the components. So what we're implementing here is this calendar application that I teased before. So in this case, we define two reactives, in this case two calendars, a work and a vacation calendar. And in LoRe, reactives are basically your CRDTs. This is where your data lives. So in this case, both of them are simply add-wins sets.

LoRe in a Nutshell

Syntax

```
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
}
```

12 / 47

Presentation frame • 00:08:41

00:09:07 • Julian Haas

All the events that I add in there, they are just combined in the set, and I add everything into a set. But because this is a reactive programming language, those reactives can also be chained together. And you see that in this data flow graph here on the right. So in addition to those source reactives, we can define what we call derived reactives. And these are simply parts of the application state that are computed based on their inputs. So in this case, we can, for example, say, okay, we want an all_appointments reactive, which combines all appointments from both calendars, and we want a remaining_vacation reactive, which always calculates how many days of vacation you have left based on your vacation calendar.

LoRe in a Nutshell

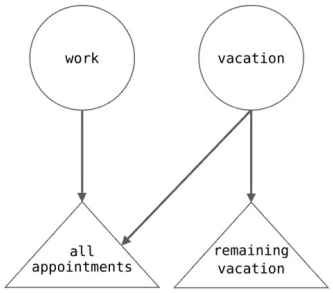
Syntax: Reactives

```
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

Presentation frame • 00:09:37

00:09:58 • Julian Haas

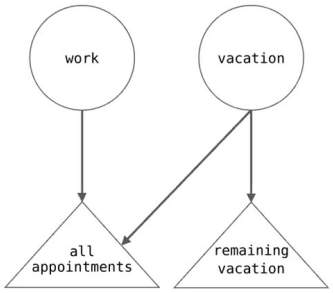
Next are interactions, which update the reactives. Here we define add_appointment to add appointments to the calendars. The requires, executes, and ensures clauses tell the verifier what the interaction does. Each interaction can have preconditions and postconditions for verification. The add_vacation example also has a modifies clause restricting which reactive it applies to: it adds vacation only to the vacation calendar and introduces a precondition specific to vacation.

LoRe in a Nutshell

Syntax: Interactions

```
1  val work: Source[Calendar]     = Source(AWSet())
2  val vacation: Source[Calendar] = Source(AWSet())
3
4  val all_appointments: Derived[Set[Appointment]] = Derived {
5   work.value.union(vacation.value)
6  }
7  val remaining_vacation: Derived[Int] = Derived {
8   30 - sumDays(vacation.value)
9  }
10 val add_appointment = Interaction[Calendar, Appointment]
11    .requires { (_, a) => a.start < a.end }
12    .requires { (cal: Calendar, a: Appointment) => !cal.contains(a) }
13    .executes { (cal: Calendar, a: Appointment) => cal + a }
14    .ensures { (cal: Calendar, a: Appointment) => cal.contains(a) }
15 val add_vacation = add_appointment.modifies(vacation)
16  .requires { (cal, a) => remaining_vacation.value - a.days >= 0 }
17
18 val i1 = Invariant {
19  all_appointments.value.forall { a => a.start < a.end }
20 }
21 val i2 = Invariant {
22  remaining_vacation.value >= 0
23 }
```

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



14 / 47

Presentation frame • 00:10:39

00:10:52 • Julian Haas

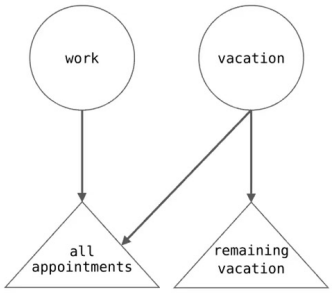
But maybe that's too much detail already. Anyway, so the third part, and maybe the most important part then, are the invariants. And this is how you specify what your application should actually behave like. So what should the guarantees be that you get out of this? And in this case, we define two invariants. One, which simply enforces that all appointments start before they end. And the second one simply requires that we never go below zero days of vacation that we have left. So remaining vacation stays above zero. Okay. Now, this probably kind of makes sense, but how do we actually verify this now? How do we ensure, or how do we see which parts can lead to problems and which interactions could cause conflicts?

LoRe in a Nutshell

Syntax: Invariants

