Journal of Parallel and Distributed Computing 144 (2020) 109–123

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

CScript: A distributed programming language for building mixed-consistency applications

Kevin De Porre [a],[*], Florian Myter [a], Christophe Scholliers [b], Elisa Gonzalez Boix [a]

[a] Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium
[b] Ghent University, Sint Pietersnieuwstraat 33, Ghent, Belgium

ARTICLE INFO

Article history:
Received 16 October 2019
Received in revised form 26 February 2020
Accepted 20 May 2020
Available online 30 May 2020

Keywords:
Distribution
Consistency models
Replicated data types

ABSTRACT

Current programming models only provide abstractions for sharing data under a homogeneous consistency model. It is, however, not uncommon for a distributed application to provide strong consistency for one part of the shared data and eventual consistency for another part. Because mixing consistency models is not supported by current programming models, writing such applications is extremely difficult. In this paper we propose CScript, a distributed object-oriented programming language with built-in support for data replication. At its core are consistent and available replicated objects. CScript regulates the interactions between these objects to avoid subtle inconsistencies that arise when mixing consistency models. Our evaluation compares a collaborative text editor built atop CScript with a state-of-the-art implementation. The results show that our approach is flexible and more memory efficient.

1. Introduction

According to the CAP theorem [6] a distributed system cannot remain both available and consistent under network partitions. This forces programmers to choose between availability (AP) and consistency (CP) in the event of a partition. This choice can be made for *each* piece of shared data [7]. We call applications that share multiple pieces of data with different guarantees *mixed-consistency applications*. When developing such applications, programmers face two major problems. First, distributed programming languages lack abstractions to share data under AP/CP guarantees. This forces programmers to manually synchronise replicas. As a result, programmers often make mistakes against consistency models [20]. Second, many AP approaches such as [8,25,26] develop common data types with hardcoded conflict resolution semantics. Unfortunately, programmers cannot compose these data types to design custom ones. Going beyond the current portfolio of available replicated data types (RDTs) requires programmers to manually engineer the RDT using ad hoc conflict resolution strategies. This has shown to be error-prone and results in brittle systems [1,15,25].

To help programmers develop mixed-consistency applications, we argue that distributed programming languages should have (1) built-in RDTs for writing AP and CP functionality and (2) built-in defence mechanisms that prevent programmers from making mistakes when mixing data with different consistency guarantees. In this paper, we propose CScript, a novel distributed programming language with native support for availability and consistency. CScript extends JavaScript with first-class *replicas* and *services*. Replicas are objects that encode their availability and consistency guarantees, and can be composed into services which are distributed over the network. CScript supports two families of AP replicated data types guaranteeing strong eventual consistency [25] (SEC): conflict-free replicated data types [25] (CRDTs) and strong eventually consistent replicated objects [10] (SECROs). CRDTs are a subset of the RDTs for which all operations commute. SECROs use semantic information provided by the programmer to reorder *conflicting* operations such that they do not need to commute. This approach is based on the idea that conflict detection and resolution naturally depend on the semantics of the application [29]. When the operations of an RDT do not commute and conflicts can be solved by reordering operations, CScript programmers can use SECROs to build the RDT. All replicas of this RDT are guaranteed to converge to the same state.

This paper complements our previous exposition of SECROs in [10] by proving convergence and showing that progress depends on the data type itself. Hence, we formulate a necessary condition for SECRO data types which enables us to give a general proof of progress.

The remainder of this paper is organised as follows. Section 2 discusses related work that is necessary to understand this paper. Section 3 introduces our motivating example

[*] Corresponding author.
E-mail address: kdeporre@vub.be (K. De Porre).

https://doi.org/10.1016/j.jpdc.2020.05.010
0743-7315/© 2020 Elsevier Inc. All rights reserved.

for mixed-consistency. Section 4 describes CScript's architecture and programming model. Section 5 describes our novel SECRO data type which is part of CScript. We then work out our motivating example in Section 6 using CScript. Section 8 evaluates CScript by benchmarking a collaborative text editing application. Finally, we discuss related work in Section 10 and close with final conclusions in Section 11.

2. Background

In this section we introduce background knowledge on the CAP theorem, its implications, and the consistency models on which the paper builds.

The CAP theorem [6,12] describes the interactions between consistency (C), availability (A) and partition tolerance (P) in a distributed system consisting of nodes that can write to a conceptually shared memory. A system is *consistent* if all reads return the latest write. A system is *available* when all nodes are able to read from and write to the shared memory at any point in time. The system is *partition tolerant* if it is able to maintain its consistency or availability guarantees in the face of network partitions. The CAP theorem proves that a distributed system cannot remain both available and consistent under network partitions. This led to a multitude of consistency models (mainly weak consistency models[1]) being developed [31]. Eventual consistency [32] for instance, states that when updates stop, all replicas will eventually converge to the same state.

Strong eventual consistency (SEC) [25] is a variation on eventual consistency [32] that imposes an additional *strong convergence* requirement: correct replicas that received the same updates (possibly in a different order) must be in the same state. Strong convergence thus defines *when* replicas converge, something that is not specified by traditional eventual consistency.

Today's only implementation of the SEC model is the conflict-free replicated data type (CRDT) [25]. CRDTs come in two flavours which have been proven equivalent: state-based CRDTs (abbreviated CvRDTs) and operation-based CRDTs (abbreviated CmRDTs). CvRDTs require replicated state to form a join-semilattice. As such, two states can always be merged deterministically by computing their least upper bound. On the other hand, CmRDTs require all operations to commute and as such guarantee strong convergence by design.

Imposing all operations to commute (or the equivalent requirement for state to form a join-semilattice) hurts the applicability of CRDTs. For this reason the literature describes only a limited portfolio of CRDTs. Furthermore, CRDTs cannot be composed out of the box. Some research [15,19] seeks to improve the composability of CRDTs, however, none of these composition mechanisms is general enough to allow arbitrary compositions for all CRDTs. JSON CRDTs [15] for instance only let programmers compose linked lists and maps. Hence, programmers often need to engineer their own CRDTs from scratch (if possible) or rely on manual conflict resolution which is error-prone and results in brittle systems [1,15,25].

3. Motivation: A mixed-consistency application

This section introduces a grocery list application which acts as a motivating example of mixed-consistency throughout this paper. Users of the application can create shared grocery lists to which they can add and delete items. Users can also request more pieces of an item or mark a certain quantity of an item as bought.

The application must meet the following consistency requirements:

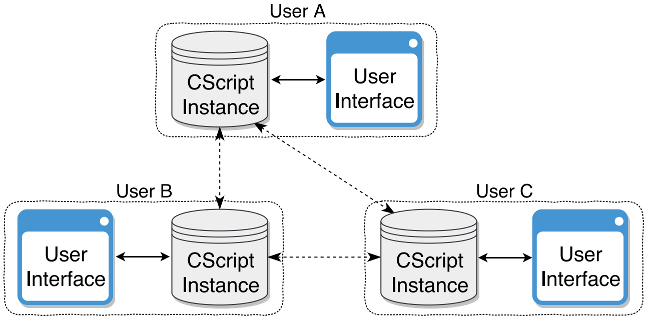


Fig. 1. Architecture of a typical CScript application with three users. Each user runs an instance of the application which consists of a CScript instance and a user interface.

Automatic Sharing Grocery lists are shared between all users. When a user creates a new grocery list, other users must automatically see that list.

Consistent Purchases Users should not be able to concurrently mark the same item as bought, i.e. purchases must happen consistently.

Offline Availability Users should be able to add, delete or update items of a grocery list, even while being offline. Updating a shared grocery list while being offline causes the list to diverge from the other replicas. The system must solve state inconsistencies when the user comes back online.

Note that the application requires multiple consistency levels. The grocery list itself is eventually consistent but marking items as bought is strongly consistent. Sometimes both levels interact, e.g. when purchasing an item we first try to mark it as bought and then update the grocery list.

Implementing this application is difficult because it not only requires programmers to implement the application logic but also to deal with aspects of distribution such as implementing service discovery, serialising objects, and implementing different consistency models to keep copies consistent. We argue that a language with appropriate replication mechanisms and built-in consistency models, can avoid this accidental complexity by hiding it in the language. As a result, programmers can focus on the application logic.

4. CScript

We now introduce CScript, our JavaScript extension for mixed-consistency applications. First, we provide a high-level description of CScript and describe the typical architecture of CScript applications. Afterwards, we introduce the building blocks of CScript's programming model.

4.1. Architecture

CScript is designed as a JavaScript library with dedicated syntax by means of macros [27]. When using the dedicated syntax an additional transpilation step is required to transform CScript into JavaScript (ECMAScript 6). The resulting application runs on top of NodeJS [23].

Fig. 1 shows the typical architecture of CScript applications. Users of the application run a CScript instance and possibly a user interface displaying the application's state. CScript instances running on the same local network are interconnected and form a full-mesh peer-to-peer network (dotted lines). Apart from the CScript instances there is no additional infrastructure, i.e. no centralised servers. All network communication is managed by the CScript runtime.

[1] Consistency models weaker than sequential consistency.

4.2. Programming model

We now describe CScript's programming model which is centred around the concepts of *replicas* and *services*. We then illustrate how these concepts facilitate the development of collaborative mixed-consistency applications.

4.2.1. Replicas

CScript introduces first-class replicated objects, called *replicas*. Like regular objects, replicas contain state in the form of fields and behaviour in the form of methods. State can be primitive data or JavaScript objects. Programmers can invoke methods of a replica but *cannot* access state directly. The state of a replica is automatically kept consistent by its consistency model.

CScript supports two types of replicas: available and consistent replicas. The former guarantee SEC [25] and thus favour availability over consistency. The latter guarantee sequential consistency [28] and thus favour correctness over availability.

4.2.2. Services

When building mixed-consistency applications, replicas alone are not enough. Programmers need a way to compose replicas – possibly with different consistency guarantees – into a bigger unit that provides specific functionality. To this end, CScript provides first-class *services*.

