Algebraic Replicated Data Types: Programming Secure Local-First Software

Christian Kuessner
Technische Universität Darmstadt, Germany

Ragnar Mogk
Technische Universität Darmstadt, Germany

Anna-Katharina Wickert
Technische Universität Darmstadt, Germany

Mira Mezini
hessian.AI, Darmstadt, Germany
Technische Universität Darmstadt, Germany

--- **Abstract** ---

This paper is about programming support for local-first applications that manage private data locally, but still synchronize data between multiple devices. Typical use cases are synchronizing settings and data, and collaboration between multiple users. Such applications must preserve the privacy and integrity of the user's data without impeding or interrupting the user's normal workflow – even when the device is offline or has a flaky network connection.

From the programming perspective, availability along with privacy and security concerns pose significant challenges, for which developers have to learn and use specialized solutions such as *conflict-free replicated data types* (CRDTs) or APIs for centralized data stores. This work relieves developers from this complexity by enabling the direct and automatic use of algebraic data types – which developers already use to express the business logic of the application – for synchronization and collaboration. Moreover, we use this approach to provide end-to-end encryption and authentication between multiple replicas (using a shared secret), that is suitable for a coordination-free setting. Overall, our approach combines all the following advantages: it (1) allows developers to design custom data types, (2) provides data privacy and integrity when using untrusted intermediaries, (3) is coordination free, (4) guarantees eventual consistency by construction (i.e., independent of developer errors), (5) does not cause indefinite growth of metadata, (6) has sufficiently efficient implementations for the local-first setting.

1   Introduction

Today, the dominant software architecture for distributed applications is centralized. This is true for a wide variety of application types, such as single user applications deployed on multiple devices (e.g., calendars, notes, email, etc.), software that enables multi-party collaboration (e.g., shared calendars, document editors, business workflows, etc.), and software for autonomous systems with remote control and interactions (e.g., home automation and autonomous vehicles). Data is collected, managed, and processed in the cloud. Devices at the edge – owned by individuals and companies – serve merely as gateways to the cloud. Such an architecture has strengths, but also several weaknesses: It causes undue lost control over data ownership and privacy, lack of offline availability, poor latency, inefficient use of communication infrastructure, and waste of powerful computing resources on the edge.

To address these issues, local-first software design principles [25] call for "data confidentiality and privacy by default" and "ultimate ownership and control" by the user – both to be achieved by moving data storage and processing to the edge. However, developing local-first applications is challenging. Crucially, suitable existing mechanisms for efficient decentralized data management, specifically *coordination-free replicated data types (CRDTs)* [52], were invented for the geo-replicated database setting, which differs significantly from the local-first setting.

First, each local-first application has its own unique application-specific data model, designed by developers to encode domain knowledge. Developers have to figure out how to map application-specific data models to CRDTs, which are only available "off-the-shelf" in the form of databases [50] or libraries with a fixed API [24, 37]. Designing application state based on a fixed set operations is known to cause application design issues [11], because it requires translation between the application domain model and the fixed set of operations.

Second, in a local-first setting, a diverse set of networks is used for state synchronization. But general off-the-self CRDTs are designed for the geo-replicated database setting. The assumed network has mostly available direct connections between replicas, i.e., systems are designed to deal with seconds or minutes of latency between data centers. In contrast, local-first replicas (on user devices) have varied network conditions, ranging from always online devices, to personal computers that are turned off when unused, to mobile devices that only synchronize data when connected to a Wi-Fi network. In such a setting, we cannot assume direct connections between devices. Notably, this implies that connection oriented security protocols (e.g., TLS) are not applicable. A common solution for such indirect communication are cloud servers that act as intermediaries, i.e., as post offices that store, manage, and forward messages – further complicating the network model.

Third, the geo-replicated and the local-first settings have different security assumptions. Local-first applications often process personal data, with better data privacy and security being a selling point to users of local-first software. Existing security efforts for CRDTs in the geo-replicated setting [4] do not apply, due to weak attacker models and reliance on encrypted direct connections. Specifically, communication over untrusted intermediaries must not jeopardize the principles of local-first software, i.e., intermediaries must be unable to inspect or modify application data.

To fill the gaps, we propose *algebraic replicated data types* (ARDTs) – algebraic data types (ADT) that provide, by construction, provably consistent decentralized data management. ARDTs are delivered as a library that integrates seamlessly with existing language support for function composition and algebraic data types. Behind the scenes, ARDTs combine the theory of consistency as logical monotonicity (CALM theorem) [12] with delta replication [2]

for efficient and correct synchronization. Moreover, ARDTs also offer an encryption layer for efficient synchronization over untrusted intermediaries. Overall, ARDTs provide the ease of development of traditional applications, the privacy advantages of local-first, with the data sharing advantages of clouds – independent of the underlying network. We evaluate the proposed ARDT-based design of local-first applications on a case study and with micro-benchmarks. The results show that (a) typical local-first applications can be implemented with negligible performance overhead compared to existing data synchronization and UI rendering costs, and (b) encryption comes with minimal computational costs and with predictable, reasonable storage overhead on intermediaries.

In summary, our contributions are:

- A critical analysis of the state-of-the-art in developing local-first applications (Section 2).
- Use of standard ADTs suitable for application design for replication the realm of local-first software (Section 3).
- A novel synchronization-free authenticated encryption scheme, itself provided as an algebraic replicated data type (Section 4). As part of designing the encryption scheme, we contribute a systematic analysis of the suitability of existing encryption primitives for decentralized synchronization protocols (Section 5).
- An implementation of our proposal as an embedding into Scala along with a systematic empirical evaluation (Section 6).

2   State of the Art and Problem Statement

Below, we discuss two families of existing approaches, which are relevant for our work: (a) dedicated systems for collaborative workflows and (b) approaches to replicated data types employed in geo-replicated data stores. We briefly introduce relevant and missing features with respect to developing secure local-first software. We also introduce existing building blocks for distributed systems programming, which we adopt and combine to exploit their advantages in our setting.

2.1   Systems for Distributed Workflows

There are two kinds of systems for distributed workflows. The first kind has automated handling of conflicts at the price of centralized coordination; prominent examples are Google Docs or Firefox Sync. The other kind has flexible replication that does not rely on centralization; the most prominent example is Git, which allows for replication via different intermediaries including specialized ones like GitHub, or general ones like email. Similarly, systems like Syncthing and Resilio Sync enable peer-to-peer file synchronization in an arbitrary network topology, and even support encrypted intermediaries. But Git, Syncthing, and Resilio Sync require manual user intervention for conflict resolution.

We aim for combining flexible and secure data synchronization à la Git with automated conflict resolution à la Google Docs. Crucially, we aim to offer this combination to general-purpose programs with unconstrained types of data. This is unlike the above solutions, which target specific use cases and specific types of data. For example, Google Docs builds on research around operational transform [54] to enable efficient synchronization specifically for text documents. Such specialized solutions are infeasible for arbitrary local-first applications from different domains. Adapting existing solutions would require developers to become experts and understand the underlying assumptions and data models, or otherwise risk to introduce errors into an adaptation.

2.2   Replicated Data Types in Geo-replicated Data Stores

Another class of solutions that are relevant for our purposes are those developed to enable availability in geo-replicated data stores in the presence of network partitions – a scenario that bears some superficial similarity with local-first software. In particular, the solution from this context that is most relevant to the development of local-first applications are conflict-free replicated data types (CRDTs) [52]. CRDTs are data types, whose API consists of a fixed set of query and update operators, which satisfy the condition that two replicas that know of the same updates return the same result for all queries (also known as *eventual consistency*). This property is key in supporting coordination-free synchronization. CRDTs are typically built into a replicated data store with specific assumptions about the underlying network for efficiency (e.g., availability of reliable causal broadcast, trustworthiness of the involved servers).

The assumptions built into the design of off-the-shelf CRDTs limits their applicability to local-first software development. First, application developers are left with not much choice but to express their application design using the fixed APIs of existing CRDTs. A similar approach – object-relational mappings in database-centric software – is known to be a leaky abstraction, requiring frequent security relevant changes, and does not work well together with language-based tooling [11]. Second, local-first software operates in varied network scenarios for which there is no "one size fits all" solution to handle network communication. Thus, some CRDT runtimes allow developers to provide a custom message dissemination system that is specific to their needs. However, for two common network scenarios – using a cloud provider to store and forward messages, and using epidemic routing in an ad-hoc network – messages are not secure by default. Adding security burdens developers with ensuring correct and efficient encryption of messages, a task that requires expert knowledge of both the CRDT implementation and the network dissemination scheme to accomplish correctly and efficiently.

To recap, local-first applications have multiple new challenging concerns including design of the application state with replication-awareness, efficient dissemination of messages given the target network topology, and security of exchanged data, considering that messages may be stored for a long time before delivery. Each of these concerns needs both, system-level expert knowledge and application-specific insight. It is too much to ask of application developers to become experts in all of these fields. Thus, we must make expert implementations of system-level concerns such as state synchronization, message dissemination over physical networks, and security, available in a reusable and composable manner to application developers.

2.3   Building Blocks for Algebraic Replicated Data Types

Our solution builds on insights from previous research, but makes them reusable and combinable by application developers.

CALM and lattices.   The first result we exploit is the *consistency as logical monotonicity* (CALM) theorem [21]. It states that consistency is possible without coordination if and only if all replicas only add to (i.e., monotonically increase) but never invalidate prior results. Existing solutions using replicated data types generally require all operations to prove (or pray for) a correctness property related to monotonicity. For example, state-based CRDTs require all operators to monotonically increase the state according to a specified order, and operation-based CRDTs require all operators to commute with one another. However, as we want to enable developers to define their own synchronization-free replicated data types

including new operations, it would be too easy for them to accidentally introduce consistency bugs (i.e., design operations that are not eventually consistent). A constructive way to enable consistency by-design, is to restrict available programming support to monotonic functions [12]. But such an approach may be too restrictive and does not integrate well into general-purpose languages.

Our solution is based on join semi-lattices – a set of states (i.e., a data type) with an associative, commutative, and idempotent *merge function*. In practical terms, associativity ensures that states can be combined before transmission, commutativity tolerates disordered arrivals of messages, and idempotence handles duplicated transmissions. Any (merge) function $\sqcup$ with the above three properties provides monotonicity in the sense above, because it implies an order on states: $s_1 \leq s_2$ if and only if $s_1 \sqcup s_2 = s_2$. Lattices have been used to reason about correctness of CRDTs since their beginning [52], but our approach makes states and merge functions directly available as building blocks for application developers.

Delta replication. We use delta replication [2] to separate efficient message dissemination from the application logic. Delta replication is a variant on state-based replicated data type design, where monotonic changes are expressed as a delta to prior state. The overall application state results by merging all deltas (according to the lattice merge). The application logic is free to generate deltas however it wants, and the message dissemination algorithm is free to optimize the transmission of deltas for the specific networking platform.

Note that not every combination of lattice semantics and message dissemination provides causal consistency (only eventual consistency). However, causal consistency always comes at the cost of waiting for messages to arrive, and most local-first applications do not require causal consistency. It is well understood how to add causal delivery to any message dissemination scheme, and doing so is compatible with our results.

But delta replication does not tell how to add encryption transparently, i.e., without having to adapt all existing message dissemination implementations (of which, each application may have its own). In general, message dissemination in a local-first setting needs to consider many application- and environment-specific interwoven concerns in addition to security. Such concerns include causal delivery, importance of messages to the application logic, and messages that become obsolete because of newer state changes. While we focus on security, our approach is designed to be parametric over the dissemination strategy for deltas, and thus enables customization.