```
1  val work: Source[Calendar]     = Source(AWSet())
2  val vacation: Source[Calendar] = Source(AWSet())
3
4  val all_appointments: Derived[Set[Appointment]] = Derived {
5   work.value.union(vacation.value)
6  }
7  val remaining_vacation: Derived[Int] = Derived {
8   30 - sumDays(vacation.value)
9  }
10 val add_appointment = Interaction[Calendar, Appointment]
11    .requires { (_, a) => a.start < a.end }
12    .requires { (cal: Calendar, a: Appointment) => !cal.contains(a) }
13    .executes { (cal: Calendar, a: Appointment) => cal + a }
14    .ensures { (cal: Calendar, a: Appointment) => cal.contains(a) }
15 val add_vacation = add_appointment.modifies(vacation)
16  .requires { (cal, a) => remaining_vacation.value - a.days >= 0 }
17
18 val i1 = Invariant {
19  all_appointments.value.forall { a => a.start < a.end }
20 }
21 val i2 = Invariant {
22  remaining_vacation.value >= 0
23 }
```

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



15 / 47

Presentation frame • 00:11:17

Invariant preservation and confluence

00:11:46 • Julian Haas

And this is what we're looking into now. So remember, the question we're asking here is, could the given interactions violate the given invariants? And here we already see how the reactive data flow that we see on the right can help us with this. Because if we wanted to verify invariant number two, this remaining vacation one, we can look at the data flow graph. And we see, okay, remaining vacation is down here. And the only source of change to remaining vacation is this vacation reactive up here. So out of the complete program, which could be much more complicated, we see that the only interaction we actually need to verify is this one down here, because this influences vacation, and this is the only source of change that could flow into the remaining vacations. Okay. And then for these interactions, we need to verify two things.

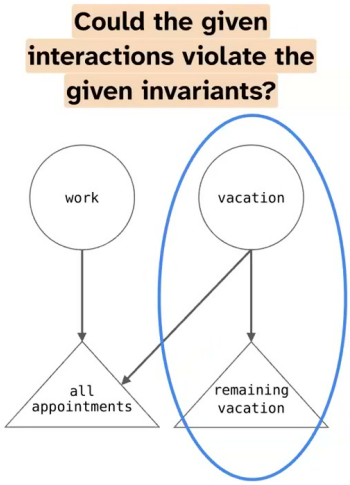
Semantics and Verification

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

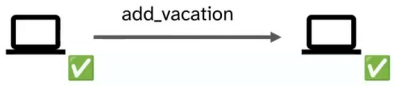
Presentation frame • 00:12:37

00:12:47 • Julian Haas

And we define them as two verification conditions. The first one is invariant preservation, and the second one is confluence. And we'll now look into them in a bit more detail. So preservation is relatively straightforward, and this is likely something that you always want in your application. It simply means if the invariant holds in one state and that device does something, so it does some kind of interaction, then we still want the invariant to hold. So so far, so good. But you probably ask yourself, okay, but what if stuff happens on other devices concurrently? Couldn't that lead to problems? And yeah, that's what the second condition is for. And we call that one confluence. And for confluence, what we want is that two concurrent interactions always lead to the same result, no matter at which point the merge happens.

Semantics and Verification

Invariant-Preservation



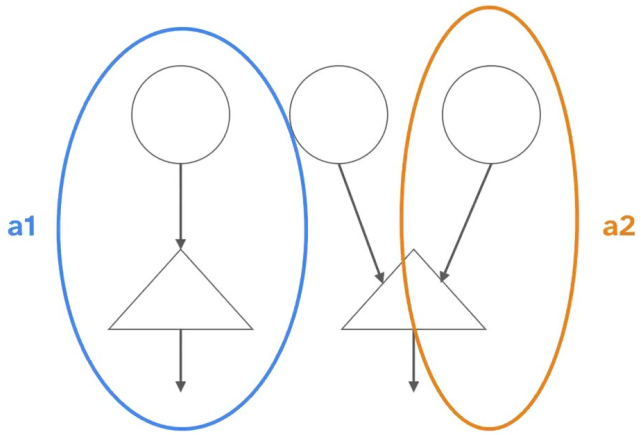
Presentation frame • 00:13:20

00:13:49 • Julian Haas

Basically, it shouldn't matter if both devices first add their events and then merge, or if they merge beforehand, or somewhere in between. And again, we can easily see that, or the data flow graph makes it a bit easier to check that. Because if we have two interactions, A1 and A2, and they simply affect two completely separate parts of the data flow graph, then we know that they're confluent. Like, they will never get into each other's way. But obviously, we've seen this example before of something that is not confluent. So here, the merge actually influences the outcome. Because if those two devices would have merged earlier, they would have seen that they can't add all of these vacation days. So what do we do in this case? We simply enforce that those interactions don't happen concurrently. So instead, they need to be coordinated.

Semantics and Verification

Invariant-Confluence

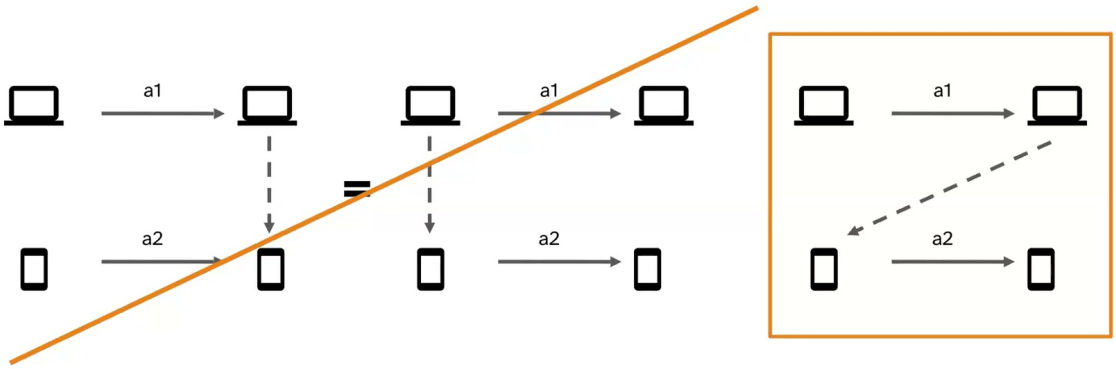


22 / 47

Presentation frame • 00:14:12

Semantics and Verification

Non-Confluence



24 / 47

Presentation frame • 00:14:46

00:14:49 • Julian Haas

And in this case, the laptop has to coordinate with the smartphone who adds the interaction first. So yeah, in summary, the idea is basically this. So every device locally checks invariant preservation, and that what they are going to do is legal invariant-wise. But for things that are not confluent, the devices also have to coordinate with each other in order to ensure that they don't do stuff concurrently.