Services encapsulate state (primitive data, objects, and replicas) and implement some methods which form the service's API. The methods use the state and coordinate between the replicas to provide specific functionality. Programmers must use the service's API as they cannot access a service's state directly.

4.2.3. Publications and subscriptions

CScript lets programmers implement replicas and bundle them into services. To share services between instances of an application running of different devices, CScript features a topic-based publish–subscribe mechanism [11]. This mechanism lets application instances share services with one another without knowing each other beforehand, making the underlying network transparent to the application. Fig. 2 shows how CScript instances can publish services on the network and discover published services. When one instance discovers a service published by another instance, it acquires a copy of the service. The replicas encapsulated by the service are automatically managed by the CScript runtime such that they uphold their consistency guarantees.

4.2.4. The interplay between consistent and available replicas

Mixed-consistency applications share several pieces of data with different consistency guarantees. When building such applications, programmers must be careful not to break these guarantees. Ideally, the programming model enforces consistency models using a strict set of rules:

1. Each replica implements one specific consistency model.
2. Programmers can only interact with replicas through their public interface, i.e. programmers cannot access or modify a replica's internal state directly.
3. Replicas are self-contained since they are replicated over the network.
4. Replicas may not leak references to their internal state as this would allow programmers to access internal state directly, thereby breaking rule 2.
5. Data that is replicated under a certain consistency model should not flow to replicas that enforce a stronger consistency model as it would break the stronger guarantees.

We now discuss how CScript enforces the aforementioned rules. Even though CScript provides consistent and available replicas, programmers can only nest replicas that guarantee the same level of consistency. Otherwise, one replica could provide different (possibly conflicting) consistency levels, thereby breaking rule 1. Hence, consistent replicas may embed other consistent replicas but not available replicas, and vice-versa. Replicas are black boxes and do not allow programmers to access or modify internal state directly (rule 2).

CScript deep copies the arguments that are passed to the methods of replicas as well as the return values. Deep copying the arguments ensures that the replica remains self-contained (rule 3). Deep copying the return value avoids leaking references to internal state (rule 4).

Regarding data flows (rule 5), CScript does not yet prohibit data obtained from available replicas to be passed as argument to a method of a consistent replica. We foresee a statically typed version of the CScript language that encodes the consistency model of data as part of its type and rejects illegal information flows at compile time.

5. Strong Eventually Consistent Replicated Objects (SECROs)

We now focus on CScript's support for available replicas: *strong eventually consistent replicated objects* (SECROs), a novel RDT that addresses the applicability issues of CRDTs discussed in Section 2. SECROs use semantic information provided by the programmer to guarantee SEC without requiring operations to commute. This makes SECROs generally applicable.

5.1. SECRO data type

Like regular objects, SECROs contain state in the form of fields, and behaviour in the form of methods. The methods define the SECRO's public interface which cannot be circumvented. Methods can be further categorised in *accessors* (i.e. methods querying internal state) and *mutators* (i.e. methods updating the internal state).

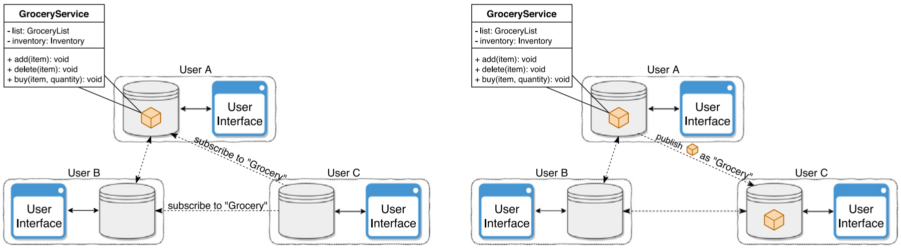
SECROs differ from regular objects in that programmers can enforce application-specific invariants by associating concurrent preconditions and postconditions to the mutators. We say that pre and postconditions are *state validators*. State validators are used by the SECRO to order concurrent operations in a way that does not violate any invariant.

5.2. State validators

State validators associate rules to mutators. Those rules express invariants over the state of the object which need to uphold in the presence of concurrent operations.[2] Behind the scenes, SECRO's replication protocol may interleave concurrent operations. From the programmer's perspective the only guarantee is that these invariants are upheld. State validators come in two forms:

Preconditions specify invariants that must hold prior to the execution of their associated operation. As such, preconditions approve or reject the state before applying the actual update. In case of a rejection, the operation is aborted and a different ordering of the operations will be tried.

[2] From now on, we use the terms operation and mutator interchangeably, as well as the terms update and mutation.



(a) User C subscribes to the "Grocery" topic.

(b) User A publishes a service under the "Grocery" topic. User C automatically discovers the service and acquires a copy.

Fig. 2. Exchanging a `GroceryService` containing two replicas (`list` and `inventory`) between CScript instances.

Postconditions specify invariants that must hold after the execution of their associated operation. A postcondition does not execute immediately after applying an operation. Instead, it executes after *all* concurrent operations complete. As such, postconditions approve or reject the state that results from a group of concurrent, potentially conflicting operations. In case of a rejection a different ordering of the operations is tried.

5.3. SECRO's replication protocol in a Nutshell

Recall that SECROs guarantee SEC (eventual consistency and strong convergence). To provide this guarantee SECROs implement a dedicated optimistic replication protocol. We now briefly discuss this protocol, a detailed explanation including pseudo code is given in Section 7.

SECRO's replication protocol asynchronously propagates update operations to all replicas. In contrast to CRDTs, the operations of a SECRO do not necessarily commute. Therefore, the replication protocol totally orders the operations at all replicas. This order respects causality and all pre and postconditions.

Replicas maintain their *initial state* and a sequence of operations called the *operation history*. Each time a replica receives an operation, it is added to the replica's history, which may require reordering parts of the history. Reordering the history boils down to finding an ordering of the operations that fulfils two requirements. First, the order must respect the causality of operations. Second, applying all the operations in the given order may not violate any of the concurrent pre or postconditions. An ordering which adheres to these requirements is called a *valid execution*. As soon as a valid execution is found each replica resets its state to the initial one and executes the operations in-order. Reordering the history is a deterministic process, hence, replicas that received the same operations find the same valid execution.

Note that the existence of a valid execution cannot be guaranteed for arbitrary pre and postconditions. It is the programmer's responsibility to define correct ones. However, the replication protocol guarantees that:

1. Eventually, all replicas converge towards the same valid execution (i.e. eventual consistency).
2. Replicas that received the same updates have identical operation histories (i.e. strong convergence).
3. Replicas eventually perform the operations of a valid execution if one exists, or issue an error if none exists.

As users perform operations, the operation histories of replicas may grow unboundedly. To alleviate this issue we allow a replica's state to be committed periodically. Concretely, replicas maintain a *version number*. Whenever a replica is committed, it clears its operation history and increments its version number. The replication protocol then notifies all other replicas of this commit, which adopt the committed state and also empty their operation history. As we explain in Section 7.1, the commit operation does not require synchronising the replicas and thus does not affect the system's availability.

6. CScript from the programmer's perspective

We now illustrate CScript's programming model by implementing a grocery application that fulfils the requirements outlined in Section 3.

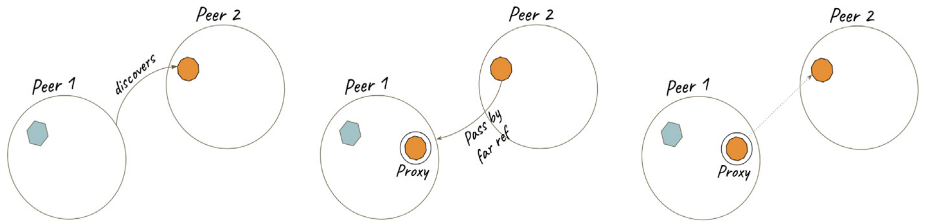
6.1. The grocery service

We model the grocery application as a CScript service, which is shown in Listing 1. On Line 1 we define the `GroceryService` using the `service` keyword. Similarly to class definitions in ES6,[3] services have a constructor method to initialise the service (Lines 4 to 7). The `GroceryService`'s constructor defines two fields: the grocery list's name and author (Lines 5 and 6). Additionally, the service encapsulates two replicas, `groceryList` and `inventory`, which are defined using the `rep` keyword (Lines 2 and 3). The former is the grocery list (an available replica) whereas the latter is the inventory containing all the items marked as bought (a consistent replica). Syntactically there is no difference between the eventually consistent `groceryList` replica and the sequentially consistent `inventory` replica because the consistency guarantees depend on the type of the replica. Finally, the service defines functionality to add, delete, and buy grocery items. This functionality is exposed through the `GroceryService`'s API which consists of the `add`, `delete`, and `buy` methods (Lines 8 to 14). The implementation of the `buy` method is discussed in Section 6.2.

```
1  service GroceryService {
2    rep groceryList = new GroceryList();
3    rep inventory   = new Inventory();
4    constructor(name, author)
5      this.name   = name;
6      this.author = author;
7    }
8    add(item) {
9      return this.groceryList.add(item);
10   }
11   delete(itemName) {
12     return this.groceryList.delete(itemName);
13   }
14   buy(itemName, buyingQuantity) { /* ... */ }
15 }
```

Listing 1: Implementation of the grocery service.

[3] ECMAScript 6.



(a) Peer 1 discovers a consistent replica at peer 2.

(b) Peer 2 passes the consistent replica to peer 1 by far reference [30].

(c) Peer 1 holds a far reference to a remote object living at peer 2.