Authenticated encryption with associated data. Local-first applications require confidentiality – the guarantee that application private data can not be accessed by unauthorized parties – and integrity – the guarantee that neither private data nor communication metadata (the associated data) can be tampered with by an attacker. These guarantees are provided by *authenticated encryption with associated data* (AEAD) [44], a family of cryptographic solutions based on symmetric-key cryptography, where only trusted parties (in our case replicas) have access to a single shared key. AEAD is well studied and widely used, e.g., in TLS 1.3 [43]. But each use of a cryptographic construction in a new field requires to carefully select concrete implementations of cryptographic functions and to ensure that they are executed with suitable parameters. The local-first setting is no exception in this respect. Specifically, in local-first applications, an unknown number of replicas need to encrypt and decrypt data using a single shared key. Widely available AEAD functions require a globally unique number (nonce) as an input for each operation using the same key. Guaranteeing global uniqueness requires coordination, which we need to avoid in the local-first setting.
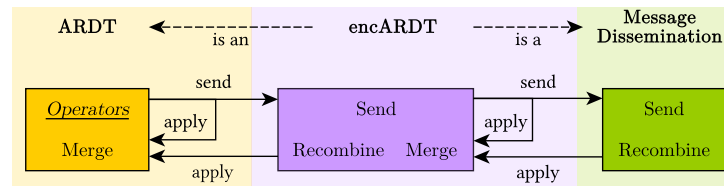
■ **Figure 1** Architecture overview.

3   Algebraic Replicated Data Types

The architecture of our solution is structured in three layers depicted as colored areas in Figure 1. The layer on the left concerns algebraic replicated data types (ARDTs), which are used to model the application logic. The layer on the right concerns message dissemination over physical networks; this includes sending messages (serialized deltas) and applying a merge function to recombine received deltas into the full application state. Finally, the middle layer concerns encrypted replication (encrypted ARDTs – encARDTs). We provide a library with implementations of each layer. It includes ARDTs for common data types auch as set and map, different encARDT implementations with different performance versus metadata tradeoffs, and implementations for message dissemination over TCP, Websockets, WebRTC, and disruption tolerant networks (DTNs). We expect that developers want to design new ARDTs specific for their application logic. In doing so, they can freely combine different implementations of each layer to address specific application needs. This section presents how developers design their own ARDTs and configure it to use a message dissemination module. Section 4 and Section 5 elaborate on encrypted ARDTs.

3.1   Programming and Replicating ARDTs

An ARDT is an (immutable) algebraic data type (ADT) of the host language (Scala in our case) plus a set of associated operators. The ADT values represent the (lattice) state of the ARDT. The values represent application data and, depending on the used lattice, may also include metadata for automatic merging. Operators are functions/methods that operate on an ARDT's state. Operators may (i) just read the ARDT state to produce a value used by the application, or (ii) produce a delta that describes the desired changes to the current state. A delta is technically an instance of the ARDT, but it must first be merged into the current state to become meaningful in the context of the application.

For illustration, assume that we want to implement a local-first social media application to be used by a group of friends in a peer-to-peer network to share messages, comments, likes, and dislikes. The ARDT in Figure 2 models the state and operators of such an application[1]. The SocialMedia type (Line 1) is defined as a product type with named components (a case class in Scala). SocialMedia wraps a Map (Scala's built-in dictionary type) of IDs to values of type SocialPost (Line 6 – type parameters are in square brackets). A social post uses the built-in type Set for comments, two Counters for likes and dislikes, and a LWW (last-writer-wins) register for the post and comment contents. The LWW register is a built-in ARDT provided by our library that can be set to a new value, with the implication that all replicas will show the newest value according to a real-time clock. Counter is an ARDT defined in Figure 3.

---

[1] All code examples in the paper use Scala 3 syntax.

```
1  case class SocialMedia(sm: Map[ID, SocialPost]):
2    def like(post: ID, replica: ReplicaID): SocialMedia =
3        val increment = sm(post).likes.inc(replica)
4        SocialMedia(Map(post -> SocialPost(likes = increment)))
5
6  case class SocialPost(message: LWW[String], comments:
        Set[LWW[String]], likes: Counter, dislikes: Counter)
```

Figure 2 Compositional design of the social media ARDTs.

```
7   case class Counter(c: Map[ReplicaID, Int]):
8     def value: Int = c.values.sum
9     def inc(id: ReplicaID): Counter =
10      Counter(Map(id -> (c.getOrElse(id, 0) + 1))
11
12  object Counter: // object for static methods
13    def zero: Counter = Counter(Map.empty)
```

Figure 3 The state and operators of a counter ARDT.

The operators of ARDTs implement their application logic. While Counter (Line 7) and SocialMedia (Line 1) are both wrappers around a Map, their operators make the difference. Each Int stored in the Map of the Counter ARDT represents an individual amount contributed by the specific ReplicaID. This is expressed by the value operator (Line 8). A zero counter is expressed by the empty map (Line 13). Like other immutable data structures, operators that modify ARDTs return a new state, e.g., inc (Line 9) increases a counter by returning a new counter. But for ARDTs it is sufficient to return a *delta* – the changed parts of the state – the rest is managed automatically by applying the merge function. For instance, inc (Line 9) returns only the entry with the increased values; unchanged entries in the Map are omitted. The like operator of the SocialMedia ARDT in Line 2, while being a bit more complex, follows the same pattern. To "like" the post with the given ID, it computes the increment of the likes counter (Line 3) and returns a new delta of the SocialMedia state, which contains only the changed ID and defines only the likes component[2] of the social post (Line 4). Returning deltas is preferable, because it is more efficient to send and merge smaller values. But since merging is idempotent, developers could also return full states without impacting behavior.

In the examples so far we assumed that a merge function for our ARDTs exist. This is indeed the case, because all built-in types we used have merge functions provided off-the-shelf by our library, and the user-defined ADTs (SocialMedia, SocialPost, and Counter) have their merge function automatically generated. For example, the merge functions for Counter and SocialMedia keep all entries of both maps and (recursively) merge the values that have the same key; and the merge function of the SocialPost merges each component individually. See Subsection 3.2 for the precise definition of these merge functions. In general, the availability of a merge function for a type S (e.g., Counter or SocialMedia) is modeled by the type class Lattice[S] below.

```
14  trait Lattice[S] { def merge(left: S, right: S): S }
```

---

[2] The syntax that looks like an assignment in Line 4 is a named parameter, and we assume that this constructor sets all other components to "empty" values (not shown in the example for brevity). The -> operator constructs a key-value pair.

```
15  class MessageDissemination[S]:
16    def send(delta: S): Unit
17    def recombine(using Lattice[S]): S
```

Figure 4 Example message dissemination module.

```
18  val smd = new MessageDissemination[SocialMedia]
19  val current: SocialMedia = smd.recombine
20  val delta: SocialMedia = current.like(myPost, replicaID)
21  smd.send(delta)
22  val updated: SocialMedia = smd.recombine
```

Figure 5 Using and replicating the social media ARDT.

Specifically, we say that S is the state of an ARDT, when there exists a correct instance of Lattice[S]; an instance is correct if its merge function is associative, commutative, and idempotent. The correctness of the merge function is the only requirement for eventual consistency in our system, and we do not want to burden developers with its definition. Thus, our library comes with a broad range of built-in ARDTs (with state, operators, and merge function). Moreover, correct instances of Lattice for standard data structures, user-defined product types, and the compositions of all of the above, are automatically generated. The system produces a compilation error if no lattice instance for a custom type can be generated. In other words, eventual consistency is always guaranteed automatically.

We put ARDTs into action by composing them with a concrete message dissemination module. The API of the message dissemination module is shown in Figure 4. It allows to send and recombine a replicated state of type S (Line 15)[3]. The send method (Line 16) sends a delta message of type S. The recombine method (Line 17) merges all received delta messages into a full state of type S, which requires a Lattice[S]. The using keyword (Line 17) asks the compiler to provide such an instance automatically if available, or report a compilation error. Figure 5 shows how to use and replicate the social media platform. Line 18 creates a MessageDissemination named smd. We access the current state of social media (current) using recombine (Line 19). To like a post named myPost, we first apply the like operator on current to compute the delta state.[4] Once we send delta (Line 21), the like is merged into the rest of social media, and we can access the full updated state by calling recombine (Line 22).

3.2   Lattice Composition

By design, the correctness of both the operators and the distributed consistency of ARDTs rely exclusively on their state forming a lattice, i.e., on having a correct merge function. We provide ready-to-use Lattice instances for a range of data types such as last-writer-wins registers, multi-version registers, and lists (RGA). We have implemented those based on existing schemes for state-based CRDTs [51, 2]. On top, our library supports automatic generation of merge functions for compound data types including associative maps, pairs, tuples, optional values, and user-defined case classes (generally all product types), given their

---

[3] This supports multiple ARDTs by composing them into a single type S.
[4] Note that while delta is of type SocialMedia, it only contains that single like.

```
23  given Lattice [Int] with
24    def merge(left: Int, right: Int): Int = max(left, right)
```

Figure 6 Lattice instance for integers using their maximum.

```
25  given [A]: Lattice[Set[A]] with
26    def merge(left: Set[A], right: Set[A]): Set[A] = left union
         right
27
28  case class LWW[A](time: Time, value: A)
29  given [A]: Lattice[LWW[A]] with
30    def merge(left: LWW[A], right: LWW[A]): LWW[A] =
31      if right.time < left.time then left else right
```

Figure 7 Set and last-writer-wins lattice.

constituents are ARDTs (i.e., have a lattice instance)[5]. For example, SocialPost's merge is automatically generated from its constituent types, Set, Counter, LWW. The generation is recursive with pre-defined ARDTs (e.g., LWW) being the recursion anchors.

The generation for product types exploits the canonical representation of a compound data type as a function (e.g., a key-value map is a function from keys to values). The merge of two functions l and r is a new function f that merges the result of applying l and r ($f(x) = merge(l(x), r(x))$). Sum types (i.e., types representing alternatives such as colors: red, green, blue) either use built-in lattice instances, or use an explicitly specified order of the cases (e.g., red < green < blue) and merging returns the larger case.

In the following, we elaborate on how concrete lattice instances are defined for different ARDTs starting with the recursion anchors and ending with automatic derivation of instances for compound data types.

3.2.1 Provided and Custom Lattice Instances

We provide ready-to-use lattice instances for primitive data types and for common CRDTs. For example, the code in Figure 6 implements lattice instances for integers. The given keyword defines an unnamed instance of Lattice[Int]. The with keyword states that the following is the implementation of the Lattice methods, in this case, the implementation of merge as the application of the max function. This definition uses Scala's support for implicit values to seamlessly integrate ARDTs into the rest of the language. Methods can access an instance with the using keyword (as seen in the recombine method from Figure 4); the developer does not need to explicitly provide the instance, when the method is called.

Figure 7 shows lattice instances for sets and last-writer-wins registers. Merging sets is delegated to the existing union method on sets. LWW is a custom type, whose state associates a unique timestamp and a value. Its merge function makes an arbitrary but deterministic decision – it selects the state with the larger timestamp and ignores the other one.

We do not expect developers to define their own merge functions. It is possible to do so, by providing custom lattice instances, but carries the risk of an incorrect implementation. Instead, our library provides support to automatically derive lattice instances for custom ADTs, as elaborated in the following sections.