Mixed consistency and coordination

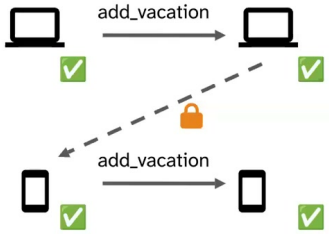
00:15:24 • Julian Haas

Taking a step back, you might now see that we have kind of a mixed consistency model. Like, most interactions just update the CRDTs, they run without coordination, and give you eventual consistency. But we also have some interactions which require coordination. And obviously, this raises the question, okay, how do I get the coordination into my local-first app, and how do I make sure that those two devices actually agree? And we were not the first to ask this question, and there are many papers that actually provide different mixed consistency programming languages and systems. Unfortunately, though, the most common approach to coordination used by these systems is to simply use a centralized server or a database again. So you end up with kind of this nice situation here, which we likely don't really want in a local-first application, because now we just shoved a server inside it again to realize this coordination. So in the second part of the talk that I'm getting into now is, we are asking the question, how do I get the coordination into my local-first app without relying on a central server? And this is where the PRDTs come in, that we quickly mentioned in the beginning. So in this work, we check, how can I add coordination to my local-first app using just replicated data types?

Semantics and Verification

Verification Summary

Safety through coordination + invariant preservation

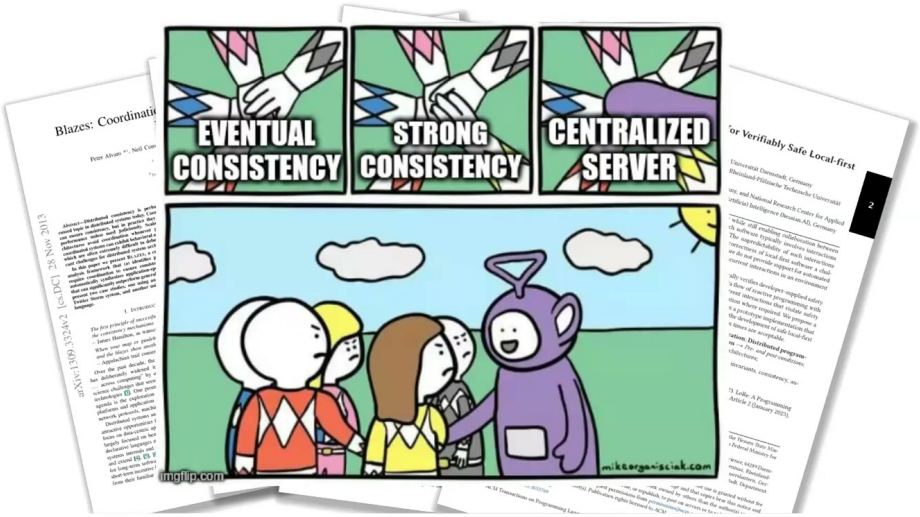


25 / 47

Presentation frame • 00:15:26

Semantics and Verification

The Problem With Mixed-Consistency



27 / 47

Presentation frame • 00:16:33

00:17:02 • Julian Haas

To restate the setting, we are trying to implement a consensus protocol. Multiple devices need to agree on a value or outcome, such as who acts first. More simply, imagine three devices choosing between cat and dog. If two vote for cat and one for dog, the outcome should be cat.

00:17:38 • Julian Haas

Now, while this might seem like a relatively simple problem, it's actually surprisingly hard to get right in practice. And the two main reasons for that are that distributed systems are unreliable. So what if those votes get dropped? And secondly, consensus is also quite expensive. Because if we make it right, so if all of those devices actually make sure that everybody has seen the votes and knows the decision, this is expensive. Everybody needs to be online, and you need to exchange a lot of messages. So as a result of that, there are actually a lot of different consensus algorithms for different settings. So whether you're in a data center, on different continents, or if you have mobile devices that communicate directly, that has a lot of impact on which consensus protocol actually works well in your setting.

The Setting

Consensus Is Challenging!

1. Distributed systems are inherently **unreliable**.
2. Consensus is **expensive**.

31 / 47

Presentation frame • 00:18:11

Building consensus with PRDTs

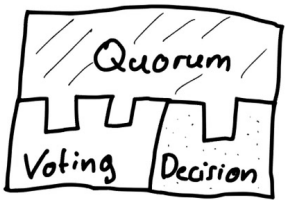
00:18:38 • Julian Haas

Now, this is where our work with PRDTs comes into play. PRDTs are protocol replicated data types. And the idea is that this is a programming model for compositional implementation of consensus protocols using replicated data types. So with PRDTs, you can stack or compose your different building blocks to build new protocols. And at the same time, we again want to provide an automated verification framework that can tell you whether your composition is actually still a safe consensus protocol. Okay. Again, how do we do that? The main inspiration, or one of the inspirations for this work, actually goes back to a paper from 1990. Back then, Halpern and Moses have shown that we can treat agreement or consensus in a distributed system as a question of knowledge.

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

Presentation frame • 00:19:11

00:19:36 • Julian Haas

