My Bachelorthesis title

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

P. Hinz

Inhaltsverzeichnis

1 Introduction

There are crdt and local first application

Why we need authorization in modern applications

CRDTs do not allow authentication

Propose ECmRDT to enable authentication and authorization

Additionally ECmRDT will enable us to create a variaty of extensions

Working on a case study including ECmRDT

case study will need a new type of architecture

introduce concepts how local first applications can be build

Enabled us to implement new use cases not possible previously

2 Case Study

Our case study is named Ratable. With Ratable users can create ratable objects and let a group of people rate them. Ratable is an application, users can interact with on their mobile devices or desktop computers.

The core idea contrary to usual rating services is that not everyone can rate a Ratable. With a newly created Ratable two links are provided. A rate and a view link. Only people with the rate link can rate and view the ratable.

Ratable is implemented as a cloud-native local-first application. Ratable allows users to use the application without an internet connection and provides updates/changes in real-time. This provides for a very fluent usage of Ratable because we do not need to have any loading screens. The only time we need to load something is when we access some Ratable for the first time. Changes to Ratables are done instantly and synchronized when possible.

Especially important for local-first applications is the feature that all trust is given to individual users instead of a server. This means that the server for Ratable plays only a secondary role and does not do much more than distributing state. All authorization and authentication are done by the clients themselves. This allows us to enable end-to-end encryption of the state.

3 Extendable CmRDT

In this chapter, we will look into what ECmRDT's are, what they are trying to do and how they are used to achieve the goal.

3.1 Overview

Extendable CmRDT's are normal CmRDT's with the ability to build extensions for them. Here CmRDT's are operation based CRDT's. Extensions are able verify and mutate events and state. They can also interact with other extensions.

With ECmRDT's we want to enable developers to create extensions that can be enabled for data types to easily extend them with additional functionality. This functionality can be for example something like verification of replicaId's in events, change of some variables in state before or after applying an event, migrations of events or authentication and authorization of events. Especially the last example is the primary reason for the usage of ECmRDT's. Contrary to normal CRDT's they allow us to build in security.

Generally ECmRDT's work with events and state. In our design we do only use event sourcing at server side. That means that an ECmRDT itself only contains state. Users can then create an event and apply it to the ECmRDT and distribute it to other clients. Because events are signed, they can not be forged. How an event is applied to state in detail depends on and is defined in the domain logic. Extensions now come into place directly before applying the event to the state. The extensions are defined in something like a pipeline and can either forward the to event to the next extension or fail. Thereby events can either changes the data coming in or coming out.

Lets say we want to implement a counter that can only be incremented by the owner. We would first define the state as a integer counter and one event that increments the state counter by one. The context and generally how these concepts relate to each other will be covered in the next section.

```
case class Counter(
  val value: Int,
)

case class CounterContext(
  val replicaId: ReplicaId,
)  extends IdentityContext

sealed trait CounterEvent extends Event[Counter, CounterContext]

case class AddCounterEvent() extends CounterEvent:
  def asEffect =
    (state, context, meta) => EitherT.pure(
      state.copy(value = state.value + 1)
    )
```

Then we would enable an build-in extensions that only lets events pass when they are send by the owner of the ratable.

```
object Counter:
  given EffectPipeline[Counter, CounterContext] = EffectPipeline(
    SingleOwnerEffectPipeline()
  )
```

Using only a few lines of code we have now a secure incrementable counter that can only be incremented by the owner. This counter can now be used like the following. The example will first create a replicaId and an initial state. With both we can create a event and apply it to our created state. In the end we print the state before and after applying our event.

```
def main(using Crypt) =
  for
    // Step 1: Create replicaId
    replicaId <- EitherT.liftF(PrivateReplicaId())

    // Step 2: Create initial state.
    counter = ECmRDT[Counter, CounterContext,
       CounterEvent](Counter(0))

    // Step 3: Create event.
    eventPrepared = counter.prepare(
      AddCounterEvent(),
      CounterContext(replicaId)
    )

    // Step 4: Verify and advance state.
    newCounter <- counter.effect(eventPrepared, MetaContext(
      AggregateId.singleton(replicaId),
      replicaId
    ))

  yield
    println(s"Old counter: ${counter.state.value}")    // 0
    println(s"New counter: ${newCounter.state.value}") // 1
```

How exactly the ECmRDT's and the SingleOwnerEffectPipeline work will be in the next section.