---

[5] This is not unlike Haskell's support for deriving instances of type classes for compound data structures.

```
32  given [K, V](using Lattice[V]): Lattice[Map[K, V]] with
33    def merge(left: Map[K, V], right: Map[K, V]) =
34      left.merged(right){
35        case ((id, v1), (_, v2)) => (id, Lattice[V].merge(v1, v2))
36      }
```

Figure 8 Map lattice.

3.2.2 Derived Lattice Instances

Deriving lattices for a compound type makes use of lattices of its component types. Technically, this is represented as given instances that take other instances as parameters. In the following, we present how to derive lattices for generic map and product types. The appendix includes proofs of their correctness by showing that the respective merge functions are commutative, associative, and idempotent.

The map lattice. Figure 8 states how any Map[K, V] has a lattice instance, if its values V also have a lattice instance. Specifically, the using keyword states that to create the lattice instance for the map we require a Lattice[V] where V is the type of values stored in the map. The merge function (Line 33) for a map delegates to the built-in merged function of Map (Line 34). The built-in merged function does not automatically handle the case when a key is assigned to value in both the left and the right map and requires a custom function to handle such conflicts. We implement this function to delegate to the merge function of the value type provided by Lattice[V] (Line 35). We prove correctness of this merge function in Appendix A.1.

The product lattice. We support automatic generation of lattices for any product type whose elements themselves have lattice instances. In Scala, product types include tuples and case classes. Consider the exemplary case class MyData in the first line of the code snippet below, and an explicit definition of the automatically generated lattice instance in the second line.

```
37  case class MyData(a: A, b: B)
38  given Lattice[MyData] = Lattice.derived[MyData]
```

The derived method generates a lattice instance for any product type S. Its implementation is shown in Figure 16 (in the Appendix). At a high-level of abstraction, a lattice instance for a product is generated as follows. (i) Acquire lattice instances for each component of the product. (ii) Define the merge function for two instances of the product type (a left and a right one) to (iii) take each component of the left product and merge it with the corresponding component of the right product and (iv) return the results wrapped in a new instance of the product.

We give the full technical details of the implementation in Appendix A.2. We prove correctness of the merge function for arbitrary products in Appendix A.3.

3.3 Qualitative Assessment of Design Features

We recap key advantages and limitations of our approach to defining and using ARDTs.

Reused implementations.   When designing a new ARDT, expert developers can reuse any existing data structure, as long as they can define a merge function. For instance, the Map data structure in our social media ARDT is a highly optimized implementation from the Scala standard library. But developers are not limited to the options in our library and are free to choose an implementation that best suits their needs. For instance, there are multiple implementation strategies for sets to chose from, including sorted trees and hash-based solutions. The decision about the particular data type to use for representing the state of custom ARDTs is decoupled from and does not affect consistency management. Our approach enables to reuse existing off-the-shelf CRDTs by providing a suitable lattice instance. State-based CRDTs have a correct merge function, which we can and have directly used for this purpose. Operation-based CRDTs can be systematically converted to state-based CRDTs [52], thus they can be reused, too.

Unified consistency management.   The CALM theorem [21] implies that monotonicity is a necessary requirement for consistency without coordination. To achieve monotonicity, existing state-based CRDT implementations [52] require that operators return a state that is larger than the original one (with respect to a pre-defined order of all possible states) and operation-based CRDTs require operators to be commutative. In contrast, our approach automatically enforces monotonicity of operators by merging their delta result with the current state. Thus, application developers do not need to reason about consistency when designing operators. The potential for introducing consistency bugs is limited to custom merge functions, which we assume to be designed by experts.

To illustrate the positive effects of this, consider again the Counter ARDT in the social media application. We only have to ensure that operators implement the intended application semantics, but we are always guaranteed consistent results. That is, the developer may make a mistake and the increment operator does not increment the value of the counter, as it is supposed to do. But it is guaranteed that the operator exhibits the same (erroneous) behavior on all replicas. This is in contrast to classic CRDTs, where an incorrect operator may lead to different states on different replicas. Due to unified consistency, distributed correctness boils down to correctness of a single replica, i.e., we get along with local reasoning, which simplifies development and testing.

Finally, since correctness relies exclusively on the properties of the merge functions, reasoning about consistency and ensuring it automatically is greatly simplified. An indication for this are the proofs (in the Appendix) for generated merge functions presented in Subsubsection 3.2.2. First, they are of manageable size. Second and more importantly, one can prove the correctness of individual merge functions independently of other merge functions and operators, because they do not rely on any global assumptions. Correctness for all composed data types then follows automatically from the individual proofs.

Versatile message dissemination.   Local-first applications may run on diverse communication infrastructures, especially when considering various potential intermediaries ranging from a centralized server, to a shared network disk, to passing data along multiple ad-hoc Wi-Fi connections, to storing messages on a USB drive and sending the latter via physical mail.[6] Even though concrete strategies for message dissemination are not a focus of our

---

[6] Networks, where messages are not exchanged directly, but rather stored and forwarded until they are eventually received, are called delay-tolerant networks (DTN) [5]). They are actively developed and researched to enable resilient communication [53, 48, 5, 6], a highly relevant area for local-first software.

contributions, our assumptions about message dissemination are explicitly designed to admit many different such strategies. Moreover, the separation of message dissemination from ARDTs enables independent improvement of separate concerns. Specifically, in Section 4, we thoroughly explore secure communication that works in any setting. On the other side of the spectrum, in Section 6, we also explore possible optimizations of message dissemination in less challenging environments such as a central server.

Limitations.   According to the CALM theorem [21], coordination-free consistent replication schemes can only express algorithms that do not require consensus. This is true for ARDTs, too. Even though we support arbitrary code to express operators, the deltas produced by operators are merged back into the current state, which enforces that the actual change to the state is monotonic. For example, a decrement operation on the counter ARDT (Figure 3) could produce a delta that decrements one of the integer values in the counter. However, because merging integers (Figure 6) returns the maximum, such a delta has no effect when merged into the current value.

4   Encrypting ARDTs

The design of ARDTs is motivated by the need for a flexible encryption mechanism suited for local-first applications. In particular, encryption should be independent of the message dissemination mechanism to provide the same guarantees in any network scenario. Moreover, the encryption should enable efficient storage of encrypted data on untrusted intermediaries.

Our solution provides encryption as a special kind of ARDTs, called *encrypting ARDTs* (encARDTs in the middle of Figure 1). EncARDTs are normal ARDTs for all purposes – they can be replicated using any message dissemination mechanism, and they can be parts of composed ARDTs. EncARDTs provide encryption via their operators, specifically, they implement the message dissemination API (send and recombine) where sending encrypts and recombine decrypts the state.

For example, a naive implementation of an encARDT is shown in Figure 9. Line 41 defines the state of the encARTD as a set of encrypted values. For encryption, we rely on *authenticated encryption with associated data* (AEAD) to ensure confidentiality of the state and integrity of both the state and the metadata. There are multiple encryption primitives that provide AEAD, and we elaborate on the challenge of correct use of AEAD in a coordination-free setting in Section 5. For now, we assume that there exists a suitable AEAD module with the following interface.

```
39  def encrypt[S, A](data: S, meta: A, key: Secret): AEAD[S, A]
40  def decrypt[S, A](aead: AEAD[S, A], key: Secret): Option[(S, A)]
```

The naive encARDT in Figure 9 stores values of type AEAD. We generally refer to encrypted states as *messages* to distinguish them from the state of the encARDT itself. The send operator (Line 42) adds new messages into the encARDT, by using the encrypt method of the AEAD module, and producing a delta containing the message. This delta is handled as usual, i.e., it is merged into the current state using the automatically derived merge function. The recombine operator (Line 44) reconstructs the plaintext ARDT state of type S. To do so, all messages are processed by the decrypt method (Line 45), whose return value for authentication failures makes flatMap discard that message. The successfully decrypted messages are merged pairwise using the lattice of the plaintext ARDT Lattice[S].

```
41  case class Naive[S](messages: Set[AEAD[S, Unit]]):
42    def send(data: S, key: Secret, rID: ReplicaID): Naive[S] =
43      Naive(Set(encrypt(data, (), key)))
44    def recombine(key: Secret)(using Lattice[S]): Option[S]=
45      messages.flatMap(aead => decrypt(aead, key)).map(_.data)
46      .reduceOption(Lattice[S].merge)
```

Figure 9 Naive encARDT stores all states.

```
47  case class Subsuming[S](messages: Set[AEAD[S, Version]]):
48    def version: Version = messages.map(_.metadata)
49      .reduceOption(Lattice.merge[Version])
50      .getOrElse(Version.zero)
51    def send(data: S, key: Secret, replicaID: ReplicaID) =
52      val cause = version merge version.inc(replicaID)
53      Set(encrypt(recombine(key) merge data, cause, key))
54
55  given [S]: Lattice[Subsuming[S]] with
56    def merge(left: Subsuming[S], right: Subsuming[S]):Subsuming[S]=
57      val combined = left union right
58      combined.filterNot(s =>
59        combined.exists(o => s.metadata < o.metadata))
60
61  extension [S] (c: Subsuming[S])
```

Figure 10 Subsuming encARDT based on version data.

Consistency of the naive encARDT directly follows from the automatic construction of the merge function, because we only ever add new messages. However, storing all messages forever is a naive solution, because the state grows indefinitely. Yet, the naive encARDT represents the realistic case where an intermediary has no further information about encrypted messages. The following sections describe how to fix the indefinite growth of required space by using associated metadata.

4.1 Pruning Subsumed States

The naive encARDT stores messages even if they are no longer relevant. As an example why this is problematic, consider the counter ARDT. The counter stores an integer per replica ID, each time a counter is incremented we store the new value and no longer need the old value for that replica ID. In such cases, we say that the old state is *subsumed* by the new state. Formally, a state s' subsumes another state s, if s' contains all updates of s, i.e., $s \sqcup s' = s'$ (where $\sqcup$ is the merge function).

The *subsuming encARDT* attaches logical timestamps [28] in the form of *version vectors* [10] to messages as associated metadata. Version vector metadata provides an order $\leq$ on encrypted states $e(s)$ that implies subsumption: $e(s) \leq e(s') \implies s \sqcup s' = s'$. This allows intermediaries to remove subsumed messages without inspecting their contents. Figure 10 shows the implementation of the subsuming encARDT, whose messages include the Version as associated metadata for the encrypted states. A version vector is semantically a counter CRDT, and Version uses the implementation from Figure 3, but is renamed to reflect its use within the subsuming encARDT.

```
63  case class Dotted[S](messages: Set[AEAD[S, (Dot, Set[Dot])]]):
64    def send(data: S, key: Secret, replicaID: ReplicaID) =
65      val cont = messages.flatMap(aead => decrypt(aead, key))
66      val sub = cont.filter(s =>
67        Lattice[S].merge(s.data, data) == data)
68          .flatMap(s => dotsIn(s.metadata)).toSet
69      Set(encrypt(data, (Dots.next(replicaID), sub), key))
```

■ **Figure 11** Dotted encARDT – precise subsumption.

The operators of the subsuming encARDT automatically add the correct metadata to messages. The helper method version (Line 48) merges all currently stored versions, thus returning the upper bound of all versions. The send operator increments the upper bound of versions (Line 52), implying that the new message subsumes all existing messages, but is not subsumed by any of them. To ensure this is true, the delta state to be send (data in Line 53) is merged with all current values in the encARDT, thus producing a full state that does contain all others. The recombine operator is the same as for the naive encARDT in Figure 9, hence not shown.