And the idea was that essentially what we're trying to do is, we try to lift distributed knowledge. So this is something that only some devices know, to common knowledge. And common knowledge is something that we know everybody will eventually know. So in this case, you could say, in the beginning, everybody only knows their vote. So everybody only knows cat or dog. But once they've exchanged enough information, they know what the outcome should be. And the key here is that this is a monotonic process. So like CRDTs, devices keep exchanging messages or keep exchanging these bits of knowledge until everybody can safely draw a conclusion. And now our idea with PRDTs was to use replicated data types as a kind of storage for this knowledge. So you add the bits of knowledge to your data type, and then over time, it accumulates to something that constitutes a decision.

Core Idea

Agreement Is a Question of Knowledge

Halpern & Moses, 1990:

distributed knowledge
known by some devices

→

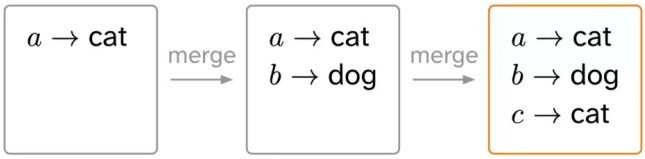
common knowledge
eventually known by every device

34 / 47

Presentation frame • 00:19:52

Core Idea

Knowledge Grows Monotonically



35 / 47

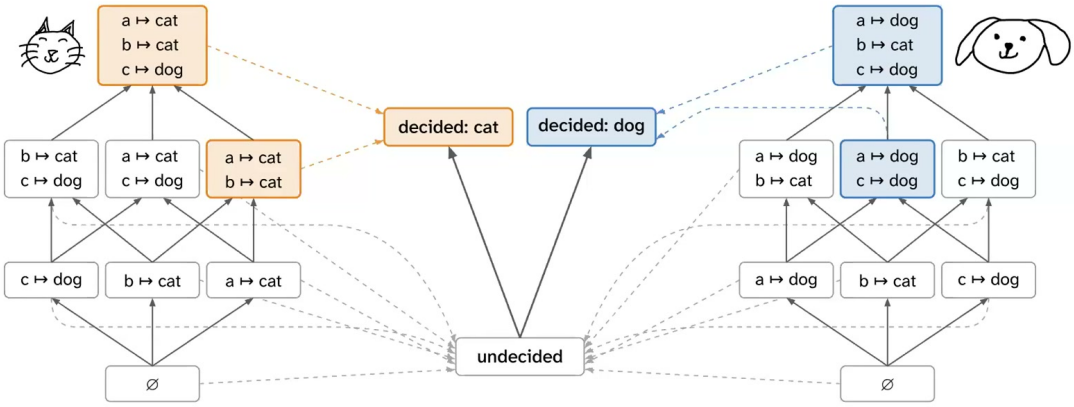
Presentation frame • 00:20:31

00:20:40 • Julian Haas

Now, to give you an example of this process, consider this diagram. On the left and right here, we see how the knowledge in this system could evolve in two different cases. And in the middle, we see the conclusions that we can draw from that. So on the left side, we see that cat wins in the end, while on the right side, we see that dog wins in the end. Now, the main thing that a correct consensus protocol, and therefore a correct PRDT, has to ensure is that we only ever end up on the left or on the right, but not in between. So basically, every device has to live either in the left world or in the right world, but we're not allowed to mix them.

Core Idea

Knowledge Grows Monotonically



36 / 47

Presentation frame • 00:21:06

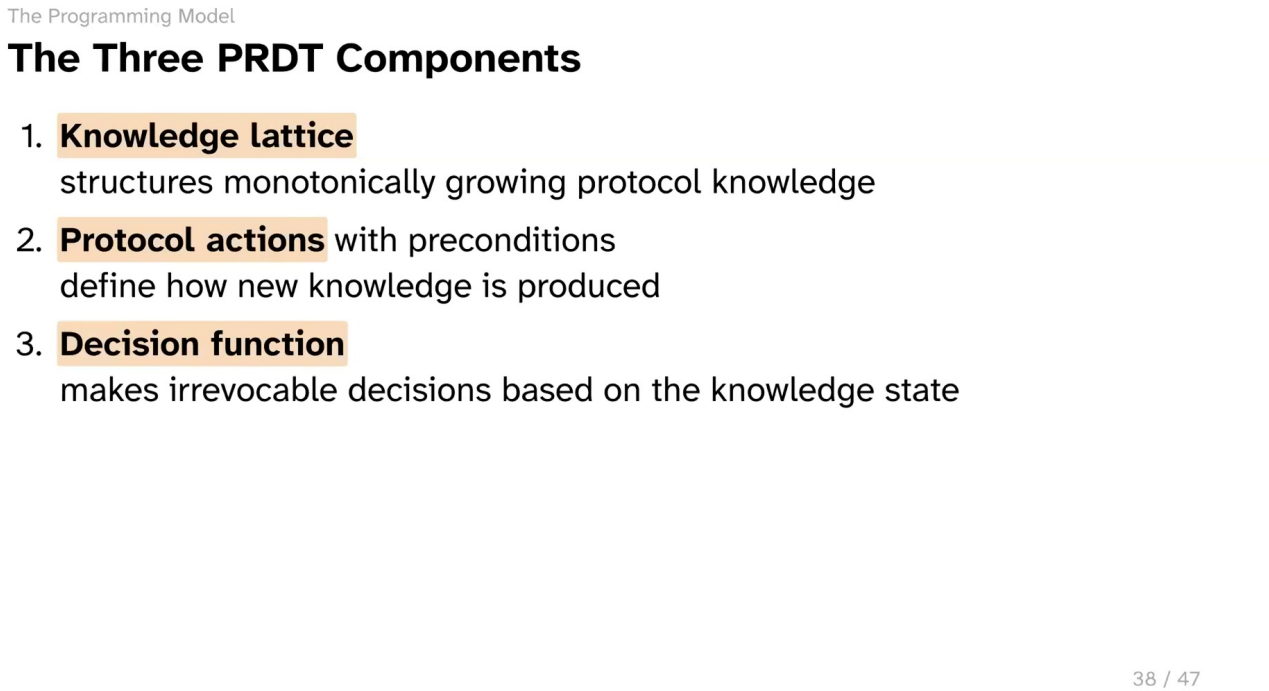
Knowledge lattices, actions and decisions

00:21:24 • Julian Haas

And in order to do that, we again propose a programming model with three components. And I'm going to show you some more syntax in a bit. But basically, the three components are first, the knowledge lattice. And this is essentially this lattice structure that you've seen before. So this is a data type that stores all the knowledge in the system and that grows monotonically. Then again, we need to define actions. And they also have preconditions, which limit what kind of knowledge we are adding to the knowledge lattice. And in this case, you can use these preconditions to restrict what kind of knowledge you're adding in order to actually make this a safe protocol. So for example, we could say everybody can only vote once.

00:22:22 • Julian Haas

And then the final component that we want here is a decision function. And the decision functions were basically the arrows that you've seen before. And the decision function makes irrevocable decisions based on the knowledge state. And that irrevocable is the key word here, because we want to really be sure that whatever we define as a decision can never be taken back again. Okay. Then we have Scala code again. And this is an implementation of this very simple voting protocol that I've shown you before. Each device can only vote once, and the value with the most votes wins.



38 / 47

Presentation frame • 00:22:42

00:23:06 • Julian Haas

We start with a knowledge lattice. The relevant knowledge is each participant's vote. This compound data type has a single field, votes, mapping each voter ID to the value they voted for. The second component is protocol actions, which define how knowledge is added. Here, voteFor lets you add a vote for a value by returning a map from your local replica ID to that value.

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

Presentation frame • 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

Presentation frame • 00:23:51

00:23:56 • Julian Haas

And you might also see that this has a precondition. And the precondition here is hasNotVoted. So we are checking if there is a vote by our local replica ID in this map already. And if that is the case, we cannot vote again. So hasNotVoted simply enforces that we only vote once. Now, the final component is the decision function, as I've mentioned before. And in this case, this is relatively simple. The only thing we're doing is we're counting the votes, checking if there's a majority. So if more than half of the participants have voted for the same thing, and if yes, we can return the decision. Otherwise, we have to return undecided.

The Programming Model

Voting as a PRDT

