My Bachelorthesis title

Bachelorarbeit von Philipp Hinz
Tag der Einreichung: 27. März 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, 27. März 2023

P. Hinz

Inhaltsverzeichnis

1 Introduction

Modern applications often rely on distributed systems that allow users to collaborate and share data across different devices and networks. One approach to achieve this is to use conflict-free replicated data types (CRDTs), which are data structures that can be replicated and modified concurrently without coordination. CRDTs enable local-first applications, which prioritize local availability and responsiveness over global consistency.

However, CRDTs also pose some challenges for application developers, especially when it comes to authentication and authorization. Authentication is the process of verifying the identity of a user or device, while authorization is the process of granting or denying access rights to resources based on predefined policies. CRDTs do not inherently support authentication or authorization mechanisms, which means that any replica can modify any part of the shared data without restrictions.

However, this also poses some challenges to ensuring the security and privacy of shared data. For example, how can a user verify that another user is who they claim to be? How can a user control who can access their data and what they can do with it? How can a user prevent unauthorized or malicious modifications of their data by other users or devices?

This thesis proposes a novel type of CRDTs called ECmRDTs (extendable conflict-free replicated data types), which allows us to write extensions enabling authentication and authorization in local-first applications. We use symmetric and assymetric encryption techniques to protect the confidentiality and integrity of shared data, as well as to enforce access control policies based on proofs provided by users.

To demonstrate the feasibility and applicability of ECmRDTs, this thesis presents a case study of a rating application called Ratable as its underlying data model. The case study also requires a new type of architecture that supports local-first principles and enables efficient synchronization of encrypted data among replicas. This thesis introduces several concepts and techniques that can help design and implement such an architecture.

1.1 CRDT

Conflict-free replicated data types (CRDTs) are data structures that can be replicated across multiple nodes in a distributed system, and can be updated concurrently without coordination or locking. CRDTs guarantee eventual consistency, meaning that all replicas will converge to the same state if no new updates are made. CRDTs are useful for applications that require high availability, low latency and tolerance to network partitions.

There are two main approaches to CRDTs: state-based (CvRDT) and operation-based (CmRDT). In state-based CRDTs, each replica maintains a full copy of the data structure, and updates are propagated by sending the entire state or a delta-state to other replicas. The merge function for state-based CRDTs must be commutative, associative and idempotent, ensuring that the order and number of merges do not affect the final state. In operation-based CRDTs, each replica maintains a partial copy of the data structure, and updates are propagated by sending only the operations that modify the state to other replicas. The delivery function for operation-based CRDTs must ensure causal order, meaning that an operation cannot be applied before its dependencies are satisfied. The operations for operation-based CRDTs must also be commutative, ensuring that the order of concurrent operations does not affect the final state.

In this thesis, we will use operation-based CRDTs to build extendable CRDTs (ECmRDTs), which are CRDTs that can dynamically incorporate new data types and operations without requiring a global agreement or a system restart. We will show how ECmRDTs can support various use cases such as collaborative editing, online gaming and social networking.

1.2 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.

Cloud-native and local-first are two design principles that enable Ratable to deliver a fast and reliable user experience. Cloud-native means that Ratable is built and deployed using cloud computing services, such as storage, databases, and serverless functions. This allows Ratable to scale up or down according to the demand, and to benefit from the security and availability of the cloud providers. Local-first means that Ratable stores and processes data on the user's device, rather than on a remote server. This allows Ratable to work offline, and to sync data with the cloud only when needed. This reduces the network latency and bandwidth consumption, and enhances the privacy and autonomy of the user. By combining cloud-native and local-first, Ratable achieves a balance between performance and functionality, and offers a smooth and seamless user experience.

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.

2 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 their goals.

2.1 Overview

CmRDTs are operation-based CRDTs that allow concurrent updates on replicated data without coordination. However, CmRDTs have some limitations, such as the lack of security and the difficulty of adding new features. To overcome these limitations, we introduce ECmRDTs, which are extendable CmRDTs that support extensions for verification, mutation, interaction, and authentication of events and state.