An explicit lattice instance implements subsumption as part of the merge function (Line 55). After computing the union of the sets of encrypted states (Line 57), the merge keeps only the states that are not subsumed by another state (Line 58); formally the kept states are $\{e(s)|\nexists e(s') : e(s) < e(s')\}$.

For an intuition to how a subsuming encARDT behaves, consider that a message subsumes all messages that are currently stored in the encARDT, and the merge function removes all subsumed messages. Thus, each time a replica sends a message, only that message (containing all deltas) is stored. However, when multiple untrusted intermediaries synchronize between each other, each may store multiple incomparable messages (generated by different replicas), and merging will keep all of these messages until a trusted replica decrypts and merges them.

In Appendix A.5, we prove that the subsuming encARDT is transparent, i.e., sending and recombining behaves as if we just merge states without encryption, and without removing them based on subsumption metadata. This proof includes correctness of the custom merge function (associative, commutative, idempotent).

4.2 Pruning Encrypted Deltas

With the subsuming encARDT, we lose the advantages of delta replication, because it combines all deltas into a single state when sending a message. To address the problem, *dotted encARDT* in Figure 11 store precise per-delta subsumption information in the metadata. Specifically, the metadata contains (a) a globally unique logical timestamp for the message, called a *dot* [40], and (b) the set of dots that the message subsumes. The send operator computes the set of messages currently contained (cont in Line 65) in the dotted encARDT. For each contained message, it uses the merge function (Line 67) to check if it is subsumed by the new message. Subsumption is transitive, thus the new subsumption info combines all dots in the metadata of all subsumed messages (sub in Line 68). Finally, the message containing only the delta (data) and subsumption info is returned (Line 69). The other methods (including the merge function) of the dotted encARDT are the same as for the subsuming encARDT, thus are not shown.

With dotted encARDTs, intermediaries still cannot decide if two concurrent messages from different trusted replicas subsume one another, but the messages are potentially much smaller compared to the subsuming encARDT. However, dotted encARDTs require more metadata, thus use more space when no concurrent messages occur. See Section 6 for an empirical evaluation, but keep in mind that the best choice is highly dependent on the used ARDTs and application behavior. It is possible to use different encARDTs for different ARDTs in the application, thus providing maximum flexibility. Correctness proofs for the dotted encARDTs are analogous to the subsuming encARDT, because using a more precise notion of subsumption only strengthens the preconditions of the proof.

4.3 Security Assessment

We have presented three different encARDT strategies that cover different points in the design space. Assuming that only trusted replicas know the shared secret, and that AEAD protects data confidentiality and authenticity of the contained messages, all encARDTs prevent the following attacks. Intermediaries cannot tamper with the order of data, because recombination is order independent. Replay attacks using duplicated messages also have no effect, since merging is idempotent. Intermediaries can forge new messages using incorrect keys, but these are ignored when decrypting. The only way for intermediaries to interfere is to selectively stop disseminating messages to (some or all) replicas – this is not worse than the scenario where the intermediary did not exist.

Encrypted communication may still leak information, e.g., the size of messages, and who sends which message at what time. In addition, different encARDTs have different tradeoffs. The naive encARDT leaks no metadata, but stores unneeded messages. The dotted encARDT is as precise as possible, but also leaks precise subsumption metadata. Subsuming encARDTs are in the middle. They leak the order of messages, which can be learned anyway by an attacker that can observe the overall network (a common threat model), while still enabling to remove unneeded messages.

Leaking metadata is considered as unproblematic when synchronizing rich data such as texts and images, because the contained data is not deducible by the order in which modifications happened. But it can be problematic for certain simple ARDTs, e.g., in the case of Counter (Figure 3), which has a single operation, one can deduce the current values by learning the number of messages. But these issues are not unique to our solution, and countermeasures exist [20, 56]. Moreover, because encARDTs do not require a central entity, it becomes easier to apply countermeasures. For example, one can split messages over multiple intermediaries (hence, no single intermediary may learn all metadata), or can use randomized routing such as TOR [13], because ARDTs are resilient against unreliable message delivery.

5 Coordination-free Encrypted ARDTs

The discussion in the previous section leaves out one open challenge: AEAD primitives require each call to the encrypt method to use a globally unique number (nonce). In general, ensuring global uniqueness of something requires coordination, which contradicts our goal to support coordination-free synchronization. Thus, the open question is how to guarantee global uniqueness while practically avoiding coordination. To answer this question, we are the first to analyze multiple stochastic methods of selecting unique numbers for their suitability for the local-first setting.

	Java	Web	libsodium	Tink
AES-GCM	•	•	•	•
AES-GCM-SIV				•
ChaCha20-Poly1305	•		•	•
XChaCha20-Poly1305			•	•

Figure 12 Overview of supported AEAD modes in various environments.

5.1   The Study Setup

Existing solutions vary in different respects: their availability, the number of replicas that are securely supported, the number of operations that can be executed without coordination. The goal of our analysis is to delimit the chances for conflicts within common security standards.

We considered the following AEAD constructions: AES-GCM, AES-GCM-SIV, and (X)ChaCha20-Poly1305. Figure 12 shows their availability in the Java Cryptography Architecture[7], Web Cryptography API[8], libsodium[9], and Tink[10]. All libraries support AES-GCM due to its use in the TLS specification [43]. The more modern AEAD construction ChaCha20-Poly1305 was introduced in TLS 1.3 [43] and is currently also supported by all libraries except Web Cryptography API. XChaCha20-Poly1305 [3] is an adaption of ChaCha20-Poly1305 with a larger nonce-size and proven to be at least as secure [7]; while not yet standardized by IETF, it is supported by libsodium and Tink [3]. AES-GCM-SIV [19] (implemented only in Tink) claims resistance to nonce reuse; it is also not standardized.

In summary, the best available options are:

AES-GCM Use a 64 bit random ID per replica and 32 bit replica specific counter as nonces. Supports up to 92,000 replicas, communicating once per second for 132 years.

XChaCha20-Poly1305 Use fully random 192 bit nonces. Supports 2^{32} replicas for communicating once per millisecond for 8900 years.

Our implementation defaults to XChaCha20-Poly1305, because it allows to completely hide the use of nonces from the developer. In the following, we elaborate on how we reached the conclusion that the above solutions are the best available options.

5.2   Coordination-free Generation of Nonces for AEAD

To encrypt and authenticate a message, AEAD schemes generally require three inputs: the message, the encryption key, and a *nonce* [45]. A nonce is a **n**umber that must only be used **once** together with the same key. If a nonce is used multiple times, then encryption schemes leak information about the plaintext, e.g., in AES, an attacker learns the bitwise exclusive-or of messages [31]. The Nonce misuse has lead to severe real-world attacks, e.g., on TLS [9] and WPA2 [57]. The issue is that the decision on how to choose nonces is left to the developer, and, unfortunately, previous research on crypto misuses has shown that developers struggle with secure choices for crypto APIs [27, 36, 41]. This is not surprising, considering that libraries like the Web Cryptography API do not even document that nonces should be unique. Ensuring uniqueness is a classical coordination problem. Thus, we discuss how to select unique nonces without coordination, while staying within generally accepted levels of certainty for the provided confidentiality.

---

[7] https://docs.oracle.com/en/java/javase/16/security/

[8] https://www.w3.org/TR/WebCryptoAPI/

[9] https://libsodium.org/

[10] https://developers.google.com/tink

5.2.1 Selecting Nonces by Space Partitioning

A textbook approach to ensure uniqueness of nonces is using a strictly monotonic counter [14]. This is the case for AEAD algorithms in TLS 1.3 [43]. Using a single counter for all replicas is not possible without coordination, since this is a prime example of mutual exclusion. An adaption of the counter approach is to partition the nonce space into multiple ranges, each exclusive to a single replica. This strategy requires coordination only once per replica. Fixed ranges are a good choice for a set of devices provided by a single authority (a single user or company). In large groups of loosely cooperating devices, however, deterministic coordination of non-overlapping nonce ranges is infeasible. An alternative approach is *cryptographically secure pseudorandom number generation* (CSPRNG).

Using replica IDs for partitioning. Certain ARDTs such as the counter (Figure 3) already require replica-specific IDs for their behavior. Therefore, it seems intuitive to reuse the replica ID to partition the space of nonces. If the chance of collisions of any two replica IDs is small enough to be negligible, this is a secure choice. In general, to ensure uniqueness, typical examples for replica-IDs are randomly generated UUIDs (as seen in automerge[11]), or a hash of a replica-specific public-key [24], with the size of such identifiers usually being 128 bits [29]. Unfortunately, this size is too large for use with popular AEAD constructions. For example, the NIST specification for AES-GCM recommends that implementations should restrict their nonce lengths in AES-GCM to 96 bits [14]. Thus, direct use of replica IDs to partition the nonce space is not possible.

Using small random replica-specific numbers for partitioning. Instead of using the replica ID, we can generate short replica-specific numbers using a CSPRNG, but this leaves us with a probability of collisions of replica-specific numbers, thus a collision of nonces. According to the NIST specification, the probability that a nonce is reused for a given key must be less or equal to 2^{-32} [14]. Considering the birthday paradox [49], there is a surprisingly high probability that two replicas choose the same replica-specific number. With a 64-bit long replica-specific number, we can have 92,000 replicas before the collision probability reaches over 2^{-32}. Assuming 92,000 replicas are sufficient, and given the explicit 96-bit nonces of AES-GCM, a 64-bit replica-specific number leaves room for 32-bit replica-specific counters. This provides $2^{32} \approx 4.3 \times 10^9$ messages to each replica. Assuming that a replica encrypts one message per second, the counter could be used for 136 years, before requiring coordination to select a new shared secret. This is a realistic choice for local-first applications, when only AES-GCM is available.

5.2.2 Selecting Fully Random Nonces

A fully coordination-free approach to nonce generation is to rely on a CSPRNG to generate a new random nonce for each message. Literature warns against random nonces in some cases [9]. For example, nonces in TLS (using AES-GCM) consist of 32-bit part specific to the sender and connection, and a 64-bit part to ensure uniqueness [46]. With 64-bit random nonces the collision probability after encrypting $2^{28} \approx 2.7 \times 10^8$ (three hundred million) messages would be around 0.2 % and for $2^{32} \approx 4.3 \times 10^9$ messages around 39 % [9].

---

[11] https://github.com/automerge/automerge

For using 96-bit random nonces with AES-GCM, the libsodium documentation recommends against it [30], while the documentation of Tink recommends it for "most uses" [17]. Specifically, Tink guarantees that AES-GCM with random nonces can be used for at least $2^{32} \approx 4.3 \times 10^9$ messages, while keeping the attack probability smaller than 2^{-32} [17].

This, however, is a global message limit, i.e., counting all messages encrypted by all replicas using the same key. The only way to enforce this limit without coordination is to restrict the number of distinct messages to $\frac{2^{32}}{n}$, where n is the maximum number of replicas that can use a single key. Thus, further limiting the number of encrypted messages. Assuming 1024 as an upper bound on the number of replicas, this leaves $\frac{2^{32}}{1024} = 2^{22} \approx 4.2 \times 10^6$ messages to each replica. Or, in other words, 7 weeks of coordination-free operation using one outbound message per second for each replica. Moreover, enforcing a limit on the number of replicas also requires coordination.

Fortunately, random nonces become practical with the very large nonce sizes supported by XChaCha20-Poly1305 [3]. The use of 192-bit nonces allows $2^{80} (\approx 10^{24})$ messages to be encrypted with a nonce collision probability of 2^{-32} [3]. Putting this in context: If every possible of the 2^{32} IPv4 devices is encrypting messages at the rate of one message per *milli*second, this leaves us with over 8900 years before we must rotate keys. Therefore, with random nonces, we only recommend XChaCha20-Poly1305.