```
1  val participants: Set[Uid] = ...       // fixed for all processes
2
3  case class Voting[A](votes: Map[Uid, A]) {       // knowledge state
4
5    def hasNotVoted(using LocalUid): Boolean =     // precondition
6      !votes.contains(replicaId)
7
8    def voteFor(value: A)(using LocalUid): Voting[A] =   // protocol action
9      precondition(hasNotVoted) {
10       Voting(Map(replicaId -> value))
11     }
12
13   def decision: Decision[A] = getLeadingValue() match  // decision function
14     case Some(value, count) if count > participants.size / 2 => Decided(value)
15     case _ => Undecided
16 }
```

39 / 47

Presentation frame • 00:24:07

Composing protocols

00:24:44 • Julian Haas

I hope that's relatively clear, but we can get back to it later. A key benefit of PRDTs that I hinted at before is their composability. And because PRDTs are essentially just Scala algebraic data types, we can also compose them like we're used to from algebraic data types. So in this example that we see here, we can simply say, okay, I want to define a parallel voting protocol where two votes happen in parallel. And this is simply composed of two voting instances up here, which are the voting that you've seen before. And in this example, we're not adding any new protocol actions. So you can still vote separately by accessing the components. But we do define a new decision function. And in this case, we say, okay, parallel voting only returns a decision when both components have made their decision already.

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
}
```