Fig. 3. How to exchange consistent replicas and interact with them.

```
 1  class Inventory {
 2    constructor(stock = []) {
 3      this.stock = new Map(stock);
 4    }
 5    approve(itemName, stockQuantity, buyingQuantity) {
 6      if (buyingQuantity <= 0)
 7        return false;
 8      const trueStock = this.stock.getOrElse(itemName, 0);
 9      if (trueStock === stockQuantity) {
10        this.stock.set(itemName, trueStock + buyingQuantity);
11        return true;
12      }
13      else {
14        return false;
15      } } }
```

Listing 2: Implementation of the grocery service’s inventory.

```
 1  buy(itemName, buyingQuantity) {
 2    return new Promise((resolve, reject) => {
 3      const stockQuantity = this.groceryList.get(itemName).bought;
 4      this.inventory
 5          .then(inventory => {
 6            return inventory.approve(itemName, stockQuantity, buyingQuantity)
 7          }).then(accepted => {
 8            if (accepted) {
 9              this.groceryList.bought(itemName, buyingQuantity);
10              resolve();
11           }
12            else { reject("Buy request rejected."); }
13          });
14    });
15  }
```

Listing 3: Buying a certain quantity of a grocery item.

6.2. The sequentially consistent inventory of purchases

Listing 2 shows the implementation of the Inventory class, which keeps a map to track how many pieces of each item were marked as bought (Line 3), this is called the “stock”. The inventory defines an approve method which is called before marking a certain quantity of an item as bought (Lines 5 to 15). This method first checks that the user’s view on the stock is equal to the actual stock for that item (Lines 8 and 9), thereby rejecting concurrent purchases of the same item. If the check succeeds, the inventory approves the buy request and updates its stock for that item (Lines 10 and 11).

By default, CScript replicas are sequentially consistent unless the data type implements SEC. CScript guarantees sequential consistency by serialising all operations on a single (remote) copy of the replica which resides at the creator of the (grocery) service, as depicted in Fig. 3. This means that there is no central server hosting the inventory, instead, the inventory is hosted by the device that created it. Interactions with consistent replicas may therefore involve network communication. For this reason, property accesses and method invocations on consistent replicas are asynchronous and return a promise.

Listing 3 shows the implementation of the grocery service’s buy method. The method first fetches the user’s local view on the stock from the eventually consistent grocery list, which may thus be outdated (Line 3). Then, it asynchronously sends a request to the inventory by calling the approve method (Line 6). If the request is approved, it informs the local grocery list replica (Line 9) which then marks the given quantity of that item as bought in the UI. This method shows that services may have to interact with replicas that exhibit different consistency guarantees in order to provide the required functionality.

6.3. The eventually consistent grocery list

We now discuss the implementation of the GroceryList, which is an available replica providing functionality to fetch the items of a list, add items to a list, and mark (a certain quantity of) items as bought. To this end, we implemented the grocery list using our SECRO data type, presented in Section 5.

Listing 4 shows the implementation of the `GroceryList` which extends the `SECRO` interface. Its public interface consists of one accessor (`get`) and three mutators: `add`, `bought`, and `delete`. It also associates a precondition to the `bought` method and a postcondition to the `add` method, using the `pre` and `post` keywords respectively (Lines 15 and 16). The side-effect free method `get` is annotated with `@accessor`, otherwise, CScript treats it as a mutator.[4] The `tojson` and `fromjson` methods serve to (de)serialise the object as it will be replicated over the network. In order for the receiver to know the `GroceryList` class, this SECRO must be registered at the CScript *factory* (Line 21).

```
 1  class GroceryList extends SECRO {
 2    constructor(items = []) {
 3      super();
 4      this.items = new Map();
 5      items.forEach(this.add.bind(this));
 6    }
 7    @accessor
 8    get() { /* ... */ }
 9    // operations to manipulate the list
10    add(item) { /* ... */ }
11    bought(itemName, quantity) { /* ... */ }
12    delete(itemName) { /* ... */ }
13    // SECRO's state validators
14    post add(originalState, state, args, res)
15      { /* ... */ }
16    pre bought(state, args) { /* ... */ }
17    // serialisation methods
18    tojson() { /* ... */ }
19    static fromjson(items) { /* ... */ }
20  }
21  Factory.registerAvailableType(GroceryList);
```

Listing 4: Structure of the grocery list.

Let us now take a look at the implementation of the `add`, `bought`, and `delete` mutators and their associated pre and postconditions.

```
 1  add(item) {
 2    const description =
 3      this.items.getOrElse(
 4        item.name, {requested: 0, bought: 0});
 5    description.requested += item.requested;
 6    this.items.set(item.name, description);
 7  }
 8  post add(originalState, state, args, res) {
 9    const [item] = args,
10          addedQuantity = item.requested,
11          resultingQuantity =
12            state.items
13                 .getOrElse(item.name, 0)
14                 .requested;
15    return resultingQuantity >= addedQuantity;
16  }
```

Listing 5: Adding items to a grocery list.

Listing 5 shows the implementation of the add method (which adds a certain quantity of an item to the grocery list) and its associated postcondition. First, add fetches the item from the grocery list in case it already exists, or, creates a new item description otherwise (Line 4). Then, it increments the requested quantity (Line 5) and updates the item's description in the underlying map (Line 6). add's postcondition[5] states that the resulting state must reflect at least the quantity requested by the operation. While this invariant always holds in a sequential system, it may be violated when operations run concurrently, e.g. due to a concurrent delete of the same item. By stating this invariant explicitly, the SECRO will ensure add-wins semantics.

```
 1  bought(itemName, quantity) {
 2    const quantities = this.items.get(itemName);
 3    quantities.bought += quantity;
 4  }
 5  delete(itemName) {
 6    this.items.delete(itemName);
 7  }
 8  pre bought(state, args) {
 9    const [itemName, quantity] = args;
10    return this.items.has(itemName);
11  }
```

Listing 6: Marking items as bought and deleting items from the grocery list.

Listing 6 shows the `bought` and `delete` methods. `bought` fetches the item's description (Line 2) and increments it with the bought quantity (Line 3). `delete` removes the item from the underlying list (Line 6). On Lines 8 to 11 we associate a precondition[6] to the `bought` method which checks that the item exists. We associate no postcondition to `delete` (only to `add`) because we expect `add`s to win over `delete`s, as shown in Fig. 4.

Having discussed the implementation of the `add`, `bought`, and `delete` operations, we now describe which operations can be generated by the users in a given state s. We call this the set of *valid updates* and denote it V_s.

$$\frac{item = \langle name, req, bought\rangle \quad name \in String \quad req \in \mathbb{N}^+ \quad bought \in \mathbb{N}_0}{add(item) \in V_s} \quad (1)$$

$$\frac{\langle name, \_, \_\rangle \in s \quad qty \in \mathbb{N}^+}{bought(name, qty) \in V_s} \quad (2) \qquad \frac{\langle name, \_, \_\rangle \in s}{delete(name) \in V_s} \quad (3)$$

The first rule states that users can always add well-formed items to the grocery list, independent of the application's state. The second rule states that users can only buy a positive quantity of an existing item. The third rule states that users can only delete existing items.

6.4. Sharing grocery services between users

Users of our grocery application can create new grocery lists at will. Each grocery list must be shared between all instances (users) of the grocery application. To this end, we use CScript's publish–subscribe mechanism.

Every time the user creates a new grocery list the `createGrocery` function from Listing 7 is invoked. This function first creates a grocery service representing the list (Line 4), then publishes the newly created service under the `Grocery` type tag using the `publish <service> as <typetag>` construct (Line 5). The typetag is the topic of publication and is defined using the `deftype` keyword on Line 1. Afterwards, the function calls `processService` which installs the necessary callbacks on the service, such that the application can react to incoming updates, e.g. when another user adds an item to the shared grocery list. Reacting to updates will be discussed further in this section.

Listing 8 shows how to subscribe to services of the `Grocery` type using the `subscribe <typetag> with <callback>` construct (Line 2). The provided callback is parametrised with the

[4] When a mutator is invoked, the operation is propagated to all replicas.

[5] Postconditions receive four arguments: the state before applying the operation, the state after applying all concurrent operations, the operation's arguments and return value.

[6] Preconditions receive the state before applying the operation and the arguments.

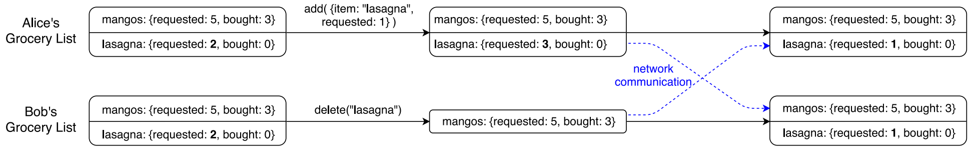

Alice's Grocery List
mangos: {requested: 5, bought: 3}
lasagna: {requested: 2, bought: 0}
add({item: "lasagna", requested: 1})
mangos: {requested: 5, bought: 3}
lasagna: {requested: 3, bought: 0}
mangos: {requested: 5, bought: 3}
lasagna: {requested: 1, bought: 0}
network communication
Bob's Grocery List
mangos: {requested: 5, bought: 3}
lasagna: {requested: 2, bought: 0}
delete("lasagna")
mangos: {requested: 5, bought: 3}
mangos: {requested: 5, bought: 3}
lasagna: {requested: 1, bought: 0}

Fig. 4. Alice adds one lasagna while concurrently Bob deletes the lasagnas from the grocery list. After propagating the operations the resulting list contains one lasagna because Bob was not aware of Alice's addition at the time of his deletion.

```
deftype Grocery
function createGrocery(name, author) {
  const gservice =
    new GroceryService(name, author);
  publish gservice as Grocery;
  processService(gservice);
  return gservice;
}
```

Listing 7: Exporting grocery services on the network.

discovered service. Upon discovering a service, CScript invokes the callback, which in this case fetches the service's name and author (Lines 3 and 4) and creates a unique identifier for the service (Line 5). The callback then stores the discovered service in a map, on Line 6. Note that the `name` and `author` fields contain regular objects. When discovering the service the application acquires a deep-copy of those fields (which contrary to replicas are not kept consistent).

```
1  const services = new Map();
2  subscribe Grocery with gservice => {
3    const name   = gservice.name,
4          author = gservice.author,
5          id     = ‘${name} by ${author}‘;
6    services.set(id, gservice);
7    processService(gservice);
8  }
```

Listing 8: Subscribing to grocery services.

