My Bachelorthesis title

Bachelorarbeit von Philipp Hinz
Tag der Einreichung: 19. 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, 19. Februar 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

In this chapter, we will introduce the concepts of our `ECmRDT` with definitions and code examples.

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`.

```
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`.

```
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.

```
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`.

```
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)

// Empty pipeline because we don't need any extensions
object CounterState:
  given EffectPipeline[Counter, CounterContext] = EffectPipeline()

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

The most important new feature of ECmRDTs and the reason for Contexts to exist is the ability to extend the functionality of an ECmRDT. The idea is, that an extension can define a new Context trait or new State trait and then implement the `EffectPipeline`. While defining the `EffectPipeline` the extension can specify its required traits or traits from other extensions as type bounds. This allows to build extensions onto each other. The EffectPipeline can then use the traits to implement either validation or also mutation of the state.

An minimal extension that counts the number of events could looks like this.

```
trait CountEventsState:
  // We define attributes as methods. This later allows
  // us to override them as attributes
  def count: Int

object CountEventsEffectPipeline:
  def apply[A <: CountEventsState, C](): EffectPipeline[A, C] =
    (effect) => (state, context, meta) =>
      for
        newState <- effect(state, context, meta)
      yield
        newState.copy(count = newState.count + 1)
```

We could define an additional extension that logs all events.

```
object LoggingEffectPipeline:
  def apply[A <: CountEventsState, C](): EffectPipeline[A, C] =
    (effect) => (state, context, meta) =>
      for
        newState <- effect(state, context, meta)
      yield
        println(s"Event ${meta.event} applied by ${context.replicaId}")
        newState
```

In our earlier example we could now use both extensions like this.

```
case class CounterState(
  value: Int,
  count: Int // We can now easily override as attribute
) extends CountEventsState:

object CounterState:
  given EffectPipeline[Counter, CounterContext] = EffectPipeline(
    // We can now use both extensions
    // They will later be used implicitly by the ECmRDT
    CountEventsEffectPipeline(),
    LoggingEffectPipeline()
  )
```

3.3 Authentication and Authorization

Now that we have proposed the design of our ECmRDT we can come back to the problem of authentication and authorization. Our primary goal is to be able to check for an event sent by an user if the user is allowed use the event.

We solved this issue by defining new concepts called `Claim`, `ClaimProof` and `ClaimProver` using asymmetric cryptography. For every `Claim` there always exist exactly one `ClaimProver` and multiple `ClaimProof`s. The `Claim` is a public key that is used to verify the signature in the `ClaimProof`. An `ClaimProof` is a signature of the `ReplicaId` by the sender signed with a private key contained in the `ClaimProver`. Because `ClaimProof` only contains an signature, the `Claim` and `ClaimProof` are considered to be public information.

The concepts are used in the following way. All `Claim`s are publicly saved in the `state` of the `ECmRDT`. Every event contains a set of `ClaimProof`s. Now every event will define in advance which `Claim`s are required to be present in the `ClaimProof`s. An EffectPipeline can then check if all provided `ClaimProof`s are valid.

Claim vs ReplicaId

This approach lets one thing open. How can we ensure that the `ReplicaId` in the `ClaimProof` is the same as the `ReplicaId` of the sender?

This issue is solved by the definition of the `ReplicaId` itself. An `ReplicaId` is an asymmetric public key. When a user wants to send an event he first creates a signature of the event with his private key. This signature is then sent with the event. The `ReplicaId` is then used to verify the signature. Therefore everyone can verify that the event was sent by the user with the `ReplicaId` and that the `ReplicaId` is the same as the one in the `ClaimProof`.

One might ask why we don't just save a set of `ReplicaId`s in the `state` instead of `Claim`s. This is actually the recommended way if the use case allows it. The reason why we use `Claim`s is that we want users to be able to aquire authorization at any time using a piece of information. We actually later show a way to reduce the amount of information needed to a short password. The approach of using `ReplicaId`s is more suitable for use cases where the authorization is granted by users.

Read Access

The here proposed approach is very flexible. While it would be possible to restrict read access we did not need it in our use case. Therefore we did not implement it but will show how it could be done in the future work section.