5.2.3   Nonce Misuse-resistant AEAD Schemes

Nonce misuse-resistant authenticated encryption schemes, such as AES-GCM-SIV [19], aim to be secure even when a nonce is reused for the same key with a different message. Thus, in theory, they are good candidates for use with shared, long-lived keys. But these schemes also do have bounds on the number of messages that can be safely sent [22]. Moreover, they are not yet standardized and fully scrutinized, and, as discussed in Figure 12, an implementation of AES-GCM-SIV is not widely available; thus, we can not give clear recommendations.

6   Evaluation

In our interim qualitative assessments, our focus was on design considerations in Section 3 (e.g., reusability, flexibility, correctness) and security guarantees in Section 4 and Section 5. The question remains: What is the cost of the properties of our approach featured in the qualitative assessments. To assess this cost, we empirically evaluate our approach along the following research questions:

- RQ1: Is the performance of ARDTs – including encryption – good enough for use in local-first applications?
- RQ2: Are the space requirements of ARDTs using intermediaries acceptable?

We use two methods to explore each of these questions: A concrete case study that makes specific choices about the used ARDTs and a set of microbenchmarks that explore encARDTs more generally to uncover their behavior in multiple dimensions, in particular the overhead caused by encryption and intermediaries. The implementation is part of the REScala project, and all implementations and benchmarks – in addition to further case studies that explore different scenarios – can be found on the project website[12]. The case study evaluated here runs on the JVM (using a JavaFX UI) due to the generally better availability of tooling for

---

[12] https://www.rescala-lang.com

empirical evaluation, but our approach works for both the JVM and on the Web platform (integrating with various Scala-based Web UI frameworks). Unless otherwise noted, we use the following hardware and software setup for this evaluation.

CPU 2015 Intel Core i7-6700HQ (laptop CPUs are the most common for local-first software).

OS Arch Linux (Linux 5.16.16).

JRE We use the Java runtime OpenJDK 17.0.3.

Microbenchmarks For performance microbenchmarks, we use JMH[13] the standard Java benchmarking tool. The time measurements we conduct have very stable runtime behavior, with a maximum relative error of 3%, thus we do not show error bars.

Libraries AEAD implementations are provided by Tink 1.6.1[14], which uses hardware acceleration for AES variants, but not for XChaCha20-Poly1305. To serialize states, we use jsoniter-scala[15] (the arguably fastest JSON serializer available on the JVM[16]).

6.1 Case Study

We implement the popular to-do list example as a JavaFX GUI application. The application manages a list of to-dos, and the user may add entries containing arbitrary text, mark to-dos as completed, change their text, or delete to-dos completely. Its correctness and consistency properties are: added to-dos remain until deleted, and all users see the same to-dos in the same order. The interactions and properties of the case study touch on most of the complexity in the design space of local-first applications. Furthermore, the state of the to-do list – a potentially ordered set of changeable entries – is complex enough to demonstrate the need for composed data types.

We experienced no limitations in implementing the to-do list application with ARDTs and encARDTs. The prototype makes heavy use of the composability of ARDTs. Concretely, the to-do list uses an add-wins last-writer-wins map for its primary state. This is a composition out of a tombstone-free add-wins set [8] and a last-writer-wins register. When two users edit the same to-do entry, a deterministic decision keeps one of the edits and the other is discarded. Changes to the primary state are normally triggered by the UI library (e.g., a button click handler), but the UI is replaced by our benchmark infrastructure. The handlers for each change are similar to the example in Figure 5. Each handler uses a corresponding operator on the to-do entries (the add-wins-last-writer-wins map) to compute the delta of the new application state. The delta is passed to the send operator of the dotted encARDT, and the operator computes its own delta that is in turn passed to the message dissemination implementation (a custom one for benchmarking the transferred data). In addition, there is a notification API (not discussed in the paper) in the message dissemination module that executes a handler whenever a change happens (caused locally or remotely), which triggers the UI to update and show the new state.

To answer our research questions, we run a deterministic simulation of the to-do list. Our simulation uses a single intermediary and simulates a total of one million operations that add, modify, and remove to-do entries (see Subsection 6.2 for a discussion of concurrent operations and multiple intermediaries). A million operations correspond to about 11 days of usage, with an interaction per second. We include serialization, encryption, and other application logic in the simulation. We omit physical network, storing the state on disk, or rendering the

---

[13] https://openjdk.java.net/projects/code-tools/jmh/
[14] https://developers.google.com/tink
[15] https://github.com/plokhotnyuk/jsoniter-scala
[16] https://plokhotnyuk.github.io/jsoniter-scala/

graphical UI, as their performance is not part of our contributions. The simulation follows a randomly generated trace of operations: adding to-dos, marking to-dos as completed, and deleting batches of the 30 oldest to-dos. To-dos are added and completed individually, but deleted in batches to reflect the expected usage of the application, which has a "remove all completed to-dos" button, but no methods of batch insertion or completion.
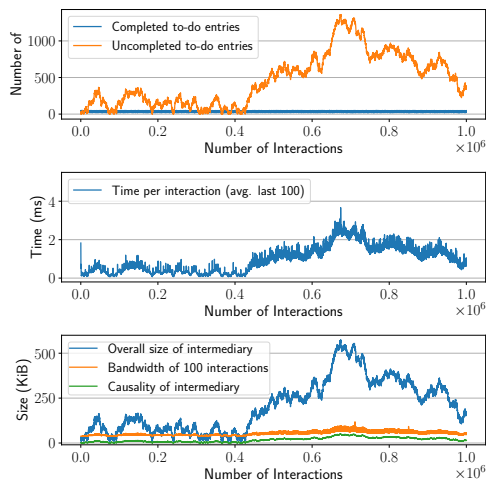
The top plot of Figure 13 shows the runtime behavior of the simulation. The x-axis represents abstract time as the number of executed interactions and the graphs show the respective state of the application, i.e., the number of open and completed to-do entries.

RQ1. Time overhead is presented in Figure 13 (middle plot). It shows the runtime per interaction (measured in batches of 100 interactions). This time includes executing the operator locally, merging it into the local state, serializing then encrypting and sending the delta, merging the encrypted delta into the encARDTs thus computing subsumption, and replicating the encARDTs to the intermediary. The spike in the beginning is due to the warm-up of the JVM. Otherwise, the overall runtime is proportional to the size of the current application state, because tasks – such as merging the add-wins-map, computing subsumption for existing deltas, and the application logic – linearly depend on the number of to-do entries. We believe that staying within 3 ms per operation is reasonable. While further optimizations are certainly possible, there is no indication that our core architecture has prohibitive costs for local-first applications.
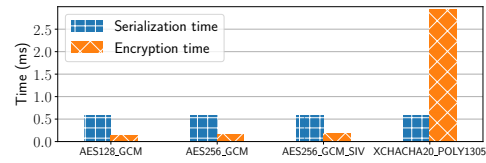
RQ2. Space overhead is presented in Figure 13 (bottom). Note that we show the accumulated bandwidth of 100 interactions (i.e., 100 deltas), because the size would otherwise not be visible at the scale of the figure. In summary, we observe that the total data stored at the intermediary has a linear relation to the actual size of the application state and grows and shrinks accordingly. We want to point out that neither the state, nor the causality metadata increases over time. While encARDTs require that we store information about subsumed deltas indefinitely (the set of subsumed dots), it is stored as efficient ranges that only grow with the number of replicas, concurrent operations, and current size of the data set, but not with the number of total interactions over time. The data transferred (used bandwidth) between the replica and the intermediary remains mostly constant because transfer time depends on the size of deltas, which are largely unaffected by the size of the application state. The slight increase in bandwidth is because each removal delta includes causality information in the encARDT that grows with the amount of currently non-removed entries. We show the difference to using a trusted intermediary (i.e., no encARDT) in Appendix A.6, which requires less space due to the impact of encrypted deltas discussed in Subsection 6.2. In conclusion, we consider the size demand and required bandwidth of ARDTs adequate for the local-first scenario.

6.2 Microbenchmarks

We perform microbenchmarks to acquire data points that the case study does not exhibit. Specifically, we investigate the isolated overhead of encryption as well as the effect of concurrent operations (e.g., due to multiple intermediaries) on the required storage size. We use the same add-wins last-writer-wins map (AWLWWMap) that was used for the to-do list case study. Yet, the results are independent from the concrete choice of ARDT, because for the microbenchmarks only the serialized size of the state matters, which we give explicitly.

Figure 14 Encryption vs. serialization time for AWLWWMap states of 256 KB.

Figure 13 To-do list case study measurement results.
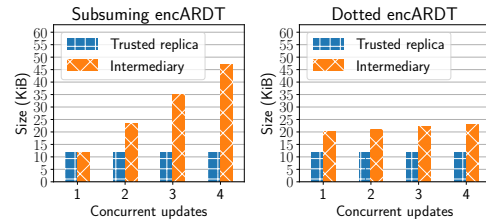
Figure 15 State size when storing concurrent encrypted messages.

RQ1: Time overhead of encARDTs. We measure how long it takes to prepare the serialized bytes compared to the time for encrypting those bytes via an encARDT. The results in Figure 14 show the difference (encryption time vs. serialization time) for different AEAD schemes, and for a payload of 256 KiB (1,000 to-do entries). Hardware accelerated AES has an overhead of a fraction of a millisecond. XChaCha20-Poly1305 is designed to be efficiently implemented in software [3], thus should be considered for systems where no hardware accelerated encryption is available. Even if it does not benefit from hardware acceleration it has an overhead of less than 3 ms.

To put these numbers into context, consider the relative sizes of operations. The full state of a to-do list with 1000 to-do entries serializes into a 256 KiB state. The dotted encARDT requires an additional 0.16 ms for encryption. The serialization of the state alone takes 0.67 ms. Sending data over the network has expected latencies of 0.1~100 ms. Receiving and processing the data on the other replica and displaying the result adds a minimum of 7~33 ms due to typical refresh rates of monitors. In summary, we consider the time overhead of encARDTs to be negligible compared to all other parts of the synchronization process.

RQ2: Space overhead of encARDTs. Intermediaries cannot merge states that are created concurrently by different replicas, due to the limitations of the plaintext metadata. The overhead depends on the encARDT and the number concurrent operations. Figure 15 shows the space requirement of storing 1 to 4 concurrent updates using a subsuming encARDT (left) and a dotted encARDT (right). The base size of the stored ARDT is an AWLWWMap with 96 (+1 to +4 added) entries requiring about 14 KiB. Any trusted replica (in blue) can always merge any received updates, thus the total stored size does not grow noticeably.

For the intermediary, however, we observe that the size of subsuming encARDT (left subfigure) grows linearly with each concurrent update. We expected this result as each update contains the full state to be stored, and the timestamps of the four updates are incomparable, because each full state differs in exactly one item, thus they do not subsume each other. For the dotted encARDT (right subfigure), the stored state is larger than the state of the trusted replica, because each delta is stored separately, which introduces a constant overhead per

delta. However, each concurrent update only marginally increases the state size to store the single additional delta. Note that the number of concurrent operations is typically limited by the number intermediaries, because once a replica is connected to an intermediary the next operations of the replica will merge and subsume concurrent operations.