In order to make services self-contained, they do not have access to enclosing lexical scopes, much like isolates in AmbientTalk [30] or spores in Scala [21].

6.4.1. Reacting to updates of the grocery list

When a user modifies a shared grocery list all replicas will eventually observe the update and in turn update the user interface. To this end, CScript replicas emit two events to which applications can react: `RemoteUpdate` and `Update`. The former is triggered when a replica receives an update from a remote replica. The latter is triggered when a replica applies an update.

Fig. 5 shows how updates are propagated between two users. Alice adds an item to her grocery list (m_1) and the operation is sent to bob (m_2). Update events are triggered on both devices (m_3 and m_7) which causes the user interfaces to be refreshed (m_4, m_5 and m_8, m_9).

7. SECRO's replication protocol

Having introduced the CScript language and our SECRO data type, we now turn our attention to the replication algorithm behind SECROs. The detailed algorithm is explained in [10]. This paper provides the correctness proofs and presents only the parts of the algorithm that are relevant to the proofs.

7.1. Algorithm

The algorithm described in this section assumes a reliable causal order broadcasting mechanism without loss of generality, i.e. a communication medium in which messages arrive in an order that is consistent with the happened-before relation [16]. It also assumes that reading the state of a replica happens side-effect free and that mutators solely affect the replica's state (i.e. the side effects are confined to the replica itself).

A SECRO replica r is a tuple $\langle v_i, s_0, s_i, h, id_c \rangle$ consisting of the replica's version number v_i, its initial state s_0, its current state s_i, its operation history h, and the globally unique identifier of the latest commit operation id_c. Reading the value of the replica simply returns its latest local state s_i. A mutator m is represented as a tuple $\langle o, p, a \rangle$ consisting of the update operation o, precondition p, and postcondition a. When a mutator is applied to a replica a *mutate* message is broadcast to all replicas. Such a message is an extension of the mutator $\langle o, args, p, a, c, id \rangle$ which additionally contains the arguments $args$ passed to the operation o, the node's logical clock time c, and a globally unique identifier id. We denote that a mutation m_1 happened before m_2 using $m_1 \prec m_2$. Similarly, we denote that two mutations happened concurrently using $m_1 \parallel m_2$. Both relations are based on the clocks carried by the mutate messages [14].

Algorithm 1 governs the replicas' behaviour to guarantee SEC by ensuring that all replicas execute operations in the same order. In particular, algorithm 1 delivers a list of *mutate* messages l to a replica r which optionally returns the updated replica r', denoted $l \Downarrow r = Some\ r'$ or $l \Downarrow r = None$. The algorithm consists of two parts. First, it appends the list of *mutate* messages to the operation history, sorts the history according to the $\gg$ total order, and generates all *linear extensions* of the replica's sorted history (see Lines 1 and 3). We say that $m_1 = \langle o_1, args_1, p_1, a_1, c_1, id_1 \rangle \gg m_2 = \langle o_2, args_2, p_2, a_2, c_2, id_2 \rangle$ iff $id_1 > id_2$, however, this could be any *total* order. The generated linear extensions are all the permutations of h' that respect the partial order defined by the operations' causal relations. Since replicas deterministically compute linear extensions and start from the same sorted operation history, all replicas generate the same sequence of permutations.

Second, the algorithm searches for the first *valid* permutation. For each operation within such a permutation it computes the transitive closure of concurrent operations[7] and checks that their pre (Lines 11 to 17) and postconditions (Lines 18 to 24) hold.

[7] The transitive closure of a mutate message m with respect to an operation history h is denoted $TC(m, h)$ and is the set of all operations that are directly or transitively concurrent with m, including m itself. A formal definition is provided in Appendix C.

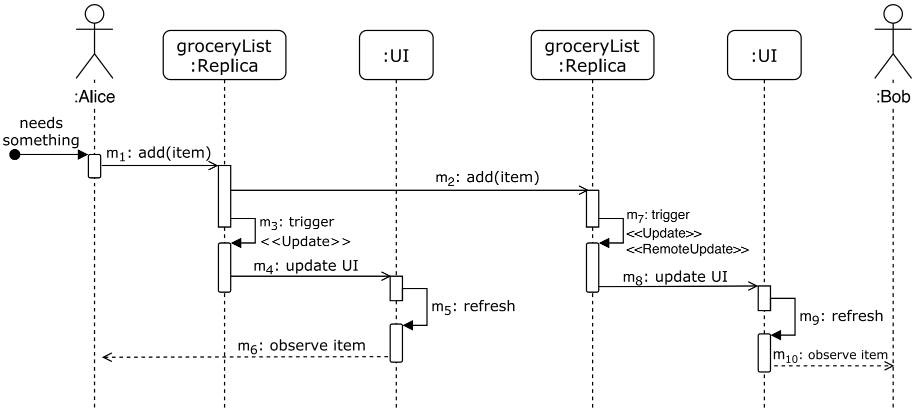


Fig. 5. Sequence diagram illustrating updates of the grocery application.

ALGORITHM 1: Handling *mutate* messages

```
arguments: A list of mutate messages $l$, a replica $r = \langle v_i, s_0, s_i, h, id_c \rangle$
$h' = h \mathbin{++} l$
$s'_i = s_i$
for $ops \in LE(sort_{>>}(h'))$ do
    $s_i = copy(s'_i)$ // Restore the replica's state
    pre = 0
    post = 0
    for $m \in ops$ do
        concurrentClosure = $TC(m, h') \cup \{m\}$
        ogStates = Map()
        retVals = Map()
        for $\langle o, args, p, a, c, id \rangle \in concurrentClosure$ do
            if $p(s_i, args)$ then
                pre $+= 1$
                ogStates.put(id, copy($s_i$)) // copy state to pass to
                    postcondition
                retVals.put(id, o(args)) // o's side-effects mutate $s_i$
            end
        end
        for $\langle o, args, p, a, c, id \rangle \in concurrentClosure$ do
            ogState = ogStates.get(id)
            retVal = retVals.get(id)
            if $a(ogState, s_i, args, retVal)$ then
                post $+= 1$
            end
        end
        ops = ops \ concurrentClosure
    end
    if $pre == |ops| \wedge post == |ops|$ then
        return Some $\langle v_i, s_0, s_i, ops, id_c \rangle$
    else
        return None
    end
end
```

Postconditions are checked only after all concurrent operations of the transitive closure executed since they happened independently and may thus conflict. The algorithm returns the replica's updated state as soon as a valid execution is found, $l \Downarrow r = Some\ \langle v_i, s_0, s'_i, h', id_c \rangle$. If no valid execution exists the algorithm fails, $l \Downarrow r = None$.

Besides reading and mutating replicas, it is possible to commit a replica. Commit clears the replica's operation history h, increments the replica's version and replaces the initial state s_0 by the current state s_i. This avoids unbounded growth of operation histories, but operations concurrent with the commit will be discarded.[8] Commit operations commute in order not to compromise availability. The detailed commit algorithm and its explanation can be found in [10].

7.2. Convergence and progress properties

As mentioned before, SECROs guarantee strong eventual consistency (SEC). This means that the replication algorithm ensures two properties: *strong convergence* and *progress* [25]. The former states that replicas which received the same operations must be in equivalent states. The latter states that if some replica generates a valid operation, applying that operation on another replica may not lead to an error state [13].

The SECRO algorithm guarantees strong convergence by deterministically reordering the operations at all replicas. Recall from the previous section that all replicas execute all operations in the same order and thus converge to the same state. Appendix A provides the complete proof of convergence.

The main advantage of SECROs over CRDTs lies in the fact that it is a *general-purpose* RDT. Programmers explicitly specify preconditions and postconditions that constrain the data type's behaviour under concurrent operations. Depending on these pre and postconditions a replica may or may not end up in an error state. Hence, we cannot provide a general proof of progress that holds for all SECROs. Instead, we require the SECRO's pre and postconditions to accept at least one causal serialisation[9] of the operations (see Lemma 1).

Lemma 1. *Given an initial state s and a set of valid updates V_s,*[10] *there exists an ordering of the updates that respects causality and all pre and postconditions.*

Appendix B provides a proof that SECROs whose pre and postconditions meet Lemma 1 guarantee progress. It is up to the programmer to prove Lemma 1 when designing custom SECROs.

8. Evaluation

To evaluate CScript we built several applications, including the grocery list application and a collaborative text editing application. The text editor is built on top of SECROs, one of CScript's core abstractions, which makes the application highly available and partition tolerant (AP). We compare the application to a state-of-the-art implementation on top of JSON CRDTs [15]. To this end, we perform various experiments which quantify the memory usage and execution time of both implementations.

[8] Since commit may drop operations, one can argue that SECROs are similar to last-writer-wins (LWW) strategies. However, SECROs guarantee invariant preservation, which is not the case with CRDTs.

[9] An ordering of the operations that respects the causality of the operations.

[10] The set of valid updates V_s is defined as the set of all updates that can be generated by the application while being in state s.

JSON CRDTs are closely related to SECROs because they allow programmers to build custom CRDTs by nesting linked lists and maps, without having to worry about conflicts. However, the extensibility of JSON CRDTs is limited to the composition of lists and maps, and conflict resolution cannot be customised because it is hardcoded by the implementation of lists and maps.

Note that SECROs are designed to ease the development of custom RDTs guaranteeing SEC. Hence, our goal is not to outperform JSON CRDTs, but rather to evaluate the practical feasibility of SECROs. The results show that SECROs are memory efficient but induce a linear time overhead on top of the operations. Overall, SECROs can be made practical by committing regularly.

8.1. A text editing application

The collaborative text editor lets users share text documents and work on them simultaneously. A naive version of this application stores text documents as a linked list of characters. An improvement would be to store documents as a balanced tree of characters, allowing for logarithmic time lookups, insertions, and deletions. We implemented both versions of the text editor using SECROs in CScript. The tree version uses a third party AVL tree and extends it with pre and postconditions to turn it into a SECRO that can freely be replicated. The implementation is publicly available at [9] and is detailed in [10].

