My Bachelorthesis title

Bachelorarbeit von Philipp Hinz
Tag der Einreichung: 21. Januar 2023

1. Gutachten: Gutachter 1
2. Gutachten: Gutachter 2
3. Gutachten: noch einer
4. Gutachten: falls das immernoch nicht reicht

Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Institut
Arbeitsgruppe

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 und § 23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Philipp Hinz, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 21. Januar 2023

P. Hinz

Inhaltsverzeichnis

1 Introduction

1.1 Case Study

2 Background and Related Work

2.1 Local-First

2.2 CvRDT and CmRDT

2.3 Technical Stack

3 Extendable Commutative Replicable Data Types

In this chapter we will introduce the concept of Extendable Commutative Replicable Data Types (ECmRDT) and present our implementation of this concept.

3.1 Motivation

CvRDTs and CmRDTs do not care about authentication and authorization. This is usally not a problem in trusted environments like a cluster of servers. But used in client running applications (local-first applications) we usally can not trust our clients. Therefore we need to add authentication and authorization to our data types.

Additionally local-first applications require end-to-end encryption if we want to store our data on a server. This is usally difficult to combine with authentication and authorization.

Our here proposed solution is specially designed for a single server, but the concepts can be easily applied to peer-to-peer applications. We explore the use in peer-to-peer applications a bit in the future work section.

Initially we tried to design a CRDT specifically for authentication and authorization. But we found out that a more abstract approach is more flexible and easier to implement. Therefore we designed a new data type called Extendable Commutative Replicable Data Type (ECmRDT) with authentication and authorization as an extension.

3.2 Overview

We base our `ECmRDT` on CmRDTs and therefore make use of event sourcing. Because we design our `ECmRDT` to be used in server/client applications we only use direct event sourcing in storing the `Events` on the server. On the client we use a more classic approach and only store the last state of the data type and all pending events to be sent to the server. Incoming events are applied to the last state to get the new state.

The core feature of our `ECmRDT` is the concept of extensions. Extensions are a way to add functionality by mutation of state through events or validation of events.

3.2.1 Concepts

ReplicaId

The `ReplicaId` is a unique identifier for a replica / the user. The `ReplicaId` is in our case a public key. We will go into detail why we use a public key as `ReplicaId` in our chapter about authentication and authentication.

```
case class ReplicaId(
  val publicKey: BinaryData
)
```

AggregateId

The `AggregateId` is a unique identifier for an `ECmRDT`. The `AggregateId` is a combination of a `ReplicaId` and some random bytes. The `ReplicaId` is the `ownerReplicaId` of the Aggregate. The random bytes are used to prevent collisions of `AggregateId`s.

```
case class AggregateId(
  val replicaId: ReplicaId,
  val randomBytes: BinaryData
)
```

Effect

An `Effect` is a function that takes a `state`, an context, an `MetaContext` and returns a new `state` or an `RatableError` in future. By returning an `RatableError` we can verify the `Event` together with the given parameters. The effect is an asynchronous operation because we need to use cryptograpic operations to verify the `Event` which are usally implemented asynchronously.

```
type Effect[A, C] = (A, C, MetaContext) => EitherT[Future, RatableError, A]
```

Event

If the user wants to take any actions on the `ECmRDT`, he has to create an event. Every event is associated with a context. An `Event` is applied to the `ECmRDT` by converting it into an `Effect`.

```
trait Event[A, C]:
  def asEffect: Effect[A, C]
```

MetaContext

The `MetaContext` is a context that is passed to the `ECmRDT` when applying an `Event`. It contains information about the `ownerReplicaId` and the `aggregateId`.

The need results from the fact, that the initialization of `ECmRDT`s also happens through an `Event`. While processing the first `Event` we do not have any `state` yet. Therefore we could normally not validate the `Event`. The `MetaContext` is mainly used to validate the first `Event` by checking if the `Event` was created by the `ownerReplicaId`.

```scala
case class MetaContext(
  val aggregateId: AggregateId,
  val ownerReplicaId: ReplicaId,
)
```

ECmRDTEventWrapper

The `ECmRDTEventWrapper` is a wrapper for an `Event` and a context. It is used to bundle the `Event` with the context and the `clock` of the `ECmRDT` when the `Event` was created. The `clock` is used to prevent duplications of `Events`.

```scala
case class ECmRDTEventWrapper[A, C, +E <: Event[A, C]](
  val time: Long,
  val event: E,
  val context: C,
)
```

Context

The context contains meta information about a single `Event` but is the same for all `Events` of a single `ECmRDT` implementation. The idea is to compose the context out of multiple traits.