3.2 Concepts

In this section, we will introduce concepts of our ECmRDT with definitions, code examples and also explain how certain relate to each other.

Aggregate

The definition of aggregate I will use here is Än AGGREGATE is a cluster of associated objects that we treat as a unit for the purpose of data changes. Each AGGREGATE has a root and a boundary. The boundary defines what is inside the AGGREGATE. The root is a single, specific ENTITY contained in the AGGREGATE."from Evans DDD.

In general it can be understood as a Domain concepts where only one aggregate at a time should be changed by a use case. Here we will define one ECmRDT per aggregate.

ReplicaId

The ReplicaId is a unique identifier for a user. The ReplicaId consists of a public/private key pair. With this we can sign and verify events send by a user. Because the private key only exists in the corresponding replica as a privateReplicaId, the ReplicaId as a data structure only contains the public key for identification. More about the usage of the ReplicaId is covered in the chapter about authentication and authorization.

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

AggregateId

The AggregateId is a unique identifier for an ECmRDT/aggregate. The AggregateId is a combination of a ReplicaId and random bytes. The ReplicaId inside the aggregateId represents the owner of the aggregate. We need to store the ReplicaId to prevent other replica's of creating aggreagtes with the same AggregateId malicously. The random bytes are used to prevent collisions of AggregateIds within a group of aggregates of a replica.

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

Effect

An Event can be converted to an Effect to apply the Event to the state. Generally 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 abort the Event and therefore verify the Event together with the given parameters if they are valid. The effect is an asynchronous operation because we sometimes need to use cryptograpic operations to verify the Event which are implemented in webbrowsers asynchronously.

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

Event

Events are created by users to change the state. Usally Events are caused directly by user actions. Events are implicitly associated with a context. Thereby Events contain information specific to this event and the context contains information that is contained in every event. Events can be converted into Effects to be late be used to advance the state.

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

MetaContext

MetaContexts contain information required for ECmRDTs but that are not stored directly in the ECmRDT. Currently it contains information about the AggregateId of the ECmRDT and the ReplicaId of the aggregate owner. The reason why dont store the aggregate owner inside the ECmRDT is because we can already get it implicitly from the aggregateId.

The need for MetaContexts comes from the problem on how to initialize ECmRDTs through an initial events and especially how to verify initial events. At creation time when processing the first event the state is yet empty/default initialized. Therefore we would normally not be able to validate the event through the provided state. Using the MetaContext or to be more precise the owner replicaId inside the MetaContext we can enforce the rule to only allow initial events to be sent by the owner of the aggregate.

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

ECmRDTEventWrapper

Events are implicitly associated with Contexts. This association becomes explicit through an ECmRDTEventWrapper. The reason why we normally only associate Events implicitly is because an ECmRDTEventWrapper contains additional information that is aquired by preparing an Event and Context through an ECmRDT. Only after preparation and conversion to an ECmRDTEventWrapper can it be used to update an ECmRDT.

The currently primary information packed additionally with an ECmRDTEventWrapper is time. An ECmRDT uses a VectorClock to prevent Event duplicates. The time from the VectorClock is then stored inside the Event to associate it with the time.

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

Context

Events only contain information specific to the aggregate and the information are neither accessible by ECmRDT nor Extensions. Therefore we use Contexts to provide common information stored with events that allow us to use them in ECmRDTs and Extensions.

A good example is the IdentityContext containing an replicaId. It is used in ECmRDTs to update the VectorClock of the replica sending the Event and also used when filtering Events for Authentication and Authorization.

It should be noted that validation of the IdentityContext itself (if the replicaId is actually the sender) is done outside of the ECmRDT. The validation happens through an signature added to events which can then be verified by the public key inside the replicaId.