Since JSON CRDTs only let programmers nest linked lists and maps, it is not possible to implement a balanced tree data structure. Hence, using JSON CRDTs we were only able to implement the naive version of the text editor.

We compare to JSON CRDTs because they are designed to build custom CRDTs and are thus similar to SECROs which are meant to build custom RDTs. We did not compare CScript to other languages because performance benchmarks would be biased by the language.

8.2. Methodology

All experiments presented in this section were performed on a cluster consisting of 10 worker nodes which are interconnected through a 10 Gbit twinax connection. Each worker node has an Intel Xeon E3-1240 processor at 3.50 GHz and 32 GB of RAM. Depending on the experiment, the benchmark is either run on a single worker node or on all ten nodes. We specify this for each benchmark.

To get statistically sound results we repeat each benchmark at least 30 times, yielding a minimum of 30 samples per measurement. Each benchmark starts with a number of warmup rounds to minimise the effects of program initialisation. We also disable NodeJS' just-in-time compiler optimisations.

We perform statistical analysis over our measurements as follows. We discard samples that are affected by garbage collection (e.g. the execution time benchmarks). For each measurement comprising at least 30 samples we compute the average value and the corresponding 95% confidence interval.

8.3. Memory usage

To compare the memory usage of the SECRO and JSON CRDT text editors, we perform an experiment in which 1000 operations are executed on each text editor. We continuously alternate between 100 character insertions followed by deletions of those 100 characters. We force garbage collection after each operation,[11] and measure the heap usage. Fig. 6 shows the results. Green and red columns indicate character insertions and deletions respectively.

Fig. 6a confirms our expectation that the SECRO implementations are more memory efficient than the JSON CRDT one. The memory usage of the JSON CRDT text editor grows unbounded since CRDTs cannot delete characters but merely mark them as deleted using *tombstones*.[12] Conversely, SECROs support true deletions by reorganising concurrent operations in a non-conflicting order. This results in lower memory usage, since all 100 inserted characters are deleted by the following 100 deletions.

Fig. 6b compares the memory usage of the list and tree-based implementations using SECROs. We conclude that the tree-based implementation consumes more memory than the list implementation because nodes of a tree maintain pointers to their children, whereas nodes of a singly linked list only maintain a single pointer to the next node. Interestingly, we observe a staircase pattern. This pattern indicates that memory usage grows when characters are inserted (green columns) and shrinks when characters are deleted (red columns). Overall, memory usage increases linearly with the number of executed operations, even though we delete the inserted characters and commit the replica after each operation. Hence, SECROs cause a small memory overhead for each executed operation, as shown by the dashed regression lines.

8.4. Execution time

In this section we discuss several aspects of the execution time of SECROs. First, we analyse the effect of committing the SECRO's operation history on the execution time of operations. Then, we compare the SECRO list implementation of the text editor to a state-of-the-art implementation with JSON CRDTs.

8.4.1. The effect of commit on the execution time

We now present two benchmarks related to the commit operation. The first quantifies the performance overhead of SECROs that results from reordering the operation history. The second illustrates the effect of commit on the execution time of the collaborative text editor and how commit improves its performance.

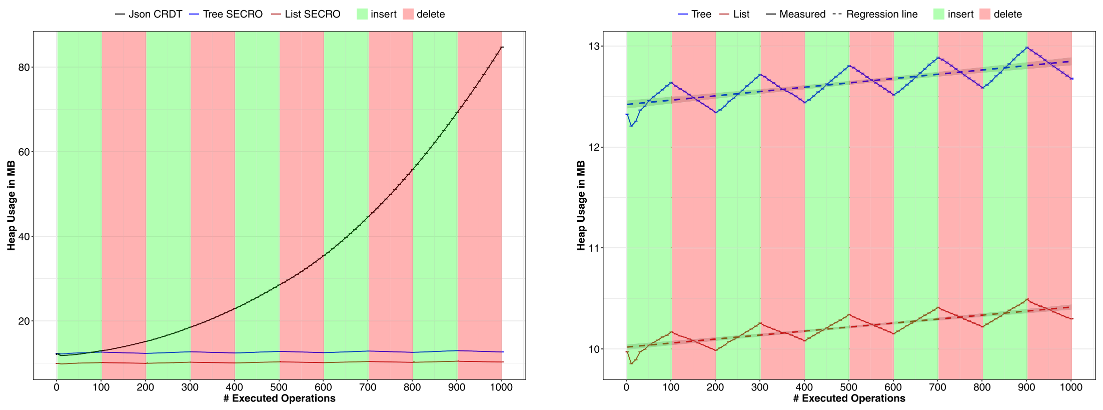
To quantify the performance overhead of SECROs we measure the execution times of 500 *constant* time operations, for different commit intervals. Each operation computes 10 000 tangents and has no associated pre or postcondition. Hence, the results reflect the best-case performance of SECROs.

Fig. 7a depicts the execution time of the aforementioned constant time operation. If we do not commit the replica (red curve), the operation's execution time increases linearly with the number of operations. Hence, SECROs induce a linear overhead. This results from the fact that the replica's operation history grows with every operation. Each operation requires the replica to reorganise the history. To this end, the replica generates linear extensions of the history until a valid ordering of the operations is found (see Algorithm 1 in Section 7.1). Since we defined no preconditions or postconditions, every order is valid and the replica generates exactly one linear extension and validates it. To validate the ordering, the replica executes each operation. Therefore, the operation's execution time is linear to the size of the operation history.

Note that commit implies a trade-off between concurrency and performance. Small commit intervals lead to better performance but less concurrency, whereas large commit intervals

[11] Forcing garbage collection is needed to get the real-time memory usage. Otherwise, the memory usage keeps growing until garbage collection is triggered.

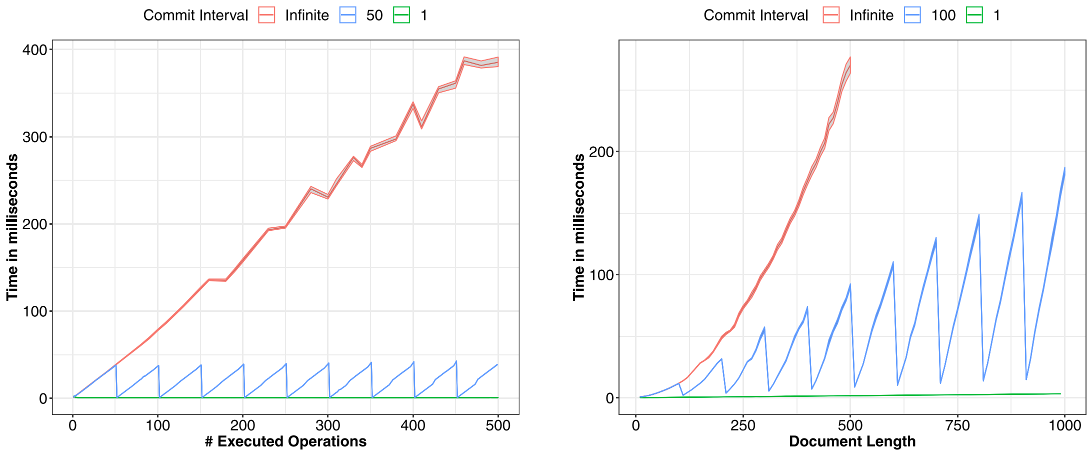
[12] Tombstones are a trick to make the insert and delete operations commute.



(a) Memory usage of the SECRO and JSON CRDT text editors.

(b) Memory usage of the list and tree implementations of the SECRO text editor.

Fig. 6. Memory usage benchmarks. Error bars represent the 95% confidence interval for the average taken from 30 samples. The experiments are performed on a single worker node. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)



(a) Execution time of a constant time operation in function of the number of executed operations.

(b) Time to append a character to the text document using the list implementation of the SECRO text editor.

Fig. 7. Execution time of SECROs for different commit intervals, performed on a single worker node of the cluster. Error bands represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

support more concurrent operations at the cost of performance. Fig. 7a illustrates this trade-off. For a commit interval of 50 (blue curve), we observe a sawtooth pattern. The operation's execution time increases until the replica is committed, whereafter it falls back to its initial execution time. This is because *commit* clears the operation history. When choosing a commit interval of 1 (green curve), the replica is committed after every operation. Hence, the history contains a single operation and does not need to be reorganised. This results in a constant execution time.

We now analyse the execution time of insert operations on the collaborative text editor. Fig. 7b shows the time it takes to append a character to a text document in function of the document's length, for various commit intervals. If we do not commit the replica (red curve), append exhibits a quadratic execution time. This is because the SECRO induces a linear overhead and append is a linear operation. Hence, append's execution time becomes quadratic. For a commit interval of 100 (blue curve) we again observe a sawtooth pattern. In contrast to Fig. 7a the peaks increase linearly with the size of the document because append is a linear operation. For a commit interval of 1 (green curve) we get a linear execution time. This results from the fact that we do not need to reorganise the replica's history. Hence, we execute a single append operation.

From these results, we draw two conclusions. First, SECROs induce a linear overhead on the execution time of operations. Second, commit is a practical solution to keep the performance of SECROs within acceptable bounds.

8.4.2. SECRO vs. JSON CRDT text editor

We now compare the naive list implementation and the advanced tree implementation of the text editor to the JSON CRDT implementation. To this end, we measure the time it takes to append characters to a text document. Although this is not a realistic edition pattern, it showcases the worst case performance.

From Fig. 8a we notice that the SECRO versions exhibit quadratic performance, whereas the JSON CRDT version exhibits linear performance. The reason for this is that reordering the SECRO's history induces a linear overhead on top of the operations themselves (as explained in Section 8.4.1). Since insert is also a linear operation, the overall performance of the text editor's insert operation is quadratic. To address this performance overhead the replica needs to be committed periodically.

Fig. 8a also shows that the SECRO implementation that uses a linked list is faster than its tree-based counterpart. To determine the cause of this counterintuitive observation, we measured the different parts that make up the total execution time in Appendix D. We found that the time overhead incurred by copying the document[13] kills the speedup we gain from organising the document as a tree. This is because each insertion inserts only a single character but requires the entire document to be copied.