An event is an operation that modifies the state of an ECmRDT. A state is the current value of an ECmRDT. Events are signed by the clients that generate them and sent to a server that stores them. The server acts as an event source, which means that it provides events to clients on demand. Clients can request events from the server and apply them to their local state to update their ECmRDTs. This way, clients do not need to store events locally and can always access the latest version of the data.

Extensions are functions that can be attached to an ECmRDT to enhance its functionality. Extensions can be defined in a pipeline, which is a sequence of extensions that are applied to an event before it is applied to the state. Extensions can perform various tasks, such as:

- Verifying the replicaId of an event to ensure its origin
- Changing some variables in the state before or after applying an event
- Migrating events from one format to another
- Authenticating and authorizing events based on some criteria

The main advantage of ECmRDTs is that they enable developers to create secure and customizable data types that can be easily extended with additional functionality. For example, one can use extensions to implement access control policies or data validation rules for an ECmRDT. Extensions can also interact with other extensions and exchange information or modify each other's behavior.

In summary, ECmRDTs are a novel approach to extend CmRDTs with security and customization features using extensions. Extensions are functions that can verify and mutate events and state in a pipeline. They can also interact with other extensions and authenticate and authorize events based on some criteria. Events are signed operations that modify the state of an ECmRDT and are stored on a server that acts as an event source. Clients can request events from the server and apply them to their local state to update their ECmRDTs.

Example

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 covered the next section.

2.2 Concepts

This section explains the main ideas of our ECmRDT with definitions, code examples and how they are connected.

Aggregate

An aggregate is a group of related objects that we handle as one unit when we change data. Each aggregate has a root and a boundary. The boundary shows what is inside the aggregate. The root is one specific object in the aggregate.

In general, an aggregate is a domain concept where only one aggregate at a time should be changed by a use case. We will define one ECmRDT for each aggregate.

ReplicaId

The ReplicaId is a unique identifier for a user. The ReplicaId has a public/private key pair. With this, we can sign and verify events sent by a user. Because the private key only exists in the corresponding replica as a privateReplicaId, the ReplicaId only has the public key for identification. More details about the ReplicaId are 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 in the aggregateId shows the owner of the aggregate. We need to store the ReplicaId to stop other replicas from creating aggregates with the same AggregateId on purpose. The random bytes are used to avoid 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 cryptographic operations to check the event which are done in web browsers asynchronously.

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

Event

Events are made by users to change the state. Usually events are caused directly by user actions. Events are linked with a context. So events have information specific to one particular event and the context has information that is contained in every event. Events can be changed into effects to be used later to update 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.

```
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.

```
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.

```
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]
```

2.3 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.

2.4 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. We want to allow aggregates to specify a set of permissions for different events. Some events may need permissions

only under certain conditions defined by the aggregate. Users who want to use these events have to provide proofs for the permissions that can be verified by other users.

Proofs are created by signing a replicaId (the user who wants to use the event) with asymmetric encryption. This means that a proof is valid only for one user. Other users have to generate new proofs. A proof consists of a public key (called a claim) and a private key (called a prover). Claims are stored in the aggregate state publicly so that other users can check incoming events.

Claims and provers are generated from the initial event, but provers are usually not stored in the aggregate. The only exception is an extension called ClaimBehindPassword that aims to reduce the size of the keys that have to be sent. It works by storing the provers encrypted with a symmetric key publicly in the aggregate with a shorter password. Anyone who knows the password can create their own proofs.

To make this system work, we have to ensure that every event that claims to be sent by a replicaId is actually sent by that replicaId. This is important because when we validate proofs, we check a signature that contains the sender replicaId. If someone could spoof the sender replicaId, they could use an existing proof sent by another user (which is possible because proofs are public by default so that everyone can validate them) and pretend to be them. Users would then not be able to verify the event correctly.

This problem is why the replicaId is a public key that only the user knows the private key for. Events are signed with the private key so that everyone who receives events from a replicaId can be sure that they are authentic

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."))
    )
```

How this extension can be used will be shown in the next section where we will look into how ECmRDTs are used to model the Ratable domain.

2.5 Integration into Ratable

Ratable itself has a simple domain. A Ratable is an object containing a title, categories and ratings. To enable authorization we are using the AsymPermissionExtension with the provided EffectPipeline and to allow easier sharing of private keys we use the ClaimByPasswordExtension. The context provides no custom information and contains the standard IdentityContext as well as the AsymPermissionExtension context.