Presentation frame • 00:24:45

The Programming Model

PRDT Composition

```
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
}
```



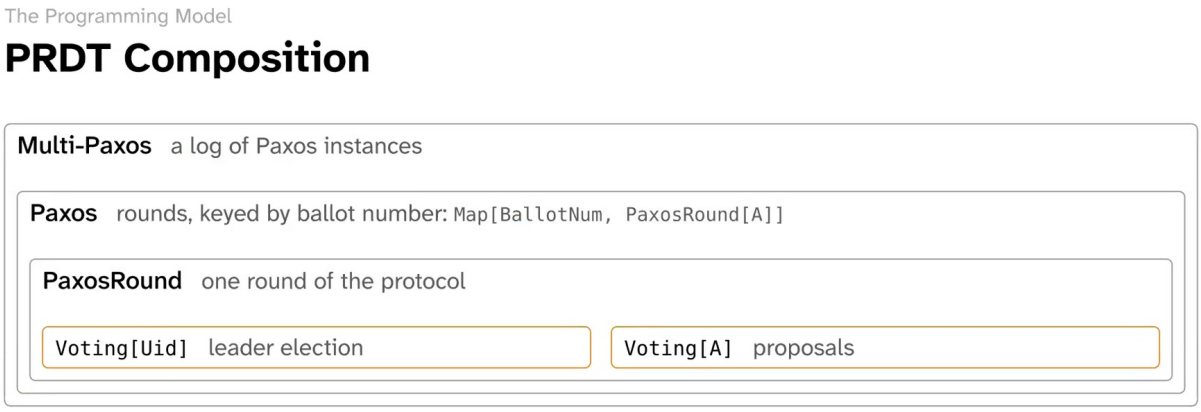
Presentation frame • 00:25:41

00:25:46 • Julian Haas

Otherwise, the result is still undecided. We could have defined this differently, depending on the application. Here we wait for both components, but we could return earlier with a partial decision: voting one has a result, while voting two is still unknown. In this example, we chose to wait.