In general, storing only deltas at intermediaries has a fixed overhead, but avoids large storage increases for concurrent updates. Which strategy is more suitable depends on how reliable connections are, and how many intermediaries are part of the system, because both unreliability and more intermediaries introduce more concurrency. In summary, we believe that a wide range of potential use cases are covered by the presented encARDTs. If other behavior is required, new variants of encARDTs with different subsumption strategies can be used.

7 Related Work

Programming methods for local-first software. Two popular general-purpose CRDT implementations that can be integrated into applications are automerge[17] (loosely based on a paper by Kleppmann et al. [24]) and Yjs[18] [37]. Both libraries are based on the operation-based variant of CRDTs. They run in the same process as the application and provide the latter with an API to update and query a single JSON document (a nested tree structure). The intended way to use the API is to have developers convert their application state into the JSON structure, with no further customization of available operations.

REScala [34] provides programming support for local-first applications. It integrates off-the-shelf CRDTs with functional reactive programming, the latter being a very common approach to UI and thus local-first applications. The rationale for the integration is that reactive applications are practically not limited by the monotonicity restriction of CRDTs, because user interactions are monotonic by nature: Users can only press, click, and touch keys and buttons and not "unpress" a prior action. However, REScala assumes that the developer provides CRDTs with suitable operations, and does not consider encryption [33, 35].

Similar to our work, other systems for local-first software consider message dissemination as an orthogonal concern that depends on the concrete network environment. Yjs, and automerge, provide default implementations to be ready to use, while REScala defers to ScalaLoci [58] – a library that abstracts over communication implementations. None of the approaches considers network security beyond encrypted direct connections. Almeida et al. and Enes et al. [2, 15] show how to achieve causal consistency for delta CRDTs independently of the underlying network. This is done by providing an additional layer on top of another message dissemination algorithm. While they do not consider security, their approach is similar in that they separate different concerns in the message dissemination.

Security in replicated systems. Preguiça et al. [39] survey CRDTs in the geo-distributed setting and point out the need for future research on security. They observe that replicas are vulnerable to harmful operations of other replicas, and that authentication and encryption between replicas is insufficient in the geo-distributed setting. This is because the trusted entities in that setting are the clients (end users) that issue operations to a replica (in the cloud). They conclude that end-to-end security between clients is (nearly) impossible in the existing architecture, and argue for "moving computations to the edge". Moving computation to the edge, while replicating state in the cloud, is exactly what ARDTs enable.

---

[17] https://github.com/automerge/automerge
[18] https://github.com/yjs/yjs

Barbosa et al. [4] implement an approach that keeps the client/replica split while proving some guarantees to clients. The clients use customized solutions from the space of homomorphic encryption to secure their data before storing it in a distributed database (AntidoteDB [50]), which then handles replication. This approach has the major shortcoming that all the cryptographic constructions are specific to individual CRDTs. Moreover, they only target *honest-but-curious* adversaries, which is an assumption where the attacker is bound to service-level agreements, and only interested in secretly extracting information (i.e., a cloud service provider hosting the database). Crucially, this entails that an adversarial provider could modify data, because operations cannot be authenticated.

High-level cryptographic APIs. We do believe that encARDTs offer an advantage even when used with simple direct connections. Security is often treated as an afterthought, and it has been shown that leaving this task of using crypto solutions to application developers often leads to insecure systems [16, 38]. The correct usage of cryptographic components is challenging in general [36], with 84 % of Apache projects containing cryptographic misuses [41]. Especially developers of end-user applications seem to have a hard time, with more than 95 % of android applications that use a cryptographic API using it incorrectly [27]. High-level abstractions with built-in cryptographic features are considered as an effective solution to support developers with writing secure software [1, 18, 32]. With encARDTs, we bring high-level cryptographic APIs to local-first applications, thus reducing the potential for misuse. In addition, encARDTs define encryption of data structures, not encryption of connections, which is better suited to the flexible dynamic connections of local-first applications.

Identities and attackers. Security and authentication of our approach require a shared secret between trusted parties. If secrets are shared with untrusted parties, our approach does not provide additional guarantees. Sanjuan et al. [47] and Kleppmann [23] investigate settings where other replicas are not trusted. They argue that CRDTs are well suited to detect Byzantine faults at the eventual consistency layer. Specifically, by including cryptographic hashes of the causal history of each change, it is possible to discard and detect messages from misbehaving replicas. We believe that these solutions also apply to ARDTs due to their similarity to CRDTs. However, even when using these solutions an attacker may still execute consistent but undesirable actions, and is able to read the system state.

To prevent undesirable actions, we need a way to manage and enforce access policies over time. EncARDTs use a shared secret to define the current set of trusted replicas, and it is possible to rotate this key to change the set of trusted replicas. Rault et al. [42] propose how to manage access control itself as a CRDT, thus answering the question of who should have access to the shared secret. Truong et al. [55] discuss authentication of the log of operations in an RDT, which allows replicas to identify and attribute tampering carried out by replicas with full access. Kollmann et al. [26] propose a solution to compress such authentication information within a snapshot of an RDT. This also allows them to keep the exact history of changes hidden from newly joined replicas by leveraging coordination-free authentication of snapshots. We believe that these approaches could be adapted for the use with ARDTs.

8   Conclusions and Future Work

Local-first applications address several weaknesses of a centralized software architecture. But designing applications with consistent replication is challenging for application developers, because it requires expertise in several system-level concerns such as consistency, networking,

and security protocols. Existing solutions such as CRDTs provide consistency "out-of-the-box", but have several shortcomings otherwise. First, they do not integrate well into the application design process: Application developers have to map application-specific data models to CRDTs, which are only available "off-the-shelf" in the form of databases [50] or libraries with a fixed API [24, 37]. Designing application state based on a fixed set operations is known to cause design issues [11]. Second, off-the-shelf systems do not support heterogeneous network environments, and authenticity and confidentiality is considered as afterthought at best.

The foundation of our solution to the above gaps is an approach for systematic, modular, and extensible design of algebraic replicated data types (ARDTs). The approach provides the same guarantees as CRDTs, but as a modular and extensible library that embraces algebraic data types, which are widely used to model application state. This approach facilitates the integration of ARDTs into existing programming models and existing network runtimes. Further, our solution provides confidentiality and authenticity by design. Specifically, we presented a family of encrypting ARDTs for different network requirements. Each such encARDT wraps around the data of an ARDT and secures the data independently of how messages are disseminated, with specific support to transmit data over untrusted intermediaries. Using our encARDTs, the application data is authenticated and encrypted, while retaining coordination freedom and preventing common misuses of cryptographic primitives. A significant partial result of the above is, that while current AEAD schemes theoretically require coordination due to the uniqueness constraint on the nonces, it is possible to avoid coordination for long enough to make them applicable in a coordination-free setting. Specifically, this result applies to all other approaches for local-first software that could adapt our techniques to encrypt and authenticate their network communication.

Our evaluation shows that we can implement typical local-first applications efficiently and that any ARDT can be securely disseminated. The performance overhead is only a small fraction of the existing dissemination cost. The additional storage requirement is limited by the amount of concurrent changes in the worst case and can be minimized further by including more precise metadata. Moreover, the storage requirement does not increase indefinitely, as ARDTs makes it possible to remove data that is no longer needed by the application logic. Together, the results of the experiments show that it is feasible to use the proposed solution in practice.

A remaining issue – common to all encrypted synchronization techniques – is that it needs to leak metadata to enable efficient dissemination of messages. However, because our approach is resilient to poor network conditions including reordering, delay, and duplication of messages, we believe that many common mitigation techniques can be applied without impeding normal operations. Such mitigations include sending fake data to make metadata less usable, or routing data on multiple intermediaries such that no single one has a full view of the system. We may also be able to apply concepts from homomorphic encryption or secure enclaves to enable intermediaries to learn which states subsume each other, without gaining any further insight into the exact metadata of each message.

Finally, it is noteworthy that besides solving the practical problem of ensuring the integrity and authenticity of replicated state in the presence of untrusted replicas, encARDTs also represent the novel concept of RDT-based implementations of what would classically be seen as a (network) protocol. An encARDT addresses protocol concerns such as transparency of encrypting and decrypting transferred data, which messages are important (and must be retransmitted), and which ones have been superseded by newer messages. Crucially, these concerns are separated from concrete issues concerning physical networks such as message

losses, retries and retransmission delays, or splitting large packages. These concerns are specific to each communication platform and are handled by concrete message dissemination modules. As a future expansion on this concept it could be possible to implement other concerns of network protocols as ARDTs. A concrete example is message delivery in causal order, which can be achieved by attaching ordering information to each message [2]. Such an ARDT would wrap another ARDT, similar to how an encARDT works, but use its operator to present a state merged in causal order (temporarily ignoring messages that were received out of order).

References

1 Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. Comparing the usability of cryptographic apis. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 154–171, 2017. `doi:10.1109/SP.2017.52`.

2 Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162–173, 2018. `doi:10.1016/j.jpdc.2017.08.003`.

3 Scott Arciszewski. XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305. Internet-Draft draft-irtf-cfrg-xchacha-03, Internet Engineering Task Force, 2020. Work in Progress. URL: `https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-xchacha-03`.

4 Manuel Barbosa, Bernardo Ferreira, João Marques, Bernardo Portela, and Nuno Preguiça. Secure conflict-free replicated data types. In *International Conference on Distributed Computing and Networking 2021*, ICDCN '21, pages 6–15, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3427796.3427831`.

5 Lars Baumgärtner, Jonas Höchst, and Tobias Meuser. B-dtn7: Browser-based disruption-tolerant networking via bundle protocol 7. In *2019 International Conference on Information and Communication Technologies for Disaster Management (ICT-DM)*, pages 1–8, 2019. `doi:10.1109/ICT-DM47966.2019.9032944`.

6 Lars Baumgärtner, Patrick Lieser, Julian Zobel, Bastian Bloessl, Ralf Steinmetz, and Mira Mezini. Loragent: A dtn-based location-aware communication system using lora. In *2020 IEEE Global Humanitarian Technology Conference (GHTC)*, pages 1–8, 2020. `doi:10.1109/GHTC46280.2020.9342886`.

7 Daniel J. Bernstein. Extending the salsa20 nonce. In *Workshop Record of Symmetric Key Encryption Workshop 2011*, 2011. URL: `https://cr.yp.to/snuffle/xsalsa-20110204.pdf`.

8 Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. Research Report RR-8083, INRIA, October 2012. URL: `https://inria.hal.science/hal-00738680`.

9 Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. USENIX Association, 2016. URL: `https://www.usenix.org/conference/woot16/workshop-program/presentation/bock`.

10 Russell Brown. Vector clocks revisited, 2015. Online; accessed 18 October 2021. URL: `https://riak.com/posts/technical/vector-clocks-revisited/index.html`.

11 Tse-Hsun Chen, Weiyi Shang, Jinqiu Yang, Ahmed E. Hassan, Michael W. Godfrey, Mohamed N. Nasser, and Parminder Flora. An empirical study on the practice of maintaining object-relational mapping code in java systems. In Miryung Kim, Romain Robbes, and Christian Bird, editors, *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 165–176. ACM, 2016. `doi:10.1145/2901739.2901758`.

12 Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In Michael J. Carey and Steven Hand, editors, *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, page 1. ACM, 2012. `doi:10.1145/2391229.2391230`.

13 Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association. URL: `https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router`.

14 Morris J. Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. Technical report, National Institute of Standards and Technology, 2007. `doi:10.6028/nist.sp.800-38d`.

15 Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero, and João Leitão. Efficient synchronization of state-based crdts. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 148–159. IEEE, 2019. `doi:10.1109/ICDE.2019.00022`.