```
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]
)

case class Rating(
  val ratingForCategory: Map[Int, Int]
)

case class Category(
  val title: String
)

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

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

Additionally, we define our domain around our state to enable better readability and code

reuse.

```
case class Ratable(...):
  def rate(
    replicaId: ReplicaId,
    ratingForCategory: Map[Int, Int]
  ): Ratable = ...

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

We only need to provide an event called RateEvent and one Claim called CanRate to create or replace a rating.

```
// we need to extend Enum[T] to allow serialization of enum
enum RatableClaims extends Enum[RatableClaims]:
  case CanRate

// sealed base trait to allow serialization
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)
```

3 Architecture

In this chapter, we present the design and implementation of Ratable, a web application that allows users to rate and compare different products and services. We describe the main components and features of Ratable and explain the rationale and challenges behind our design and implementation choices. We also show some code snippets and diagrams to illustrate how Ratable works.

3.1 Overview

Ratable is a web application that allows users to rate and compare different products and services. The web application is built with Scala and uses a serverless architecture to provide cost-effective and scalable services. The main components of the architecture are:

- A static web application that is delivered through a CDN and can be installed as a PWA on the user's device.
- A serverless backend that handles the business logic and state management of the application. The backend is deployed as a cloud function that can be triggered by events or HTTP requests.
- A serverless nosql database that stores the data of the application. The database is accessed by the backend through a cloud service API.
- A pubsub service that enables real-time updates and asynchronous messaging between the web application and the backend. The pubsub service manages the websocket connections and publishes or subscribes to messages based on events.
- A protobuf protocol that defines the message format and structure for the communication between the web application and the backend. The protobuf protocol supports 'one of' relationships and reduces the message size compared to JSON.

The following diagram illustrates the architecture of Ratable

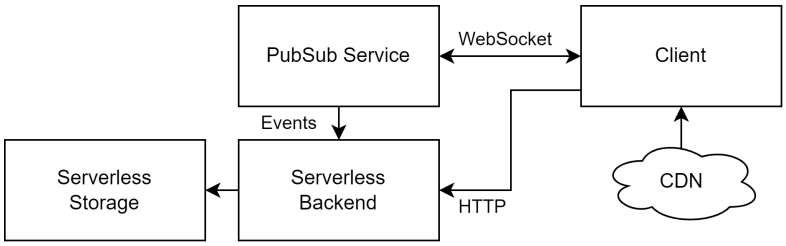

Abbildung 3.1: Architecture overview

3.2 Reasoning

In this section, we explain the reasons why we chose the architecture described in the previous section for Ratable. We discuss the advantages and disadvantages of each component and how they fit our requirements and goals.

ECmRDT would allow us to use peer-to-peer communication instead or additionally to client-server communication. The reason why we decided against it is that it would be out of the scope of this thesis. Implementing peer-to-peer communication would add multiple new problems in implementing the communication, state management and event ordering. It should be noted that theoretically we have indirect peer-to-peer because our server does not do anything more than forwarding all Events.

3.2.1 Static web application

We chose to provide our services through a web application because it allows us to reach a larger audience (e.g. mobile and desktop devices) in a more accessible way without downloads. Additionally, we provide PWA functionality to enable users to install Ratable as an app on their device and access it offline. We used Scala as the programming language for the web application because it has good tooling and support for web development. We did not use server-side rendering because it did not have many benefits for our use case and the tooling for Scala was not good. Instead, we used a simple and cost-effective approach of delivering the web application to users by giving access to the static files through a CDN.

3.2.2 Serverless backend

We chose to use a serverless backend because it is very cost-effective (in fact free with our cloud provider), simple to deploy without any system configuration issues, and scalable to handle variable workloads. The main drawback of serverless is that it requires a special design for the application to work in a stateless and event-driven environment. However, this was not a major issue for us because we wanted to create a cloud-native solution and we implemented enough abstraction to be able to change the cloud provider without a complete rewrite. We used Scala as the programming language for the backend because it is compatible with the web application and it has good support for concurrency and distribution. We used an event sourcing pattern to persist the state changes as a sequence of events in the database, which enables us to make state always accessable by users without violating local-first principles.

3.2.3 Serverless nosql database

We chose to use a serverless nosql database because it is also very cost-effective (again free with our cloud provider), easy to access through a cloud service API, and scalable to handle large amounts of data. The main advantage of nosql is that it allows us to use a flexible schema design that can accommodate changes in the data model without affecting the existing data. The main disadvantage of nosql is that it does not support complex queries or joins that are common in relational databases. However, this was not a problem for us because we did not have specific requirements for querying or joining data and we followed the recommended way by our cloud provider to use this service. The database also supports transactions and consistency guarantees for the data operations, which are important for ensuring data integrity and reliability.

3.2.4 Pubsub service

We chose to use a pubsub service because it enables us to provide real-time updates and asynchronous messaging between the web application and the backend. The pubsub service manages the websocket connections and publishes or subscribes to messages based on events. This way, we can avoid polling or refreshing the page to get updates from the backend, which improves the user experience and reduces the network traffic. The pubsub service also allows us to implement a publish-subscribe pattern that decouples the