An core context trait is the `IdentityContext`. This context trait is always required and contains the `ReplicaId` of the creator of this `Event`. The validation of this identity is not done by the `ECmRDT` but must be done externally. More on this in the section about authentication and authorization.

```scala
trait IdentityContext:
  def replicaId: ReplicaId
```

ECmRDT

Our core concept is the `ECmRDT`. The `ECmRDT` consists of a `state` and a `clock`. The `state` is the actual data of the `ECmRDT`. The `clock` is a `VectorClock` to prevent duplications of `Events`. Our `ECmRDT` does not handle distribution of `Events` nor storing pending `Events`. This has to be done by the user of the `ECmRDT`.

Our `ECmRDT` supports two operations. One to `prepare` an `Event` and one to apply an prepared event called `ECmRDTEventWrapper`. The `prepare` operation is used to aquire additional information from the `ECmRDT`, specifically the `clock` and bundle the event with an context into one `ECmRDTEventWrapper`. The `apply` operation is used to apply the `Event` to the `ECmRDT`.

```scala
case class ECmRDT[A, C <: IdentityContext, E <: Event[A, C]](
  val state: A,
  val clock: VectorClock = VectorClock(Map.empty)
):
  def prepare(
    event: E, context: C
  )(
    using effectPipeline: EffectPipeline[A, C]
  ): ECmRDTEventWrapper[A, C, E] = ...

  def effect(
```

```
    wrapper: ECmRDTEventWrapper[A, C, E], meta: MetaContext
  )(
    using effectPipeline: EffectPipeline[A, C]
  ): EitherT[Future, RatableError, ECmRDT[A, C, E]] = ...
```

EffectPipeline

The `EffectPipeline` is a simple structure that converts an existing `Effect` into a new `Effect`. It is the core piece in implementing extensions. The `EffectPipeline` is implicitly specified by a `ECmRDT` implementation and is used to implement additional functionality like logging, validation or mutation. Functionality provided by the `EffectPipeline` is applied to all `Events` of a `ECmRDT`.

```
trait EffectPipeline[A, C]:
  def apply(effect: Effect[A, C]): Effect[A, C]
```

3.2.2 Usage and Example

The usage of an `ECmRDT` is generally simple. The user has to specify an State, a Context, some `Events` and a `EffectPipeline`.

When the user wants to apply an `Event` with an Context, he first has to prepare it. The preparation will add an time to the `Event` and bundle it with the Context. The user can then apply the prepared `Event` to the `ECmRDT`.

To apply the prepared event it is first converted into an `Effect`. This `Effect` is then converted into a new `Effect` by the `EffectPipeline`. With all provided information the `ECmRDT` can then convert its previous `state` using the `Effect` into a new `state`. The `state` with an updated `VectorClock` is then returned to the user.

An minimal example of an `ECmRDT` implementation is shown in the following listing. The `ECmRDT` is a simple counter that can be incremented.

```
case class CounterState(value: Int = 0)

case class CounterContext(replicaId: ReplicaId) extends IdentityContext

sealed trait CounterEvent extends Event[CounterState, CounterContext]

case class IncrementEvent() extends CounterEvent:
  def asEffect: Effect[CounterState, CounterContext] =
    (state, context, meta) => state.copy(value = state.value + 1)

// Somewhere in user code
val counter = ECmRDT[CounterState, CounterContext, CounterEvent](CounterState())
```

3.2.3 Extensions

3.3 ECmRDT Example

3.4 Conclusion

4 Ratable

4.1 Architecture

4.1.1 Core

4.1.2 Functions

4.1.3 Webapp

4.2 Implementation

4.3 Evaluation

5 Future Work

5.1 Ratable

5.2 ECmRDT