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

ECmRDT

The core concept is the ECmRDT. The ECmRDT consists of a state and a clock. The state contains the actual data of the aggregate. The clock is a VectorClock to prevent duplications of Events. Our ECmRDT does neither handle distribution of Events nor storing pending Events nor validation of identities (see Context section). 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. 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 while also advancing the vector clock.

```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

Extensions are implemented by transforming an Effect to a new Effect. The function that transforms this Effect is called EffectPipeline. EffectPipelines are specified by aggregates to enable extensions to add functionality like logging, validation, mutation or more. Important is that the functionality provided by EffectPipelines is will be used for all Events of an ECmRDT.

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

3.3 Extensions

Extensions allow developers to define functionality used in the whole aggregate. Extensions can enable things like event validation, state mutation, replace/adjust State, Context or MetaContext parameters and more. Additionally, Extensions also allow for easier code reuse between aggregates and Extensions can build on existing Extensions. An Extension consists of a EffectPipeline and optionally of a Context and State.

An EffectPipeline itself is only a function transforming an Effect which itself is a function. The idea now is to intercept the Effect function call and execute custom logic before or after the call. If an aggregate wants to define multiple Extensions we can chain these EffectPipelines into a call chain which results in a single resulting functions which itself is again a EffectPipeline. This does in practice look like the following. Here we log the replicaId of the sender.

```
object TestEffectPipeline:
  def apply[A, C <: IdentityContext](): EffectPipeline[A, C] =
    effect => (state, context, meta) =>
      for
        newState <- effect(state, context, meta)
      yield
        println(s"Sender replicaId: \${context.replicaId}")
        newState
```

For the most important feature is that an Effect can also fail. Because of that we are able to shortcircuit the Extension call chain when an error happens or some validation fails. One example for this is the SingleOwnerEffectPipeline. Here we validate that the Context replicaId (event sender replicaId) is the same as the meta replicaId (aggregate owner replicaId). If the check fails we do not continue in calling the given Effect. To make development easier because verification Extensions are common we are using a verifyEffectPipeline helper function. This helper function only expects a list of errors as return. If the list is empty we succeeded and continue the Effect call.

```
object SingleOwnerEffectPipeline:
  def apply[A, C <: IdentityContext](): EffectPipeline[A, C] =
    verifyEffectPipeline[A, C]((state, context, meta) => List(
      Option.when(meta.ownerReplicaId != context.replicaId)(
        RatableError(s"Replica ${context.replicaId} is not the owner
           ${meta.ownerReplicaId} of this state.")
      )
    ))

// Helper to build a synchronous verify only effect pipelines
def verifyEffectPipeline[A, C](
  f: (A, C, MetaContext) => List[Option[RatableError]]
): Effect[A, C] => Effect[A, C] =
  verifyEffectPipelineFuture((a, c, m) => f(a, c,
     m).map(OptionT.fromOption(_)))

// Helper to build a asynchronous verify only effect pipelines
def verifyEffectPipelineFuture[A, C](
  f: (A, C, MetaContext) => List[OptionT[Future, RatableError]]
): Effect[A, C] => Effect[A, C] =
  (effect) =>
```

```
    (state, context, meta) =>
      for
        _ <- f(state, context, meta).map(_.toLeft(())).sequence
        newState <- effect(state, context, meta)

      yield
        newState
```

In the previous example we have used the IdentityContext by specifying type bounds. Additionally to defining EffectPipelines, Extensions can also define their own Context as well as their own state. When an aggregate wants to use an Extension it has to include the Extension Context and State into its own Context and State. Because of the type bounds we would get a type error if forgotten. How custom Context and State can be used to create more flexible functions will be shown in the section about Authentication and Authorization.

3.4 Authentication and authorization

In this section we will look into what assumptions are met and how a Extension for ECmRDTs is created to enable authentication and authorization. Our goal is that aggregates can define a list of permissions. Certain events may require permissions conditionally as specified by the aggregate. User then need to add proofs for the permissions so that every other user can verify it.

Proofs are implemented as signatures of a replicaId (the user who wants to use the event) using asymmetric encryption. This also means, that a proof only allows a single user to use an event. For other users new proofs have to be created. The public key of a proof is called a claim and the private key is called a prover. Claims are stored publicly in the aggregate state so that every user is able to verify incoming events.