16 Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 50–61, New York, NY, USA, 2012. Association for Computing Machinery. `doi:10.1145/2382196.2382205`.

17 Google. Authenticated encryption with associated data (aead). Online; accessed 12 October 2021. URL: `https://developers.google.com/tink/aead`.

18 Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016. `doi:10.1109/MSP.2016.111`.

19 Shay Gueron and Yehuda Lindell. GCM-SIV: Full Nonce Misuse-Resistant Authenticated Encryption at Under One Cycle per Byte. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 109–119, New York, NY, USA, October 2015. Association for Computing Machinery. `doi:10.1145/2810103.2813613`.

20 Christoph Hagen, Christian Weinert, Christoph Sendner, Alexandra Dmitrienko, and Thomas Schneider. All the numbers are US: large-scale abuse of contact discovery in mobile messengers. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. `doi:10.14722/ndss.2021.23159`.

21 Joseph M. Hellerstein and Peter Alvaro. Keeping calm: When distributed consistency is easy. *Commun. ACM*, 63(9):72–81, 2020. `doi:10.1145/3369736`.

22 Antoine Joux. Nonce misuse-resistant authenticated encryption, 2019. `doi:10.17487/RFC8452`.

23 Martin Kleppmann. Making CRDTs Byzantine fault tolerant. In *9th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC 2022, pages 8–15. ACM, April 2022. `doi:10.1145/3517209.3524042`.

24 Martin Kleppmann and Alastair R. Beresford. A conflict-free replicated json datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017. `doi:10.1109/tpds.2017.2697382`.

25 Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, pages 154–178, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3359591.3359737`.

26 Stephan A Kollmann, Martin Kleppmann, and Alastair R Beresford. Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2019(3):210–232, July 2019. `doi:10.2478/popets-2019-0044`.

27 Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An extensible approach to validating the correct usage of cryptographic APIs. *IEEE Transactions on Software Engineering*, 47(11):2382–2400, 2019. `doi:10.1109/TSE.2019.2948910`.

28 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. `doi:10.1145/359545.359563`.

29 Paul J. Leach, Rich Salz, and Michael H. Mealling. A universally unique identifier (uuid) urn namespace. RFC 4122, 2005. `doi:10.17487/RFC4122`.

30   Libsodium Project. Aes256-gcm. Online; accessed 14 October 2021. URL: `https://libsodium.gitbook.io/doc/secret-key_cryptography/aead/aes-256-gcm`.

31   David McGrew. An interface and algorithms for authenticated encryption. RFC 5116, 2008. `doi:10.17487/RFC5116`.

32   Kai Mindermann, Philipp Keck, and Stefan Wagner. How usable are rust cryptography apis? In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 143–154, 2018. `doi:10.1109/QRS.2018.00028`.

33   Ragnar Mogk. *A Programming Paradigm for Reliable Applications in a Decentralized Setting*. PhD thesis, Technische Universität Darmstadt, Darmstadt, March 2021. `doi:10.26083/tuprints-00019403`.

34   Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. Fault-tolerant distributed reactive programming. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 1:1–1:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.ECOOP.2018.1`.

35   Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. A fault-tolerant programming model for distributed interactive applications. *Proc. ACM Program. Lang.*, 3(OOPSLA):144:1–144:29, 2019. `doi:10.1145/3360570`.

36   Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: why do java developers struggle with cryptography APIs? In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 935–946. ACM, 2016. `doi:10.1145/2884781.2884790`.

37   Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Yjs: A framework for near real-time p2p shared editing on arbitrary data types. In Philipp Cimiano, Flavius Frasincar, Geert-Jan Houben, and Daniel Schwabe, editors, *Engineering the Web in the Big Data Era*, pages 675–678. Springer International Publishing, 2015. `doi:10.1007/978-3-319-19890-3_55`.

38   Marten Oltrogge, Nicolas Huaman, Sabrina Amft, Yasemin Acar, Michael Backes, and Sascha Fahl. Why eve and mallory still love android: Revisiting TLS (In)Security in android applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4347–4364. USENIX Association, August 2021. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/oltrogge`.

39   Nuno Preguiça, Carlos Baquero, and Marc Shapiro. Conflict-free replicated data types crdts. In Sherif Sakr and Albert Zomaya, editors, *Encyclopedia of Big Data Technologies*, pages 1–10. Springer International Publishing, 2018. `doi:10.1007/978-3-319-63962-8_185-1`.

40   Nuno Preguiça, Carlos Bauqero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Brief announcement: Efficient causality tracking in distributed storage systems with dotted version vectors. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 335–336, New York, NY, USA, 2012. Association for Computing Machinery. `doi:10.1145/2332432.2332497`.

41   Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3319535.3345659`.

42   Pierre-Antoine Rault, Claudia-Lavinia Ignat, and Olivier Perrin. Distributed access control for collaborative applications using CRDTs. In *9th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC 2022, pages 33–38. ACM, April 2022. `doi:10.1145/3517209.3524826`.

43   Eric Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, 2018. `doi:10.17487/RFC8446`.

44   Phillip Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 98–107, New York, NY, USA, November 2002. Association for Computing Machinery. `doi:10.1145/586110.586125`.

45   Phillip Rogaway. Nonce-based symmetric encryption. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 2004. `doi:10.1007/978-3-540-25937-4_22`.

46   Joseph A. Salowey, David McGrew, and Abhijit Choudhury. Aes galois counter mode (gcm) cipher suites for tls. RFC 5288, 2008. `doi:10.17487/RFC5288`.

47   Hector Sanjuan, Samuli Poyhtari, Pedro Teixeira, and Ioannis Psaras. Merkle-CRDTs: Merkle-DAGs meet CRDTs. *CoRR*, April 2020. `arXiv:2004.00107`.

48   Sebastian Schildt, Tim Lüdtke, Klaus Reinprecht, and Lars Wolf. User study on the feasibility of incentive systems for smartphone-based dtns in smart cities. In *Proceedings of the 2014 ACM International Workshop on Wireless and Mobile Technologies for Smart Cities*, WiMobCity '14, pages 67–76, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2633661.2633662`.

49   Bruce Schneier. *Applied cryptography – Protocols, algorithms, and source code in C, 2nd Edition*. Wiley, 1996. URL: `https://www.worldcat.org/oclc/32311687`.

50   Marc Shapiro, Annette Bieniusa, Nuno M. Preguiça, Valter Balegas, and Christopher Meiklejohn. Just-right consistency: Reconciling availability and safety. *CoRR*, abs/1801.06340, 2018. `doi:arXiv.1801.06340`.

51   Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, 2011. URL: `https://hal.inria.fr/inria-00555588`.

52   Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer Berlin Heidelberg, 2011. `doi:10.1007/978-3-642-24550-3_29`.

53   Milan Stute, Florian Kohnhauser, Lars Baumgartner, Lars Almon, Matthias Hollick, Stefan Katzenbeisser, and Bernd Freisleben. RESCUE: A resilient and secure device-to-device communication framework for emergencies. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2020. `doi:10.1109/TDSC.2020.3036224`.

54   Chengzheng Sun. Reflections on collaborative editing research: From academic curiosity to real-world application. In Weiming Shen, Pedro Antunes, Nguyen Hoang Thuan, Jean-Paul A. Barthès, Junzhou Luo, and Jianming Yong, editors, *21st IEEE International Conference on Computer Supported Cooperative Work in Design, CSCWD 2017, Wellington, New Zealand, April 26-28, 2017*, pages 10–17. IEEE, 2017. `doi:10.1109/CSCWD.2017.8066663`.

55   Hien Thi Thu Truong, Claudia-Lavinia Ignat, and Pascal Molli. Authenticating operation-based history in collaborative systems. In *17th ACM International Conference on Supporting Group Work*, GROUP 2012, pages 131–140. ACM, October 2012. `doi:10.1145/2389176.2389197`.

56   Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 137–152, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2815400.2815417`.

57   Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1313–1328, New York, NY, USA, October 2017. Association for Computing Machinery. `doi:10.1145/3133956.3134027`.

58   Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with scalaloci. *Proc. ACM Program. Lang.*, 2(OOPSLA):129:1–129:30, 2018. `doi:10.1145/3276499`.

```
70   inline def derived[S<:Product](using pm:ProductOf[S]):Lattice[S]=
71     val lattices =
72       summonAll[Tuple.Map[pm.MirroredElemTypes, Lattice]]
73         .toIArray.map(_.asInstanceOf[Lattice[Any]])
74     new Lattice[S]:
75       def merge(left: S, right: S): S = pm.fromProduct(
76         new Product {
77           def productElement(i: Int): Any =
78             lattices(i).merge(left.productElement(i),
79                               right.productElement(i))
80         })
```

Figure 16 Automatic derivation of lattice instances for product types.

A   Appendix

A.1   Map Merge is Correct

Proof. Given that K is the set of all keys (replica ids), x_k is a lookup of key k in map x that returns 0 if the key is not present, $\{k \to v\}_{k \in K}$ constructs a new map that associates the key k to the value v, that m_0 is a correct merge function for the values stored in the map, and $m(x, y) = \{k \to m_0(x_k, y_k)\}_{k \in K}$ is the implementation of the merge function. All three proofs are calculations that first expand the definition of m, then use the respective property of the m_0 function, and finally use the reverse definition of m (except in the idempotence case which is already done).

$$
\begin{aligned}
\text{Commutative:} \quad m(x, y) &= \{k \to m_0(x_k, y_k)\}_{k \in K} \\
&= \{k \to m_0(y_k, x_k)\}_{k \in K} = m(y, x)
\end{aligned} \tag{1}
$$

$$
\begin{aligned}
\text{Associative:} \quad m(m(x, y), z) &= \{k \to m_0(m_0(x_k, y_k), z_k)\}_{k \in K} \\
&= \{k \to m_0(x_k, m_0(y_k, z_k))\}_{k \in K} = m(x, m(y, z))
\end{aligned} \tag{2}
$$

$$
\begin{aligned}
\text{Idempotent:} \quad m(x, x) &= \{k \to m_0(x_k, x_k)\}_{k \in K} \\
&= \{k \to x_k\}_{k \in K} = x
\end{aligned} \tag{3}
$$

◀

A.2   Derived Product Merge Implementation

We elaborate on the technical details of the implementation of method derived in Figure 16. The method is marked inline to make use of compile-time meta programming, which we use to acquire lattice instances of the individual components of the product, specifically, the summonAll method used later. The using keyword asks the compiler to provide a product mirror (named pm) for the product type S to allow inspection of the components of S.

The summonAll method in Line 72, similar to the using keyword, "summons" instances provided by the given keyword based on their types. Specifically, the type we request are the component types (pm.MirroredElemTypes) of our product mapped to the Lattice type. As an example, the component types of our MyData class returns the type (A, B) and mapping

that onto the lattice type results in the type (Lattice[A], Lattice[B]) which is the type for which we "summon" the instances. The result is a tuple of typed lattice instances, but we throw away all type information (Line 73) and rely on the fact that all used products have the same component type at the same structural position.

Having computed lattice instances of the components, we create a new instance of the lattice trait (Line 74). The merge function of that instance (defined in Line 75) uses the pm.fromProduct helper to generically create a new instance of the result product S (e.g., a new instance of our MyData class) in Line 76. The parameter to pm.fromProduct essentially assigns each component at index i (productElement in Line 76) the result of using merge function i to merge the left and right components at position i.

The technical challenges of generating merge functions for arbitrary products are mostly related to practical concerns in the programming language.