producers and consumers of messages, which increases the modularity and scalability of our system.

3.2.5 Protobuf protocol

We chose to use protobuf as the protocol for defining and encoding the messages between the web application and the backend because it has several advantages over JSON, which is commonly used for web applications. Protobuf supports 'one of' relationships, which are useful for representing different types of messages with different content. Protobuf also reduces the message size compared to JSON, which improves the network performance and bandwidth usage. Protobuf also provides type safety and compatibility checks for the messages, which prevents errors and conflicts in communication.

3.3 Project Structure

In this section, we describe the project structure of Ratable and how it reflects the architecture and design choices we made in the previous sections. We explain how we organized the code into modules, layers, and services, and how we ensured testability and modularity of our system.

Ratable consists of two main applications: the serverless backend and the web application. The serverless backend runs as a cloud function that accepts messages over HTTP or events over websockets from the web application or the pubsub service. The web application runs as a static web page that provides an interface to the user and communicates with the backend or the pubsub service. Additionally, we have a common core module that contains the domain model, the ECmRDT implementation, and all protobuf definitions that model the communication between the backend and the web application.

Both the web application and the backend are structured similarly into two main layers: the application layer and the device layer. The application layer contains the business logic and the use cases of the system. The device layer contains the interaction with the device resources, such as the network, the database, or the user interface. Between these two layers, there can be additional supportive layers that abstract away the device layer or add new functionality, such as a state layer that manages the state changes and events.

Layers interact with each other using services. Services are classes or objects that provide specific functionality or operations to other components. We use dependency injection to inject the services that each component needs. Generally, services should only request services from the same layer or from a lower layer. This is to prevent cyclic dependencies and to maintain a clear separation of concerns.

The application layer is structured differently between the web application and the backend. In the web application, we use a use case oriented approach. This means that we define different use cases that correspond to different user actions or scenarios, such as rating a Ratable. Each use case is implemented as a service that interacts with other services from the same layer or lower layers. We use a state layer to manage the state changes and events that result from executing a use case. The state layer uses ECmRDTs to ensure eventual consistency and conflict resolution for concurrent updates. The state layer also interacts with the device layer to persist or synchronize the state with the backend or the pubsub service. The device layer uses protobuf to encode or decode the messages that are sent or received over HTTP or websockets. The device layer also provides an interface to access or manipulate the user interface aspects, such as routing.

In the backend, we use an event driven approach. This means that we define different gateways that correspond to different sources or sinks of events in our system, such as HTTP requests or websocket connections from pubsub messages. Each gateway is implemented as a service that interacts with other services from the same layer or lower layers. We use handlers to process and respond to the events that are received or sent by the gateways. In handling the state the handlers only persist the list of events without actually caring about the contents. Its only responsibility is to act as a event store. The device layer uses protobuf to encode or decode the messages that are sent or received over HTTP or websockets. The device layer also provides an interface to access or manipulate the database or the pubsub service.

3.4 State managment

In this section, we describe the state layer that we use in the web application to manage the state changes and events that result from using ECmRDTs. ECmRDTs are a core part of this thesis and they provide a new way to work with data in a distributed and eventually consistent system. We explain how we designed the state layer to provide a convenient and simple interface for the programmer to interact with ECmRDTs, while also abstracting away the complexity of operation based CRDTs.

The state layer consists of two main components: repositories and aggregate views. A repository is a class that represents a collection of aggregates that share the same ECmRDT type. An aggregate is an instance of an ECmRDT that has a unique identifier and a state. The state is a result of a sequence of events that are applied to the ECmRDT to produce a value. The actual events are only stored in the server acting as our event store. The programmer can use the repository to create a new aggregate or load an existing one by providing its identifier. It is currently not possible to delete an aggregate from the repository.

An aggregate view is an object that represents a single aggregate and allows the programmer to listen for changes and submit events. The aggregate view is the only way to interact with aggregates outside of the state layer. The programmer can use the aggregate view to access or modify the value of the aggregate by applying events. The events are encoded using protobuf and sent to the backend or the pubsub service using the device layer. The aggregate view also receives events from the backend or the pubsub service and applies them to the aggregate state using the ECmRDT logic. The aggregate view notifies the programmer about any changes in the value of the aggregate using callbacks or observables.