To validate this hypothesis, we re-execute the benchmark shown in Fig. 8a but insert 100 characters per operation. Fig. 8b shows the resulting execution times. As expected, the tree implementation now outperforms the list implementation. This means that the speedup obtained from 100 logarithmic insertions exceeds the copying overhead induced by the tree. In practice, this means that single character manipulations are too fine-grained. Manipulating entire words, sentences or even paragraphs is more beneficial for performance.

Overall, the execution time benchmarks show that deep copying the document induces a considerable overhead. We believe that this overhead is not inherent to SECROs, but to its implementation on top of mutable objects.

9. Guidelines for designing Replicated Data Types (RDTs)

We now provide some guidelines for designing replicated data types under (strong) eventual consistency. When designing available systems, programmers need to use existing RDTs or design their own. If a data type's operations naturally commute then replicating it will guarantee strong eventual consistency out of the box, given that updates are eventually propagated to all replicas. This is for instance the case of a counter data type, whose increment and decrement operations commute.

When the data type's operations do not naturally commute, one can browse the literature for an equivalent CRDT. A CRDT may exist that applies some clever tricks to make the operations commute (e.g. OR-Sets [24]).

If none of the above applies one can resort to SECROs to build their replicated data type without worrying about commutativity. SECROs are able to omit the commutativity requirement by (re)ordering operations deterministically. This naturally entails some performance cost, as shown in Section 8. Note that some conflicts may not be solvable solely by reordering operations and can thus not be tackled using SECROs. This is the case for mutually exclusive operations. When two mutually exclusive operations execute concurrently, a conflict will arise that can only be solved by discarding at least one of the operations. In those cases, the programmer may resort to synchronising the mutually exclusive operations, similarly to [3,4], or implement an ad-hoc conflict resolution scheme.

Finally, we draw the relation between CmRDTs (operation-based CRDTs, see Section 2) and SECROs. Both data types ensure SEC, but CmRDTs require all concurrent operations to commute. As such, all valid serialisations of the operations – those respecting causality – yield the same valid state. Interestingly, SECROs guarantee that all replicas agree on one valid serialisation (without having to synchronise with one another). Pre and postconditions are used to confine the set of serialisations from which to pick one, e.g. to ensure that the given serialisation guarantees a certain conflict resolution strategy. Since all serialisations of a CmRDT are equivalent, any CmRDT can be implemented as a SECRO that associates no pre or postconditions to the operations. We thus conclude that CmRDTs are a subset of SECROs.

10. Related work

We now describe work that is closely related to the ideas presented in this paper. We distinguish between two research areas. First, we discuss programming languages and abstractions that like CScript help programmers trade off consistency for availability and vice-versa. Second, we discuss research on (strong) eventual consistency that is related to the SECRO data type.

Programming languages. CAPtain [22] is a programming model with two types of replicated objects: consistents and availables. The former guarantee strong consistency whereas the latter guarantee availability but only eventual consistency. These two types of objects are completely separated and form CAPtain's unit of distribution. In contrast to CAPtain, CScript bundles replicas into services which can be partly consistent and partly available, and distributes those services over the network. Each service exposes specific functionality through its API by coordinating between consistent and available replicas.

Geo [5] is an actor system for geo-replication that combines caching with replication techniques to hide latency and benefit from data locality where possible. Geo supports "single-instance" and "multi-instance" caching policies for actors across clusters. The single-instance caching policy is similar to consistent replicas in CScript, as it ensures a single instance of the actor that serialises all updates. The multi-instance caching policy replicates the actor to every cluster. These actors can be kept strongly consistent using Geo's distributed cache coherence protocol, or eventually consistent using Geo's Versioned API.

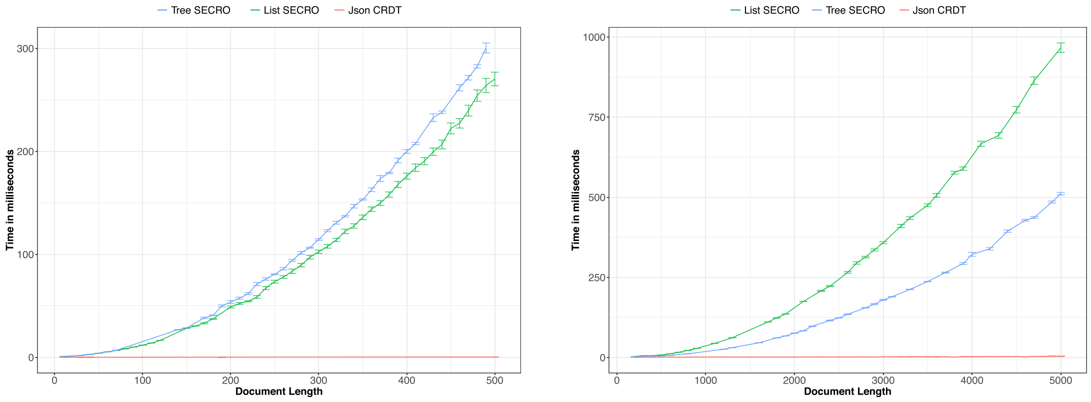
The MixT programming language [20] proposes mixed-consistency transactions to manipulate data with different consistency levels within a single database transaction. Using information flow analysis, MixT can break down mixed-consistency transactions into subtransactions for each consistency level and still guarantee atomicity. MixT works on top of existing databases whereas CScript's programming model integrates replication at the object-level.

Lasp [19] is the first programming language where CRDTs are first-class citizens. New CRDTs are defined through functional transformations over existing ones. In contrast, CScript provides SECROs, general-purpose RDTs which are not limited to a portfolio of builtin data types. Existing data structures can be turned into SECROs by associating state validators to the operations.

Eventual consistency. Central to SECROs is the idea of using application-specific information to reorder conflicting operations. Bayou [29] was the first system to use application-level semantics for conflict resolution by means of user-defined merge procedures. However, our work does not require manual conflict resolution; programmers instead specify the invariants the application must uphold in the face of concurrent updates, and the underlying update algorithm deterministically orders operations as to respect these invariants.

IPA [2] is closely related to SECROs as it preserves application invariants without coordinating operations. IPA extends the operations of traditional CRDTs with effects that guarantee the preservation of invariants in the face of concurrent updates. IPA differs from SECROs in that they modify operations whereas SECROs reorder concurrent operations.

[13] Since JavaScript objects are mutable, our prototype implementation of SECROs needs to copy the state before tentatively executing its operation history.



(a) Execution time of an operation that appends one character to a document.

(b) Execution time of an operation that appends 100 characters to a document.

Fig. 8. Execution time of character insertions in the collaborative text editors. Replicas are never committed. Error bars represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection are discarded.

JSON CRDTs [15] ease the construction of CRDTs by hiding the commutativity restriction that traditionally applies to the operations. Programmers can build new CRDTs by nesting lists and maps in arbitrary ways. The major shortcoming is that nesting lists and maps does not suffice to implement arbitrary RDTs. Moreover, programmers cannot customise conflict resolution because it is hardcoded by the implementation of lists and maps. Hence, JSON CRDTs are not truly general-purpose as opposed to SECROs.

Cloud types [8] are RDTs that like SECROs do not impose restrictions on the operations of the data type. However, cloud types hardcode how to merge updates coming from different replicas of the same type. As such, programmers have no means to customise the merge procedure of cloud types to fit the application's semantics. Instead, they are bound to implement a new cloud type and the accompanying merge procedure that fits the application. Hence, conflict resolution needs to be manually dealt with.

Some work has considered a hybrid approach offering SEC for commutative operations, and requiring strong consistency for non-commutative ones [3,4]. There are some similarities to SECROs as they employ application-specific invariants to classify operations as safe or unsafe under concurrent execution. These hybrid approaches synchronise unsafe operations, whereas SECROs reorder them as to avoid conflicts without giving up on availability. Partial Order-Restrictions (PoR) consistency [18] uses application-specific restrictions over operations but cannot guarantee convergence nor invariant preservation since these properties depend on the restrictions over the operations specified by the programmer.

11. Conclusion

In this work we propose CScript, a distributed programming language featuring consistent and available *replicas*. Consistent replicas guarantee strong consistency but are not available under network partitions. On the other hand, programmers can always execute operations on available replicas but they only guarantee strong eventual consistency (SEC) [25]. CScript lets programmers bundle replicas into larger components called *services*. Services can mix available and consistent replicas which eases the development of mixed-consistency applications. The CScript runtime manages all replicas automatically, thereby freeing the programmer from manually synchronising them.

CScript supports two types of available replicas: conflict-free replicated data types (CRDTs) [25] and strong eventually consistent replicated objects (SECROs). Several CRDTs are built-in and programmers can implement custom ones. When CRDTs are not applicable, programmers can use our general-purpose SECRO data type. A SECRO is an RDT that guarantees SEC without imposing restrictions on the data type's operations. Upon concurrent operations, SECROs compute a global total order of the operations that is conflict-free, without synchronising the replicas. To this end, SECROs use *state validators*: application-specific invariants that determine the object's behaviour in the face of concurrency. By specifying state validators arbitrary data types can thus be turned into available replicas.

To the best of our knowledge, SECROs are the first approach to support truly general-purpose RDTs while still guaranteeing SEC. In this paper, we prove that SECROs guarantee convergence and we formulate a necessary condition for SECRO data types which is sufficient to then prove progress.

To evaluate our work, we implemented a collaborative text editing application using SECROs in CScript and compared it to a state-of-the-art implementation that uses JSON CRDTs. The memory usage benchmarks reveal that SECROs are more memory efficient than JSON CRDTs. Time complexity benchmarks reveal that SECROs induce a linear time overhead which is proportional to the size of the operation history. Performance wise, SECROs can be competitive to state-of-the-art solutions if committed regularly.