```
case class Claim[I](
  val publicKey: BinaryData,
  val id: I, // I is a enum value representing the permission
)

case class ClaimProof[C](
  val proof: BinaryData,
  val id: C,
):
  def verify(claim: Claim[C], replicaId: ReplicaId) // ...

case class ClaimProver[ID](
  val privateKey: BinaryData,
  val id: ID
):
  def prove(replicaId: ReplicaId) // ...
```

Claims and provers are created using the initial event but the provers are usually not stored inside the aggregate. The only exception is an additional extension called ClaimBehindPassword which tries to reduce the key size that has to be transmitted. The idea is to store the private keys encrypted symmetrically publicly in the aggregate with a shorter password. Now everyone who has the password can create their own proofs.

For this whole system to work we need to make sure that every event telling it was send by an replicaId is actually send by this replicaId. This is important because when validating proofs we verify a signature containg the sender replicaId. If one would be able to spoof the sender replicaId we could use an existing proof send by a different user (that is possible because proofs are public by default because everyone needs to be able to verify them) and spoof their replicaId. Users would then not be able to verify the event correctly.

This issue is the main reason why the replicaId is a public key where only the user itself does know the private key. Events are signed using the private key so that everyone receiving events from a replicaId can be sure that the replicaId used in the event are legitimit.

Extension

The Extension is implemented using a Context, a State and a EffectPipeline. In the Context we store all the proofs containg the signatures added by the user using an event and in the state we store all claims containing the public key to verify the claims. ClaimProvers are not directly part of the extension and have to be handled by the developers using the Extension.

```
trait AsymPermissionContextExtension[I]:
  def proofs: List[ClaimProof[I]]

  def verifyPermission[A](permission: I): EitherT[Future,
      RatableError, Unit] = EitherT.cond[Future](
    proofs.exists(_.id == permission), (),
    RatableError(s"Missing permission \$permission.")
  )

trait AsymPermissionStateExtension[I]:
  def claims: List[Claim[I]]
```

The State and Context now are used inside the EffectPipeline. The idea is to verify all proofs contained in the Event Context, no matter if they are actually required by the Event. The Extension itself does not even need to know what proofs are required, it only makes sure that all provided proofs are in fact valid. The Effect of the Event itself later can call the verifyPermission method of the Context to easily make sure that alle required permissions are proofed by the context.

So in the EffectPipeline we first have to find the Claim for the respective Proof and then make sure it is valid. If either the claim does not exist or the proof is invalid we cancel the pipeline call chain by returning an error. For this we are using the verifyEffectPipelineFuture helper.

```
object AsymPermissionEffectPipeline:
  def apply[
    A <: AsymPermissionStateExtension[I],
    I,
    C <: AsymPermissionContextExtension[I] with IdentityContext
  ](using Crypt): EffectPipeline[A, C] =
    verifyEffectPipelineFuture[A, C]((state, context, meta) =>
      for
        proof <- context.proofs
      yield
        state.claims.find(_.id == proof.id) match
          case Some(claim) =>
            OptionT(proof
              .verify(claim, context.replicaId)
              .map(Option.unless(_)(RatableError("Proof is
                  invalid.")))
            )

          case None =>
            OptionT.pure(RatableError("Claim does not exist."))
    )
```

3.5 Integration into ratable

How the ratable domain is designed using ECmRDT

Specific benifits of ECmRDT in Ratable

4 Architecture

Explain three projects composing ratable

Why we decided against peer to peer

Additional services used shown in a diagram to get an overall view

4.1 Project Structure

Project consists of layers

Device layer for hardware abstraction

Application layer containing use cases and additionally ui

Structure enabling testing of each use case by mocking device layer only

4.2 State managment

What our state managment can do

How users interact with our state

Inner and outer aggregate abstractions

How state is managed between abstractions and outer access

How side effects for distribution and co work

5 Future

Real peer to peer systems using ecmrdt’s

Why do we have events and contexts. Couldnt we just say Event extends Context and explicitly associate them from the beginning?

Include event signing in general ECmRDT architecture.