00:26:24 • Julian Haas

Now, in the paper, there are many more complicated composition examples. So for example, we show how to compose multi-Paxos from different Paxos instances, which is composed of different Paxos rounds, which uses multiple votings internally. I don't have the time, and I don't have the space on the slide to show you the code for that. Yeah. But I hope you believe me that this scales to more complicated cases, and you can have a look at the paper if you're interested in that.



41 / 47

Presentation frame • 00:26:43

Verifying protocol safety

00:26:54 • Julian Haas

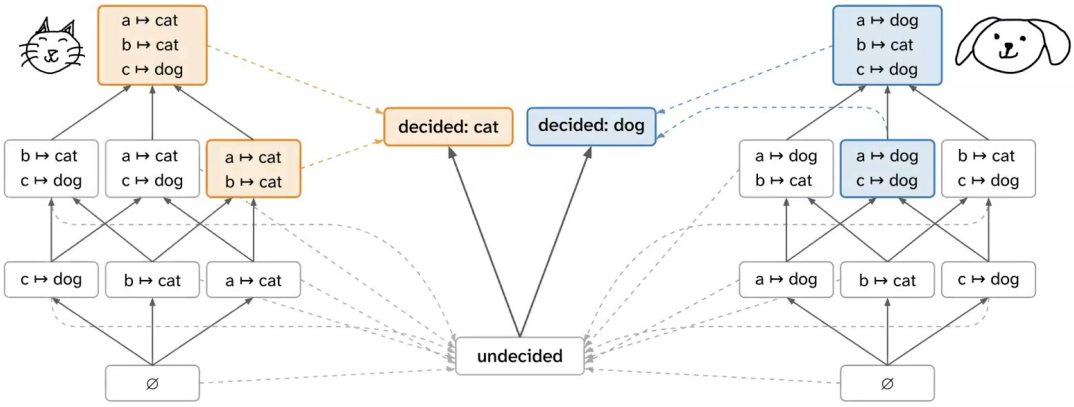
So far, we have talked about the design part of PRDTs, which allows you to assemble different protocols from smaller building blocks and easily create variations of existing implementations. However, as with LoRe, I have also argued that the correctness is the hardest part of consensus algorithm design and that we want automatic verification for that. So how do I know if my composed protocol is actually a correct one?

00:27:25 • Julian Haas

And for this, I'm coming back to this image from before to give you some intuition. And basically, what we are trying to do is to ensure that in this decision order that you see here in the middle, decisions can only grow. So we can go from undecided to decided cat, but we can never go back. And similarly, once we've committed to one side, so once we're either in the cat kingdom on the left or the dog domain on the right, there's no going back. So we have to make sure that decisions only grow, but they also grow into one direction that is the same for all the devices.

Verification

When Is a PRDT Safe?



Presentation frame • 00:28:01

00:28:09 • Julian Haas

With PRDTs, we can reduce this global condition to two local verification conditions: action monotonicity and stable preconditions. The global condition concerns the states of every replica in the network. Action monotonicity requires a replica's local actions only to advance its decision, never move it backwards. Stable preconditions make this reasoning possible: an action that is enabled locally cannot be disabled by something another device does. This resembles what we saw with LoRe. Essentially, we check that the preconditions determining whether an action is legal cannot be invalidated by another device's concurrent action. These two conditions are relatively easy to check with automated tools. In the voting example, there is one protocol action, voteFor, which adds a vote to the local map. We first check that it can only advance the decision.

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

Presentation frame • 00:28:56

00:29:57 • Julian Haas

Then we look at the decision, and we see, okay, the decision only returns when there's a majority. And assuming there is a majority already, adding a vote to that set cannot change that majority. This is because we assume a fixed set of participants. Similarly, if there's no decision yet, so we're still at undecided, then adding a vote to that can only make the decision grow, so we're also fine. Now, the second condition is stable preconditions. And here, we only have one precondition, hasNotVoted, which checks that there is no vote yet for the local ID. And again, this is quite easy to check because this is kind of trivially stable, at least as long as we assume that nobody else can add a vote for our local ID.

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

Presentation frame • 00:30:14

00:30:54 • Julian Haas

To reiterate, the assumption here is that we have a fixed set of replicas, and also that every replica has a fixed ID that only they can access and use to add new votes.