3.3.1 Example

We want to extend our previously defined `Counter` with the ability to restrict access to the `IncrementEvent`. For this we have define an `CounterClaimEnum` to idenify `Claim`s, extend the `CounterState` to contain a set of `Claim`s and extend the `CounterContext` to contain a set of `ClaimProof`s. Additionally we have to install a new `EffectPipeline` that checks if the `ClaimProof`s are valid and check for the needed claims in the event.

```
enum CounterClaimEnum:
  case Increment

case class CounterState(
  value: Int,
  count: Int,
  claims: Set[Claim[CounterClaimEnum]]
) extends CountEventsState

case class CounterContext(
  replicaId: ReplicaId,
  claimProofs: Set[ClaimProof[CounterClaimEnum]]
) extends CountEventsContext

object CounterState:
  given EffectPipeline[Counter, CounterContext] = EffectPipeline(
    CountEventsEffectPipeline(),
    LoggingEffectPipeline(),
    // We install a new EffectPipeline that checks if the ClaimProofs are valid
    AsymPermissionEffectPipeline[CounterState, CounterClaimEnum, CounterContext]
  )

sealed trait CounterEvent extends Event[CounterState, CounterContext]

case class IncrementEvent() extends CounterEvent:
  def asEffect: Effect[CounterState, CounterContext] =
    (state, context, meta) =>
      for
        newState <- context.verifyClaim(CounterClaimEnum.Increment)
      yield
        newState.copy(count = newState.count + 1)
```

This definition could now be used in the following way.

```
def main(using Crypt) =
  for
    replicaId <- EitherT.liftF(PrivateReplicaId())

    // Step 1: Create claims and claimProvers.
    claimsPair <- EitherT.liftF(
      Claim.create(List(CounterClaimEnum.Increment))
    )

    (claims, provers) = claimsPair

    proof <- EitherT.liftF(
      provers.head.prove(replicaId)
    )

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

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

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

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

3.4 ECmRDT Example

3.5 Conclusion

4 Ratable

In this chapter we will look into how Ratable is implemented and how concepts from the previous chapter are used.

4.1 Architecture

Ratable consists of three main parts. The Core, the Functions and the Webapp. The Webapp is the interface to the user. Using ECmRDTs, our Webapp contains most of our application logic and our state. The Functions are the backend of our Webapp. They are responsible for storing and passing Events to and from the Webapp. The Core is shared between the Webapp and the Functions. It contains the ECmRDT library and all of the ECmRDT aggregates.

The communication between our Webapp and our Functions works in two ways. Since we want to pass Events to the Functions in realtime, we have to use a websocket connection. Additionally for synchronous requests we use a REST API. The communication protocol uses Protobufs. We are not using JSON because we use a lot of 'oneof' relationships in our data and Scala 3 json libraries do not satisfactorily support this.

The two main projects Functions and Webapp are structured similarly. The most central aspect to our design is seperation into multiple layers of modules and heavy usage of dependency injection to allow for easy testing and modularization.

Both projects consist of a root layer, an application layer, a device layer and multiple other layers. This structure can be thought of similarly to a tree as shown in figure 4.1. Every layer contains services and other modules. The device layer abstracts over the underlying platform. This allows us to mock it and test the rest of the application. The root layer contains core services and modules like logging or configuration. The application layer is the core of both projects containing important logic specific two each project.

Dependency Injection

We implemented dependency injection by structural typing. This allows us to inject services in the following way.

```
class MyService(services: {
  val myDependency: MyDependency
}):
  def doSomething() = ???

class MyDependency(services: {})

class Services:
```

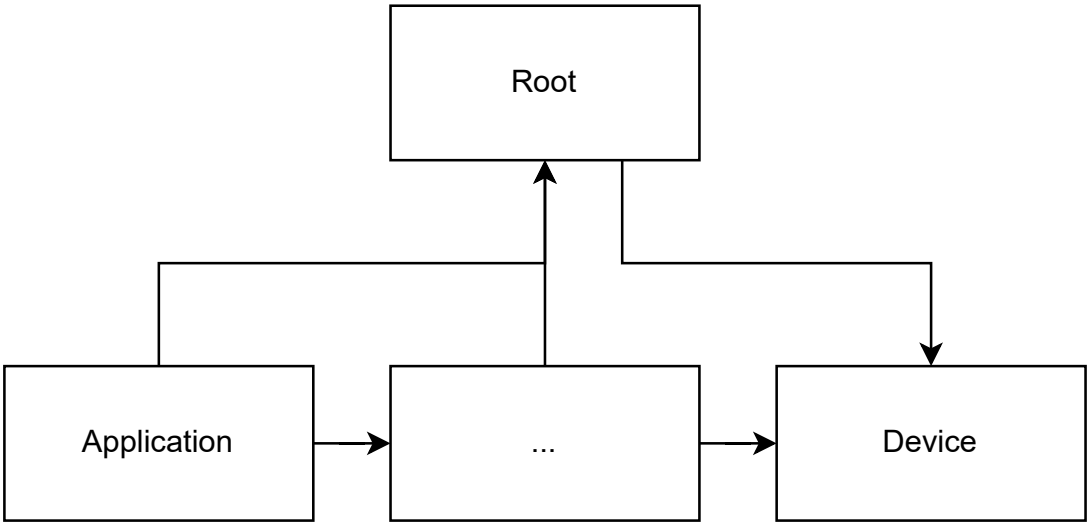

Abbildung 4.1: Architecture