CRediT authorship contribution statement

Kevin De Porre: Conceptualization, Formal analysis, Funding acquisition, Investigation, Methodology, Software, Supervision, Writing - original draft, Writing - review & editing. **Florian Myter:** Formal analysis, Funding acquisition, Methodology, Writing - original draft. **Christophe Scholliers:** Formal analysis, Methodology, Supervision, Writing - original draft, Writing - review & editing. **Elisa Gonzalez Boix:** Conceptualization, Funding acquisition, Methodology, Supervision, Writing - original draft, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

Kevin De Porre is funded by an SB Fellowship of the Research Foundation - Flanders, Belgium. Project number: 1S98519N.

Appendix A. General proof of convergence for SECROs

In this appendix we prove that the SECRO data type guarantees *strong convergence*. In other words, we prove that SECRO replicas which received the same updates are in equivalent states.

In what follows we consider the SECRO implementation without commits and can therefore simplify the representation of SECRO replicas to a tuple $r = \langle s, h\rangle$ consisting of the replica's initial state s and its history h.

Definition A.1. Assume a replica r_1 with initial state s_1 and history h_1, and a replica r_2 with initial state s_2 and history h_2. We say that r_1 is equivalent to r_2 iff they have the same initial state and the same operation history: $\forall r_1 \forall r_2 : r_1 \equiv r_2 \iff s_1 = s_2 \wedge h_1 = h_2$.

We now prove convergence for the SECRO implementation without commits. We use the notation $l \Downarrow r$ to deliver a list of updates l to a replica r which optionally yields the updated replica. We denote the set of permutations of a list l by $Perm(l)$.

Theorem A.1. *Replicas that received the same updates (possibly in a different order) are in equivalent states:*

$$\forall r_1, r_2 \forall l_1, l_2 : r_1 = \langle s_1, h_1\rangle \wedge r_2 = \langle s_2, h_2\rangle \wedge s_1 = s_2 \wedge \\ h_1 ++ l_1 \in Perm(h_2 ++ l_2) \implies l_1 \Downarrow r_1 \equiv l_2 \Downarrow r_2$$

Proof. When we deliver the updates l_1 to the replica r_1, Algorithm 1 appends the incoming updates to the history on Line 1: $h_1' = h_1 ++ l_1$. Similarly, when we deliver the updates l_2 to replica r_2, we add the updates to the history: $h_2' = h_2 ++ l_2$. Since h_1' and h_2' are permutations of one another, sorting them according to a total order $\gg$ yields the same list of updates: $l = sort_{\gg}(h_1') = sort_{\gg}(h_2')$. Both replicas then deterministically generate the linear extensions of l on Line 3: $LE(l)$ and search for the first extension that is valid (i.e. respects all pre and postconditions). Given that the pre and postconditions are deterministic, finding the first valid extension is also deterministic. Hence, either both replicas find the same ordering of operations h' and end up in equivalent states $\langle s_1, h'\rangle \equiv \langle s_2, h'\rangle$, or, both replicas end up in an error state because no valid extension exists. □

Appendix B. Proof of progress for SECROs

As argued in Section 7.2, we cannot provide a general proof of progress for SECROs because the pre and postconditions are defined by the programmers. Instead, we require the SECRO's pre and postconditions to meet Lemma 1.

Lemma 1. *Given an initial state s and a set of valid updates V_s, there exists an ordering of the updates that respects causality and all pre and postconditions.*

Using Lemma 1 we define *correctness* of replicas.

Definition B.1. A replica $r = \langle s, h\rangle$ is *correct* iff the replica is an instance of a SECRO whose pre and postconditions meet Lemma 1 and all updates from its history h are valid.

Given a replica $r = \langle s, h\rangle$ we can compute the replica's current state by successive applications of the updates from its history, denoted $s \lhd h$. Listing 9 defines the generic $\lhd$ operator which applies a list of updates on some state, in Haskell. Updates are functions from state to state.

```
type Update state = state -> state
(◁) :: state -> [Update state] -> state
s ◁ h = foldl (\state update -> update state) s h
```

Listing 9: Definition of the $\lhd$ operator to compute a replica's current state given its initial state and update history.

We now prove that correct replicas guarantee progress.

Theorem B.1. *For any correct replica $r_1 = \langle s, h_1\rangle$ and any valid update u issued by some other correct replica $r_2 = \langle s, h_2\rangle$ while being in state $t = s \lhd h_2$, delivering the update u at replica r_1 does not fail:*

$$\forall r_1, r_2, u : r_1 = \langle s, h_1\rangle \wedge r_2 = \langle s, h_2\rangle \wedge t = s \lhd h_2 \wedge \\ u \in V_t \implies [u] \Downarrow r_1 \neq None$$

Proof. Let V be the set of valid updates observed by replica r_1, i.e. V contains all (and only those) updates from its history h_1. Upon delivering the update u at replica r_1, $[u] \Downarrow r_1$, the algorithm generates all linear extensions of the updates in $V' = V \cup \{u\}$ (Line 3 in Algorithm 1). Those linear extensions are all the serialisations of the updates that respect the causality of the updates. The algorithm then continues by searching for the first valid extension (Lines 7 to 32). Since r_1 is a correct replica, at least one linear extension is valid (cf. Lemma 1). The algorithm will find that linear extension and return the updated replica on Line 28, $[u] \Downarrow r_1 = Some\ r_1'$. Hence, delivering a valid update at a correct replica cannot fail. □

Appendix C. Transitive closure of concurrent operations

Recall from Algorithm 1 in Section 7.1 that checking preconditions and postconditions requires computing the transitive closure of concurrent operations. We now formally define the transitive closure of concurrent operations.

Definition C.1. An operation $m_1 = (o_1, p_1, a_1, c_1, id_1)$ happened before an operation $m_2 = (o_2, p_2, a_2, c_2, id_2)$ iff the logical timestamp of m_1 happened before the logical timestamp of m_2: $m_1 \prec m_2 \iff c_1 \prec c_2$.

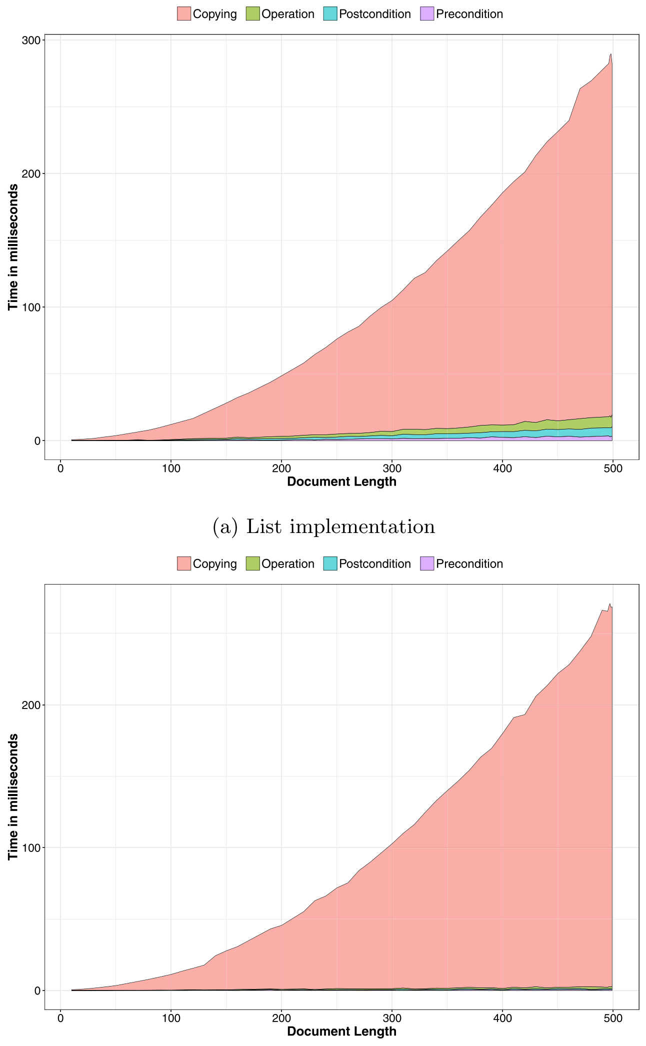
Definition C.2. Two operations m_1 and m_2 are concurrent iff neither one happened before the other [17]: $m_1 \parallel m_2 \iff m_1 \not\prec m_2 \wedge m_2 \not\prec m_1$.

Definition C.3. We define $\parallel^+$ as the transitive closure of $\parallel$.

Definition C.4. The set of all operations that are transitively concurrent to an operation m with respect to a history h is defined as: $TC(m, h) = \{m' \mid m' \in h \wedge m' \parallel^+ m\}$.

Appendix D. Detailed execution time of the text editor

In Section 8.4.2 we found that the SECRO implementation that uses a linked list is faster than its tree-based counterpart. To determine the cause of this counterintuitive observation, we measure the different parts that make up the total execution time:



(a) List implementation

(b) Tree implementation

Fig. D.9. Detailed execution time for appending characters to the SECRO text editor. The replica is never committed. The plotted execution time is the average taken from a minimum of 30 samples. Samples affected by garbage collection are discarded.

Execution time of operations Total time spent on append operations.

Execution time of preconditions Total time spent on preconditions.

Execution time of postconditions Total time spent on postconditions.

Copy time Due to the mutability of JavaScript objects our prototype implementation in CScript needs to copy the state before validating the potential history. The total time spent on copying objects (i.e. the document state) is the copy time.

Figs. D.9a and D.9b depict the detailed execution time for the list and tree implementations respectively. The results show that the total execution time is dominated by the copy time. We observe that the tree implementation spends more time on copying the document than the list implementation. The reason being that copying a tree entails a higher overhead than copying a linked list as more pointers need to be copied. Furthermore, the tree implementation spends considerably less time executing operations, preconditions and postconditions, than the list implementation. This results from the fact that the balanced tree provides logarithmic time operations.