```
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]
```

Internally the aggregate is contained by an AggregateFacade. It contains the concrete ECmRDT value and allows free mutations around it. This free ability of free mutations is needed for operations like acknowledging sent events. This additional abstraction is needed because we do not store the ECmRDT directly. We need additional wrappers to enable communication and asynchronous mutations.

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

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

These AggregateFacades are then created and cached by the AggregateFacadeProvider so that only one exists per aggregate at any moment. Around the AggregateFacadeProvider we have the AggreagteViewProvider which converts AggregateFacades into AggregateViews.

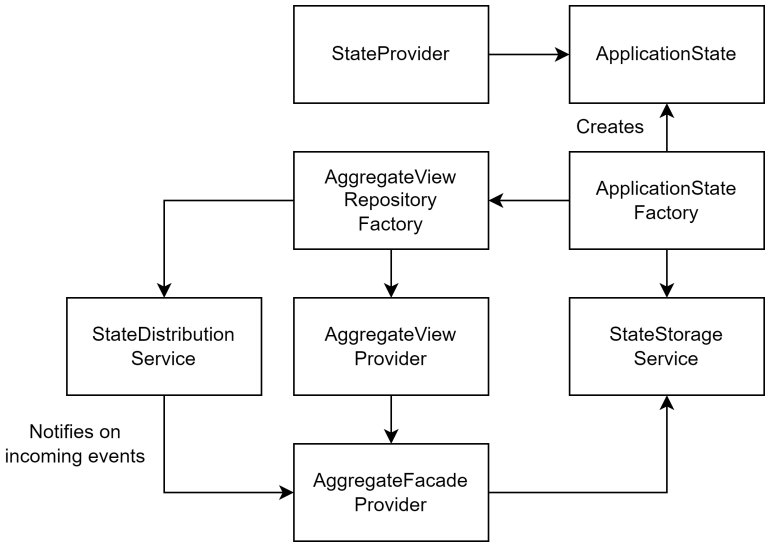

Abbildung 3.2: Architecture overview

Access to the AggregateViewProvider is handled through AggregateViewRepositories which are created through the AggregateViewRepositoryFactory and then saved as ApplicationState. The separation between AggregationViewProvider and AggregateFacadeProvider is needed so that other state services can access the AggregateFacade instead of the AggregateView. The two services currently using this are the StateStorageService handling loading and saving aggregates on the user device and the StateDistributionService handling outgoing and incoming events.

Using this abstraction we can now provide an ApplicationState like the following.

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

4 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.

- evaluation (ecmrdt als konzept messen -> speed, time) - negative aspekte darstellen - examples for ecmrdts - Conclusion and future - The ratable case study (title der case study)