```
  lazy val myService = new MyService(this)
  lazy val myDependency = new MyDependency(this)

// ...

val services = new Services()
services.myService.doSomething()
```

4.2 Core

The Core contains three big parts. The ECmRDT library and some additionally needed definitions, the ECmRDT aggregates containing the actual domain logic and the protobuf definitions defining the communication protocol between the Webapp and the Functions.

The actual domain is for our use case very small and will show how the ECmRDT library can be used to implement a simple application.

Domain - Ratable

The state is pretty simple. We have a Ratable containing a title, a list of categories and a list of ratings. Additionally for our extensions we define claims and claimsBehindPassword. Note here, that the State can and should already contain domain logic.

```
case class Category(
  val title: String
)

case class Rating(
```

```
  val ratingForCategory: Map[Int, Int]
)

enum RatableClaims extends Enum[RatableClaims]:
  case CanRate

case class Ratable(
  val claims: List[Claim[RatableClaims]],
  val claimsBehindPassword: Map[RatableClaims, BinaryDataWithIV],

  val title: String,
  val categories: Map[Int, Category],
  val ratings: Map[ReplicaId, Rating]
)
extends AsymPermissionStateExtension[RatableClaims],
   ClaimByPasswordStateExtension[RatableClaims]:
  def rate(replicaId: ReplicaId, ratingForCategory: Map[Int, Int]): Ratable = ???

  def categoriesWithRating: Map[Int, (Category, Int)] = ???


object Ratable:
  given (using Crypt): EffectPipeline[Ratable, RatableContext] = EffectPipeline(
    AsymPermissionEffectPipeline[Ratable, RatableClaims, RatableContext]
  )

  given InitialECmRDT[Ratable] = InitialECmRDT(Ratable(
    List(), Map(), "missing-title", Map(0 -> Category("missing-categories")), Map()
  ))


case class RatableContext(
  val replicaId: ReplicaId,
  val proofs: List[ClaimProof[RatableClaims]]
  )
  extends IdentityContext
    with AsymPermissionContextExtension[RatableClaims]


sealed trait RatableEvent extends Event[Ratable, RatableContext]

case class RateEvent(
  val ratingForCategory: Map[Int, Int]
) extends RatableEvent:
  def asEffect: Effect[Ratable, RatableContext] =
    (state, context, meta) =>
      for
        _ <- context.verifyClaim(RatableClaims.CanRate)
        _ <- EitherT.cond(ratingForCategory.size == state.categories.size, (),
          RatableError(s"Rating must contain ${state.categories.size} categories,
              but ${ratingForCategory.size} given"))
      yield
```

```
        state.rate(context.replicaId, ratingForCategory)


case class CreateRatableEvent(
  val canRate: Claim[RatableClaims],
  val canRateBehindPassword: BinaryDataWithIV,

  val title: String,
  val categories: List[String]
) extends RatableEvent:
  def asEffect: Effect[Ratable, RatableContext] =
    (state, context, meta) =>
      for
        _ <- EitherT.cond(meta.ownerReplicaId == context.replicaId, (),
          RatableError(s"Replica ${context.replicaId} is not the owner
              ${meta.ownerReplicaId} of this object."))

      yield
        Ratable(
          List(canRate),
          Map(RatableClaims.CanRate -> canRateBehindPassword),
          title,
          categories
            .zipWithIndex
            .map((title, index) => (index, Category(title)))
            .toMap,
          Map()
        )


case class CreateRatableResult(
  val event: CreateRatableEvent,
  val password: String
)

def createRatable(title: String, categories: List[String])(using crypt: Crypt):
    Future[CreateRatableResult] =
  val password = Random.alphanumeric.take(18).mkString

  for
    (canRateClaim, canRateProver) <- Claim.create(RatableClaims.CanRate)
    claimBehindPassword <- ClaimBehindPassword(canRateProver.privateKey.inner,
        password)