A.3 Derived Product Merge is Correct

Proof. Given that K is the set of product indices (this would be the field names of a case class), x_k is a lookup of index k in product x, the syntax $\{k \to v\}_{k \in K}$ constructs a new product of correct type that associates the index k to the value v, each component type at index k has a merge function m_k, and $m(x, y) = \{k \to m_k(x_k, y_k)\}_{k \in K}$ is the implementation of the merge function for the product. We show that m is commutative, associative, and idempotent. All three proofs are calculations that first expand the definition of m, then use the respective property of the component merge functions, and finally use the reverse definition of m (except in the idempotence case which is already done).

$$
\begin{aligned}
\text{Commutative:} \quad m(x, y) &= \{k \to m_k(x_k, y_k)\}_{k \in K} \\
&= \{k \to m_k(y_k, x_k)\}_{k \in K} = m(x, y)
\end{aligned}
\tag{4}
$$

$$
\begin{aligned}
\text{Associative:} \quad m(m(x, y), z) &= \{k \to m_k(m_k(x_k, y_k), z_k)\}_{k \in K} \\
&= \{k \to m_k(x_k, m_k(y_k, z_k)\}_{k \in K} = m(x, m(y, z))
\end{aligned}
\tag{5}
$$

$$
\begin{aligned}
\text{Idempotent:} \quad m(x, x) &= \{k \to m_k(x_k, x_k)\}_{k \in K} \\
&= \{k \to x_k\}_{k \in K} = x
\end{aligned}
\tag{6}
$$

◀

A.4 Naive encARDT is Transparent

Proof. We show that for any subset of states $c \subset S$ sending (encrypting) and recombining (decrypting and merging) the set c is equivalent to merging the set c directly. This uses the secret key k, the merge function m_S for states in S, the merge function $m_e = $ union of the encARDT, the encrypt e_k and decrypt d_k function with $d_k(e_k(s)) = s$, the send function $\text{send}_k(s) = \{e_k(s)\}$, and the recombine function $\text{rec}_k(c) = m_S(\{d_k(s)|s \in c\})$. The proof is done by expanding the above definitions (highlighted in blue) when appropriate.

$$\text{rec}_k(m_e(\{\text{send}_k(s')|s' \in c\}))$$
$$= m_S(\{d_k(s)|s \in m_e(\{\text{send}_k(s')|s' \in c\})\}) \quad \text{def of rec}$$
$$= m_S(\{d_k(s)|s \in m_e(\{\{e_k(s')\}|s' \in c\})\}) \quad \text{def of send}$$
$$= m_S(\{d_k(s)|s \in \{e_k(s')|s' \in c\}\}) \quad \text{def of } m_e \quad (7)$$
$$= m_S(\{d_k(e_k(s))|s \in c\}) \quad \text{simplify set ops}$$
$$= m_S(\{s|s \in c\}) \quad \text{def of } e_k \text{ and } d_k$$
$$= m_S(c)$$

◄

A.5 Subsuming encARDT is Transparent

Proof. Given a secret key k, a subset of states $c \subset S$, individual states s, x, y, z, a merge function m_S for states in S, associated data for each state a_s where $a_x \leq a_y$ if $m_S(x,y) = y$, the filter function $f(c) = \{x \in c | \nexists y \in c : a_x < a_y\}$, the merge function $m_e(c) = f(\bigcup(c))$ of the encARDT, the encrypt e_k the decrypt d_k function with $d_k(e_k(s)) = s$, the current encrypted states c_e, the send function $\text{send}_k(c_e, s) = \{e_k(m_S(\text{rec}_k(c_e), s))\}$, and the recombine function $\text{rec}_k(c_e) = m_S(\{d_k(s)|s \in c_e\})$.

It holds that filtering distributes over union $f(p \cup q) = f(f(p) \cup q)$, because all elements of $f(p)$ are larger or equal to all elements in p, so filtering them out first does not change the result of $f(p \cup q)$.

Filter distributes over decryption, i.e., $\{d_k(s)|s \in f(c)\} = f(\{d_k(s)|s \in c\})$, because filtering is defined on associated data which is also available in the encrypted state.

It holds that filtering is subsumed by merging $m_S(f(c)) = m_S(c)$, because for each removed element $r \in p \setminus f(p)$ it is subsumed by one of the remaining elements $q \in f(p)$ thus merging it again makes no difference $m_S(r,p) = p$.

We first show that the merge function of the encARDT m_e is associative, idempotent, and commutative. Note, that up until now, we have proven a slightly stronger version of idempotence that requires less calculation, but we can not do so here, because the filtering function does not provide the stronger guarantee that $f(a) = a$, thus we only have $m(x,x) = f(x)$. Instead of strong idempotence, we prove that $m(m(x,y),y) = m(x,y)$, that is, merging y multiple times still makes no difference, but we must merge at least once.

$$\text{Commutative:} \quad m_e(x,y) = f(x \cup y) = f(y \cup x) = m_e(y,x) \quad (8)$$

$$\text{Associative:} \quad m_e(m_e(x,y),z) = f(f(x \cup y) \cup z) = f(x \cup y \cup z)$$
$$= f(x \cup f(y \cup z)) = m_e(x, m_e(y,z)) \quad (9)$$

$$\text{Idempotent:} \quad m_e(m_e(x,y),y) = f(f(x \cup y) \cup y) = f(x \cup y \cup y)$$
$$= f(x \cup y) = m_e(x,y) \quad (10)$$

Finally, transparency of the subsuming encARDT, i.e., that receiving a set of send and filtered states is equivalent to merging those states directly. The applied definitions are listed and the changes highlighted in blue.

$$
\begin{aligned}
&\mathrm{rec}_k(m_e(\{\mathrm{send}_k(c_e, s')|s' \in c\})) \\
&= m_S(\{d_k(s)|s \in m_e(\{\mathrm{send}_k(c_e, s')|s' \in c\})\}) && \text{def of rec} \\
&= m_S(\{d_k(s)|s \in m_e(\{\{e_k(m_S(\mathrm{rec}_k(c_e), s'))\}|s' \in c\})\}) && \text{def of send} \\
&= m_S(\{d_k(s)|s \in f(\{e_k(m_S(\mathrm{rec}_k(c_e), s'))|s' \in c\})\}) && \text{def of } m_e \\
&= m_S(f(\{d_k(s)|s \in \{e_k(m_S(\mathrm{rec}_k(c_e), s'))|s' \in c\}\})) && \text{filter distributes} \\
&= m_S(f(\{d_k(e_k(m_S(\mathrm{rec}_k(c_e), s')))|s' \in c\})) && \text{simplify set ops} \\
&= m_S(f(\{m_S(\mathrm{rec}_k(c_e), s')|s' \in c\})) && \text{def of } e_k \text{ and } d_k \\
&= m_S(\{m_S(\mathrm{rec}_k(c_e), s')|s' \in c\}) && \text{filter subsumed} \\
&= m_S(\mathrm{rec}_k(c_e), m_S(c)) && \text{merge properties} \\
&= m_S(m_S(\{d_k(e_k(s))|e_k(s) \in c_e\}), m_S(c)) && \text{def of rec} \\
&= m_S(m_S(\{s|e_k(s) \in c_e\}), m_S(c)) && \text{decrypted} \\
&= m_S(\{s|e_k(s) \in c_e\} \cup c) && \text{merge properties}
\end{aligned}
$$

(11)

◄

A.6 Case Study with Trusted Intermediary

Figure 17 shows the benchmark results for the to-do list case study when we trust the intermediaries and do not use an encARDT. The overall trends are similar, both the time per interaction and the size stored on the intermediary have a linear correlation to the current number of to-do entries. This is because those costs are inherent to the ARDT of the to-do list. There are notable differences. First, the overall runtime when using encARDTs is better (each interaction is faster), because merging the encARDT on the intermediary (i.e., pruning subsumed deltas) is faster than merging the to-do list on the intermediary (i.e., merging the two add-wins-last-writer-wins maps). The encryption overhead is negligible compared to that cost. Second, the overall size of the stored data on the trusted intermediary is smaller, because storing individual encrypted deltas requires more space as discussed in Subsection 6.2. Third, the client does not have to transmit any additional causality information and also does not create subsuming deltas that would reduce the overall size of an encARDT, but lead to larger deltas in some cases. This leads to a nearly constant bandwidth use, with small variations for the random difference between the relative amount of added, completed, and removed to-dos, as well as differences in to-do description lengths.
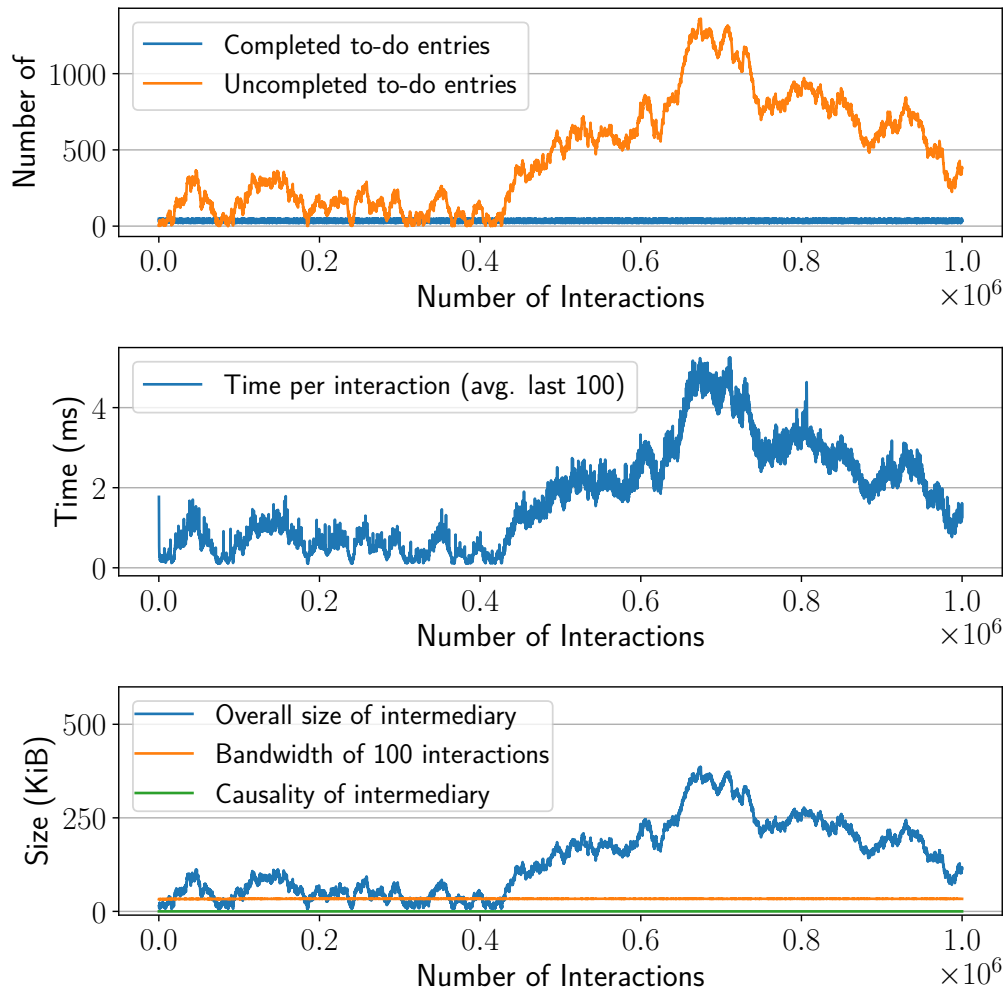
Figure 17 To-do list case study measurement results with trusted intermediary.