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
}
```

2. Stable preconditions:

Enabled preconditions stay enabled in the presence of concurrent actions.

46 / 47

Presentation frame • 00:31:06

Talk recap

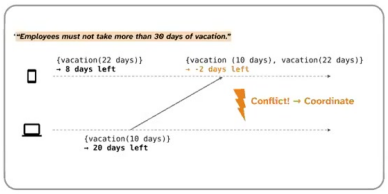
00:31:10 • Julian Haas

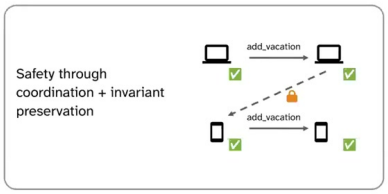
I know that was a lot of content. So I think I should come to the end and quickly recap and summarize so we still have some time for the Q&A. So in the first part of the talk, I discussed how local-first apps sometimes hit the limits of eventual consistency, like we've seen with this calendar example. Then I introduced LoRe and its verification framework. LoRe helps you analyze where this is the case, so where your application actually needs coordination. But unfortunately, coordination is costly, and depending on the setting, we want to apply different solutions or algorithms. And this is what PRDTs give you, because they allow you to implement coordination on top of replicated data types without relying on central servers. So you can easily integrate this into an existing local-first application, and you can stack and nest them to easily implement new protocol variants.

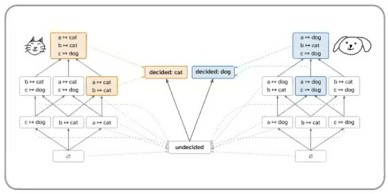
00:32:14 • Julian Haas

And yeah, with this, that's my talk. Here are the QR codes for both of the papers. So if you're interested, I encourage you to read them. They have much more content. And yeah, I'm happy to answer any questions that you might have.

Summary







julian-haas.de
types.pl/@haaase

LoRe: doi.org/10.1145/3633769

PRDTs: doi.org/10.48550/arXiv.2504.05173

47 / 47

Presentation frame • 00:32:36

Discussion and Q&A

00:32:57 • Julian Haas

There is a comment in the chat: “This is really elegant work, expressing consensus as a monotone decision over a replicated knowledge state. That is a lovely idea.” Thank you. Unfortunately, I cannot read the second comment. Could someone translate it? I am not sure whether it is a question or a translation of the first comment.

00:33:33 • Audience

That comment just offers help to anyone who needs assistance asking a question. I can help translate too.

How the research ideas developed

00:34:16 • Audience

I actually got a question.

00:34:19 • Julian Haas

Go ahead.

00:34:21 • Audience

My question is more about your experience: how did you come across these ideas, and how did you come to view consistency as a question of common knowledge? I am interested in how you arrived at that.

00:34:49 • Julian Haas

It was an iterative process. Our group in Darmstadt develops a Scala library for algebraic replicated data types, where you can combine and stack components to build new ones. At some point we asked: what if we built the whole program this way, including its application logic? Could we implement Raft or Paxos with these building blocks? I could not see why not, so we started trying. The common-knowledge insight came later. While we were experimenting, someone recommended a paper from the 1990s. After reading it, we realized, “That is exactly what we have been doing, and this explains why it works.” I did not find the paper first and then have the idea. We were experimenting to see what would work.

00:36:17 • Julian Haas

And then somebody pointed this out and was like, oh, yeah, I think what you're doing has been done 30 years before. But yeah, I mean, theoretically, they didn't have any language or implementation for that. But it's quite a fun read, actually. So if you're interested in that paper, I can quite recommend it. It's a bit philosophical even, because they are really discussing, okay, what can you know based on what you know about what others know. Yeah. It's a bit confusing if you say it that way, but yeah.

00:36:53 • Audience

Many thanks.

00:36:55 • Julian Haas

No worries.

Blockchain as a coordinator

00:36:58 • Audience

Hi. Do you think we can also use blockchain as a coordinator, which is a public common knowledge database everyone can access without centralization? But of course, you need internet access to blockchain.

00:37:18 • Julian Haas

Thanks for the question. You could definitely do that. I think it depends a bit on the application, on what makes sense. A blockchain also is a consensus algorithm, right? And you would usually use a blockchain, I think, if you want to have an open network. So what we are assuming here with PRDTs is that you have a fixed set of replicas, and I know them. You can change the set. You can add new devices, but you can't allow that fluidly. So the system would kind of decide whom to add and whom to remove. And with blockchains, you can kind of do that because they use different consensus algorithms based on proof of work or proof of stake that let you add members or remove members at any time. So if that's what you need for your application, then yes, you could use a blockchain.

00:38:18 • Julian Haas

But honestly, I think that for many applications, it would be a bit overkill. And as you mentioned, yeah, you would also have the centralization problem again. Yeah. But maybe for some applications, it makes sense. If you really don't have any servers that you can trust, but you want something that is available online so that everybody can have a look at it, then I think blockchain could be a viable approach.

Membership changes and revocation

00:38:52 • Audience

Thank you. And the other question is, I think in the asynchronous environment, the hardest part is about membership change. And especially about permission revocation. I think the PRDT thing is very cool. But I can imagine it's a little bit hard to model the dynamic of the membership. You are adding a member, and you need to make sure everything, everyone agree with this. And when you remove a member, you also need to make sure other people are not sending private messages to the removed node.

00:39:54 • Julian Haas

The paper includes examples of how to do this. You make the membership list part of the data type and the consensus process. For instance, in each consensus round, you also decide which members will participate in the next round. Changes require a majority of the existing set. That is generally true for consensus algorithms. For example, consider Etcd, a distributed key-value store based on Raft. It can run on multiple servers that coordinate with one another. You need access to a majority of those servers to add or remove members safely. PRDTs can do the same: make decisions about the next membership set part of consensus. Does that answer the question?

Majorities and quorum intersection

00:41:19 • Audience (chat question read by the speaker)

For the operations that conflict, the non-confluent ones, like the vacation example, does coordinating always require a majority of the devices to participate, or can a PRDT resolve them with fewer, for example, only devices involved?

00:41:39 • Julian Haas

I think that question boils down to the application-specific part to some extent. It depends a bit on how you structure and how you implement your application. For example, we could say that there are different subsets of devices in the application. Maybe only some of them have access to the vacation calendar. Some of them have access to the work calendar. And then you could have multiple PRDTs, and then only those subsets would have to coordinate those parts. But generally, it is a bit harder to do something without a majority, because then you cannot really be sure that it is safe anymore. What you could, of course, do is replace the consensus algorithm with some centralized measure again. So you could use a blockchain. You could use a centralized key-value store, something like that, to do this.

00:42:49 • Julian Haas

But yeah, in general, I think you need the majority, unless you do some other tricks. But yeah, maybe that was too much into detail. But yeah, just to give you a quick hint, there are consensus algorithms that work grid-based, so they don't need a majority of nodes. But they instead say, okay, all the replicas form this grid, so like a matrix. And then we can say, okay, majority in our setting means any complete row or column in that matrix needs to agree. So that ensures you that whenever you make a decision, at least one member of that decision has to be part of every other decision. So you can do tricks like that. But yeah, you have to ensure that those quorums overlap. In consensus, you call that quorum intersection. And without that, it wouldn't be safe.

Verification with TLA+

00:43:55 • Audience (chat question read by the speaker)

How to prove with TLA+?

00:44:07 • Julian Haas

I'm not sure which part exactly you mean, like which one you would prove. I'm fairly certain that those two conditions that we define for the PRDT model, you could also verify them in TLA+. They are relatively simple to write down, I think. And you could also do that in TLA+. The downside of TLA+ usually is that in TLA+, you verify a model of your implementation. And then you have to make sure that this model actually corresponds to the code that you're running in the end. For example, if I'm programming in Scala and then I'm verifying in TLA+, then I have to make sure that what I'm doing in TLA+ actually corresponds to what I implemented in Scala or Rust or whatever. Yeah. But if you talk about just the theoretical framework and about the theorems, then I think those can easily be verified in TLA+.

00:45:20 • Julian Haas

Do we have any other questions? I don't see any so far.

Application-specific conditions and decisions

00:45:45 • Host

I have one question. We can write different protocol conditions or threshold conditions to define the conditions under which decisions should happen, right? Can you say a little bit more about the possibilities that PRDTs could support? That is my question.

00:46:31 • Julian Haas

Thanks for the question. So basically, I think the core part of our theory is that you have to make sure that first of all, the preconditions of your actions are strong enough to ensure these deterministic decisions. And then secondly, you have to make sure that those preconditions are also stable. So nothing anybody else could do could violate them. And the standard trick that we're doing is to tie this to a majority. So to simply say, okay, once we've seen a majority, there can't be a different majority. However, I believe that you could also define this differently if it kind of makes sense for your application and if you can still prove that it is going to be monotone in the end.

00:47:45 • Julian Haas

I guess one example would be, okay, I can also say, instead of doing the consensus algorithm, I write to a centralized database or I write to a blockchain and read the result. That is always stable because nobody else could influence it. But I guess what you were more hinting at is maybe defining more fine-grained decisions that can happen concurrently and without everybody, every device participating. And I think that hints at what I've said before. If your application has different kinds of decisions and they're somehow independent and maybe only some devices influence certain decisions and are unimportant for other decisions, then you can try to split that up a bit into different PRDTs who can then progress independently or maybe even adapt to stuff that happens on the application.

00:48:52 • Julian Haas

One example in the paper is a shopping application. Checkout, payment, and shipping are related and need ordering, but only for a particular customer. Different customers can shop concurrently, with their decisions progressing independently. The key is to make these dependencies explicit in the PRDT. I am not sure I am explaining this well, but the paper has an example where part of a PRDT is a map of predecessors. Each decision is tied to a set of previous decisions. Once those are complete, the new decision can be made. A decision with no predecessors can progress independently. You specify what each decision depends on, allowing more fine-grained modeling.

00:50:16 • Host

Thank you so much. Any other questions?

Parallel execution

00:50:29 • Audience

I think this language is also good at making the program parallelized. Because you can split a complex program into a lot of subprocesses, and you only coordinate those processes with these conditions. And otherwise, they can automatically run parallel.

00:51:12 • Julian Haas

Yes. These are immutable data types, and the decision procedure or PRDT tells you exactly where coordination is required. Other parts do not need it, which is useful.

CORDA and the broader research program

00:51:55 • Host

Can I also ask a little bit about another ongoing project, CORDA? Its title is very interesting: “Think Locally, Act Globally,” right?

00:52:23 • Julian Haas

Yes, that is the title. CORDA is an umbrella project bringing these components together, including LoRe and PRDTs. Its idea is to design a programming model based on local reasoning. You first write an application that runs locally and makes local decisions, but it is also local-first: what you do locally eventually reaches other devices and produces global outcomes. CORDA asks how we can do that safely.

00:53:24 • Julian Haas

How can we make sure that decisions are correct? And a colleague of mine is working on the access control part, for example. This is also quite an important bit. In a decentralized setting, how do I enforce that only some of my peers can read certain parts of the data and others maybe cannot? So yeah, I guess that's one part of CORDA that I haven't really talked about today. Yeah. But we are working on that part and hopefully also have a solution soon-ish. Yeah.

00:54:05 • Host

It is a larger project to encompass all these things.

00:54:14 • Julian Haas

Yeah, exactly.

Closing remarks

00:54:31 • Host

If there are no more questions, we can finish here. Thank you so much, Julian.

00:54:41 • Julian Haas

Thanks for your attention. This was interesting. Let's stay in touch about the blog post and recording, and good luck with the remaining talks and local-first week.

00:54:58 • Host

Thank you so much. We will send you the materials and keep in touch.

00:55:04 • Julian Haas

Thank you. Bye, everyone, and thanks for your questions.