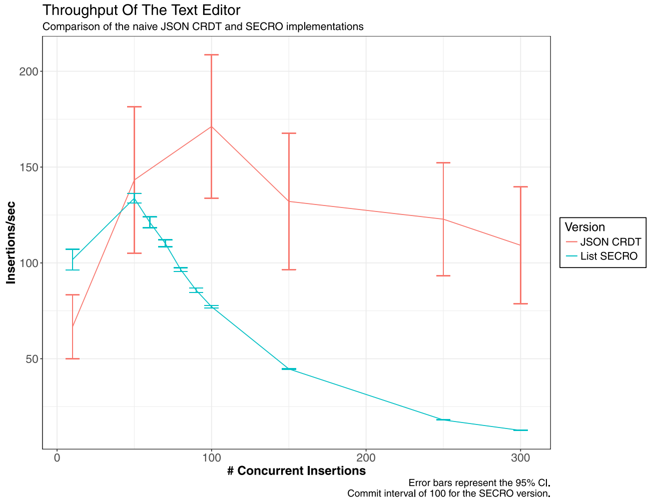


Fig. E.10. Throughput of the list-based SECRO and JSON CRDT text editors, in function of the number of concurrent operations. The SECRO version committed the document replica at a commit interval of 100. Error bars represent the 95% confidence interval for the average of 30 samples.

Unfortunately, the time overhead incurred by copying the document kills the speedup we gain from organising the document as a tree. This is because each insertion inserts only a single character but requires the entire document to be copied.

Appendix E. Throughput of the text editor

The experiments presented in Section 8 focused on the execution time of sequential operations on a single replica. To measure the throughput of the text editors under high computational loads we also perform distributed benchmarks. To this end, we use 10 replicas (one on each node of the cluster) and let them simultaneously perform operations on the text editor. The operations are equally spread over the replicas. We measure the time to convergence, i.e. the time that is needed for all replicas to process all operations and reach a consistent state. Note that replicas reorder operations locally, hence, the throughput depends on the number of operations and is independent of the number of replicas.

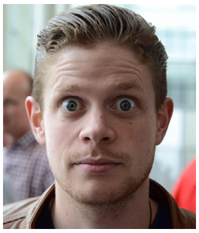
Fig. E.10 depicts how the throughput of the list-based text editor varies in function of the load. We observe that the SECRO text editor scales up to 50 concurrent operations, at which point it reaches its maximal throughput. Afterwards, the throughput quickly degrades. On the other hand, the JSON CRDT implementation achieves a higher throughput than the SECRO version under high loads (100 concurrent operations and more). Hence, the JSON CRDT text editor scales better than the SECRO text editor. However, SECROs are truly general-purpose which allowed us to organise documents as balanced trees of characters, which is not possible using JSON CRDTs.

References

[1] P.S. Almeida, A. Shoker, C. Baquero, Efficient state-based CRDTs by delta-mutation, in: A. Bouajjani, H. Fauconnier (Eds.), Int. Conference on Networked Systems, Springer-Verlag, Agadir, Morocco, 2015, pp. 62–76.

[2] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, IPA: Invariant-preserving applications for weakly consistent replicated databases, Proc. VLDB Endow. 12 (4) (2018) 404–418.
[3] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, M. Shapiro, Putting consistency back into eventual consistency, in: 10th European Conference on Computer Systems, in: EuroSys '15, 2015, 6:1–6:16.
[4] V. Balegas, C. Li, M. Najafzadeh, D. Porto, A. Clement, S. Duarte, C. Ferreira, J. Gehrke, J. Leitao, N. Preguiça, et al., Geo-replication: Fast if possible, consistent if necessary, IEEE Data Eng. Bull. 39 (1) (2016) 12.
[5] P.A. Bernstein, S. Burckhardt, S. Bykov, N. Crooks, J.M. Faleiro, G. Kliot, A. Kumbhare, M.R. Rahman, V. Shah, A. Szekeres, J. Thelin, Geo-distribution of actor-based services, Proc. ACM Program. Lang. 1 (OOPSLA) (2017) 107:1–107:26.
[6] E. Brewer, Towards robust distributed systems, in: 19th Annual ACM Symp. on Principles of Distributed Computing, in: PODC '00, 2000, p. 7.
[7] E. Brewer, CAP twelve years later: How the "Rules" have changed, Computer 45 (2012) 23–29.
[8] S. Burckhardt, M. Fähndrich, D. Leijen, B.P. Wood, Cloud types for eventual consistency, in: 26th European Conference on Object-Oriented Programming, in: ECOOP'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 283–307.
[9] K. De Porre, Cscript repository, 2018, https://gitlab.com/iot-thesis/framework/tree/master (Accessed: 09-10-2019).
[10] K. De Porre, F. Myter, C. De Troyer, C. Scholliers, W. De Meuter, E. Gonzalez Boix, Putting order in strong eventual consistency, in: J. Pereira, L. Ricci (Eds.), Distributed Applications and Interoperable Systems, Springer International Publishing, Cham, 2019, pp. 36–56.
[11] P.T. Eugster, P.A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Comput. Surv. 35 (2) (2003) 114–131, http://dx.doi.org/10.1145/857076.857078.
[12] S. Gilbert, N. Lynch, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, SIGACT News 33 (2) (2002) 51–59.
[13] V.B.F. Gomes, M. Kleppmann, D.P. Mulligan, A.R. Beresford, Verifying strong eventual consistency in distributed systems, Proc. ACM Program. Lang. 1 (OOPSLA) (2017) 109:1–109:28.
[14] R. de Juan-Marín, H. Decker, J.E. Armendáriz-Íñigo, J.M. Bernabéu-Aubán, F.D. Muñoz-Escoí, Scalability approaches for causal multicast: a survey, Computing 98 (9) (2016) 923–947.
[15] M. Kleppmann, A.R. Beresford, A conflict-free replicated JSON datatype, IEEE Trans. Parallel Distrib. Syst. 28 (10) (2017) 2733–2746.
[16] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.
[17] L. Lamport, The temporal logic of actions, ACM Trans. Program. Lang. Syst. 16 (3) (1994) 872–923.
[18] C. Li, N. Preguiça, R. Rodrigues, Fine-grained consistency for geo-replicated systems, in: 2018 USENIX Annual Technical Conference (USENIX ATC 18), USENIX Association, Boston, MA, 2018, pp. 359–372.
[19] C. Meiklejohn, P. Van Roy, Lasp: A language for distributed, coordination-free programming, in: 17th Int. Symp. on Principles and Practice of Declarative Programming, in: PPDP '15, 2015, pp. 184–195.
[20] M. Milano, A.C. Myers, MixT: A language for mixing consistency in geodistributed transactions, in: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI 2018, ACM, New York, NY, USA, 2018, pp. 226–241.
[21] H. Miller, P. Haller, M. Odersky, Spores: A type-based foundation for closures in the age of concurrency and distribution, in: R. Jones (Ed.), ECOOP 2014, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 308–333.
[22] F. Myter, C. Scholliers, W. De Meuter, A capable distributed programming model, in: Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, in: Onward! 2018, ACM, New York, NY, USA, 2018, pp. 88–98.
[23] Node.js: a javascript runtime built on chrome's v8 javascript engine, 2020, https://nodejs.org (Accessed: 15-01-2020).
[24] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, A comprehensive study of Convergent and Commutative Replicated Data Types, Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, 2011, p. 50.
[25] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, Conflict-free replicated data types, in: X. Défago, F. Petit, V. Villain (Eds.), 13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems, in: SSS'11, Springer-Verslag, Grenoble, France, 2011, pp. 386–400.
[26] C. Sun, C. Ellis, Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements, in: Proc. of the 1998 ACM Conference on Computer Supported Cooperative Work, CSCW '98, 1998, pp. 59–68.
[27] Sweet.js - hygienic macros for javascript, 2020, https://www.sweetjs.org (Accessed: 15-01-2020).
[28] A.S. Tanenbaum, M. Van Steen, Distributed Systems: Principles and Paradigms, second ed., Prentice-Hall, Upper Saddle River, New Jersey, USA, 2007.
[29] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, C.H. Hauser, Managing update conflicts in bayou, a weakly connected replicated storage system, in: M.B. Jones (Ed.), 15th ACM Symp. on Operating Systems Principles, in: SOSP '95, 1995, pp. 172–182.
[30] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, W. De Meuter, Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks, in: Proceedings of the XXVI International Conference of the Chilean Society of Computer Science, in: SCCC '07, Iquique, Chile, 2007, pp. 3–12, http://dx.doi.org/10.1109/SCCC.2007.4.
[31] P. Viotti, M. Vukoliundefined, Consistency in non-transactional distributed storage systems, ACM Comput. Surv. 49 (1) (2016).
[32] W. Vogels, Eventually consistent, Commun. ACM 52 (1) (2009) 40–44.

Kevin De Porre is a PhD student at the Software Languages Lab (SOFT) of the Vrije Universiteit Brussel (VUB) in Belgium. His research focuses on distributed systems, more specifically on language abstractions for data replication in peer-to-peer systems and consistency models for replicated data. Contact him at kdeporre@vub.be.



Florian Myter is a PhD student at the Software Languages Lab (SOFT) of the Vrije Universiteit Brussel (VUB) in Belgium. His main research area is distributed programming and more concretely the design and implementation of programming techniques to deal with distributed state. Contact him at fmyter@vub.be.

Christophe Scholliers is professor in foundations of programming languages at Ghent University. His current research is mainly situated in the field of parallel and distributed programming language abstractions. Contact him at christophe.scholliers@ugent.be.

Elisa Gonzalez Boix is an Associate Professor at the Software Languages Lab (SOFT) of the Vrije Universiteit Brussel (VUB) in Belgium. She obtained her Master in Informatics Engineering in 2004 from the Universitat Politecnica de Catalunya (Spain) and her PhD in Sciences in 2012 from VUB on programming language abstractions and tools for handling partial failures in distributed applications running on mobile ad hoc networks. Her PhD heavily relied on reflection and meta-level programming. Since 2014, she leads a group on concurrent and distributed systems, studying programming abstractions and dynamic software tools like debuggers. You can contact her at egonzale@vub.be.