  yield
    CreateRatableResult(
      CreateRatableEvent(
        canRateClaim,
        claimBehindPassword,
        title,
        categories
      ),
      password
```

)

Protobuf

4.3 Functions

Our goal was to use similar technologies in front and backend. Additionally, our goal was to use cloud services as much as possible. Therefore, we decided to use the serverless approach provided by Azure Functions. This is especially useful because the cost of small projects is very low. Additionally, we can easily scale up if needed.

Azure functions provide a lot of different languages to implement them including JVM languages and Javascript. We decided to use Javascript because the JVM implementation does not support WebPubSubs yet.

It is possible with scalajs to compile Scala to multiple Javascript files but because of lack of documentation for scala 3 we decided to deploy everything in one single file. Because we still need to provide Azure Functions as multiple Javascript files, we decided to use a proxy approach. Each of the functions is a proxy to the main file. The main file now contains two gateways that handle the messages using a message handler for each message type.

The first gateway is the Http gateway. This gateway is used to handle Http requests. This is especially useful for synchronous requests where the user is waiting for a response like login or requesting data. The second gateway is the socket gateway. This gateway is used to handle messages from our WebPubSub send via WebSockets. Contrary to the Http gateway, this gateway is used for asynchronous messages. This is useful for messages like sending or receiving events in real time.

Because we need to use Javascript libraries to interact with our Azure services we abstract them away in our device layer. This allows us to test each of our message handlers without the need to use the Azure services.

ECmRDT

Different from our WebApp implementation, the state management in our functions is minimal. Because we only need to distribute the events between our clients, we only need to store the events and not act on them. Especially in an end-to-end encryption scenario, acting on the events would not even be fully possible.

4.4 Webapp

WebApp is the primary part of Ratable. It is our interface to Ratables while also containing and handling all logic. It can be thought of similarly to a Frontend combined with a Backend.

4.4.1 Application Layer

The application layer implements the user interface and interactions with the state.

We tried to approach the WebApp in a use-case-oriented approach. This allows us to easily write tests for each use case and test all scenarios through the whole application. The application layer is split into pages

and use-cases. Use cases are called directly by pages and are the only way to manipulate the state from the application layer. Additionally, we provide localization, routing and popups through services.

4.4.2 State Layer

The state layer provides access to the state, distributes it and stores it in the local database.

All state is accessible through a root layer service providing a simplified interface. The application state is bundled from multiple so called aggregate view repositories types as shown in the following listing.

```
case class ApplicationState(
  ratables: AggregateViewRepository[
    Ratable, RatableContext, RatableEvent
  ],
  library: AggregateViewRepository[
    RatableLibrary, RatableLibraryContext, RatableLibraryEvent
  ]
)
```

The goal of an aggregate view repository is to provide a repository for views of one aggregate type. In our case, a repository allows us to get, add and remove views of one aggregate type. It should be noted, getting a view does not only return the view to its current state but also provides all changes to the view in the future.

```
trait AggregateViewRepository[A : InitialECmRDT, C, E <: Event[A, C]]:
  def all: Future[Seq[(AggregateGid, A)]]

  def create(id: AggregateId): AggregateView[A, C, E]
  def get(id: AggregateId): EitherT[Future, RatableError, Option[AggregateView[A, C,
     E]]]
```

An aggregate is represented by views and facades. Views are provided publicly to the outside of the state layer and allow us to listen for changes and trigger events.

```
trait AggregateView[A, C, E <: Event[A, C]]:
  def effect(event: E, context: C)(using EffectPipeline[A, C]): EitherT[Future,
     RatableError, Unit]
  def listen: Signal[A]
```

Facades are used internally and allow broader access to the aggregate. Facades allow us to manipulate the ECmRDT of our aggregate without restrictions and listen to changes to the ECmRDT instead of the aggregate only. This is especially needed to implement the synchronization of the ECmRDT. It should be noted that the AggregateFacade is the place where the ECmRDT is stored.

```
class AggregateFacade[A, C <: IdentityContext, E <: Event[A, C]](
  private val initial: EventBufferContainer[A, C, E]
):
  private val variable = Var(initial)
```

```
  private var aggregateInFuture = Future.successful(initial)

  def listen: Signal[EventBufferContainer[A, C, E]] = variable

  def mutateTrivial(
    f: EventBufferContainer[A, C, E] => EventBufferContainer[A, C, E]
  ): EitherT[Future, RatableError, EventBufferContainer[A, C, E]] =
    mutate(aggregate => EitherT.rightT(f(aggregate)))

  def mutate(
    f: EventBufferContainer[A, C, E] =>
      EitherT[Future, RatableError, EventBufferContainer[A, C, E]]
  ): EitherT[Future, RatableError, EventBufferContainer[A, C, E]] = ???
```

One might wonder why the AggregateFacade works with EventBufferContainer and not ECmRDTs directly. This is because ECmRDT does not provide storage for events out of the box. This is needed to be able to distribute the events to the server in case of failure. Therefore, we created a wrapper for our ECmRDT that saves all events that are applied to it while also providing a method to acknowledge and remove all events that are successfully distributed.

4.5 Implementation

4.6 Evaluation

5 Future Work

5.1 Ratable

5.2 ECmRDT