Version Control Is for Your Data Too

Gowtham Kaki
Purdue University, USA
gkaki@purdue.edu

KC Sivaramakrishnan
IIT Madras, India
kcsrk@iitm.ac.in

Suresh Jagannathan
Purdue University, USA
suresh@cs.purdue.edu

Abstract

Programmers regularly use distributed version control systems (DVCS) such as Git to facilitate collaborative software development. The primary purpose of a DVCS is to maintain integrity of source code in the presence of concurrent, possibly conflicting edits from collaborators. In addition to safely merging concurrent non-conflicting edits, a DVCS extensively tracks source code provenance to help programmers contextualize and resolve conflicts. Provenance also facilitates debugging by letting programmers see diffs between versions and quickly find those edits that introduced the offending conflict (e.g., via `git blame`).

In this paper, we posit that analogous workflows to collaborative software development also arise in distributed software execution; we argue that the characteristics that make a DVCS an ideal fit for the former also make it an ideal fit for the latter. Building on this observation, we propose a distributed programming model, called CARMOT that views distributed shared state as an entity evolving in time, manifested as a sequence of persistent versions, and relies on an explicitly defined *merge semantics* to reconcile concurrent conflicting versions. We show examples demonstrating how CARMOT simplifies distributed programming, while also enabling novel workflows integral to modern applications such as blockchains. We also describe a prototype implementation of CARMOT that we use to evaluate its practicality.

2012 ACM Subject Classification Computing methodologies → Distributed programming languages; Software and its engineering → Software configuration management and version control systems; Software and its engineering → API languages

Keywords and phrases Replication, Distributed Systems, Version Control

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.8

1 Introduction

Building distributed applications is hard. The crux of the problem is the management of concurrent updates to distributed shared state that maintain user-level invariants and properties. The problem is especially pronounced in the context of modern data-intensive applications, which replicate large amounts of data across geographically diverse locations to enable trust decentralization, guarantee low latency access to state, and provide high availability even in the face of node and network failures. In general, these systems allow each replica instance of a distributed application to concurrently accept updates to shared data, which could potentially conflict with other updates, and consequently violate data integrity. In addition, transient faults in the underlying network, such as network partitions, message reorderings etc., can yield counter-intuitive anomalous executions that are hard to predict and even harder to preempt [9, 25, 15]. While conventional concurrency control criteria, such as linearizability and serializability, are designed to preclude such executions, applications are reluctant to impose them given their prohibitive cost in a distributed setting

© Gowtham Kaki, KC Sivaramakrishnan, and Suresh Jagannathan;
licensed under Creative Commons License CC-BY
3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 8; pp. 8:1–8:18

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

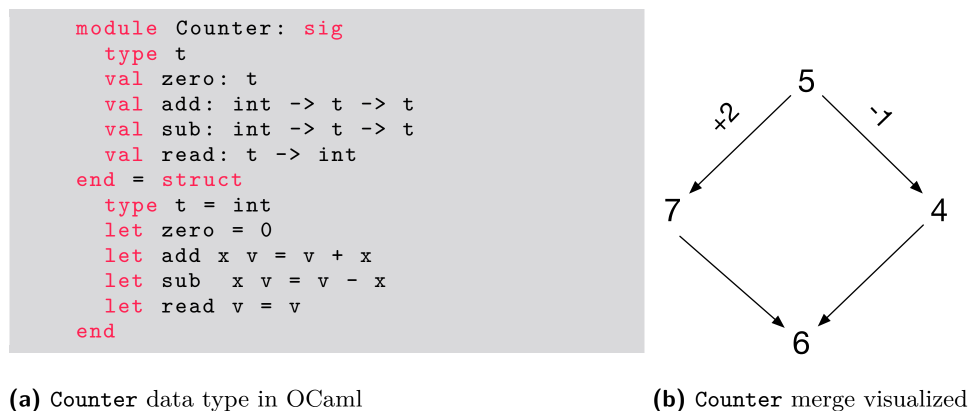
(an observation succinctly captured in the CAP theorem [12]). Instead, applications mostly operate within the weak guarantees provided by an asynchronous state replication model, occasionally resorting to stronger forms of concurrency control for "risky" operations, using *ad hoc* and error-prone reasoning to distinguish between them. Another available alternative is to restructure applications around a library of distributed data structures that are carefully designed by experts and proven to be correct under an asynchronous distributed setting[1]. Such re-engineering, however, may not always be feasible since applications often use bespoke data structures to serve specific needs. Consequently, the cognitive overhead in building distributed applications remains high, limiting program development to distributed system experts.

It may therefore come as a surprise to note that programmers, as humans, are exposed to the complex realities of distributed computing almost on a daily basis, and they seem to be doing just fine. Almost all programmers these days use some form of a distributed version control system (DVCS), such as Git or Mercurial, and when doing so, emulate the logic of a distributed application consciously or otherwise. A DVCS lets a programmer safely *merge* concurrent *versions* of source code created by her collaborators working independently and concurrently, fork-off her own *branch* from any existing version to work in isolation, group together multiple related changes as a single *commit* to be pushed to a *remote*, look at the provenance information to understand how the code has evolved across multiple versions, and track which collaborator is responsible for which piece of code. The overwhelming adoption of DVCS as a paradigm for distributed and collaborative software development is indicative of the utility of the model it supports. Much of its attractiveness stems from the intuitive mental picture it offers the developer to reason about the integrity of source code as it evolves in the face of concurrent modifications.

In this paper, we ask whether the benefits of DVCS can be transplanted to manage *data* in addition to source code. In particular, we propose the thesis that building safe distributed applications would be dramatically simpler if concurrent conflicting updates to application state are explicitly recognized and resolved, rather than preempted. Supporting this thesis requires addressing a number of challenging questions: (a) How can the provenance of data be systematically exploited to automate the resolution of merge conflicts? (b) Would the ability to fork-off a version of the state, and group together multiple changes to the state as a single commit give useful transactional semantics? (c) Can we generate sufficiently high-level provenance data to serve as a transaction ledger and satisfy the auditing requirements of emerging applications such as those built on distributed blockchains ledgers? It is to explore the answers to these questions that we conceived CARMOT- a distributed programming model build around the same concepts as distributed version control systems.

At the core of CARMOT is the principle that any data structure that ascribes a well-defined merge semantics to merge its concurrent versions becomes a distributed data structure. Our experience with DVCS informs us that provenance information greatly helps in contextualizing merges and resolving conflicts. True to that spirit, CARMOT allows a data structure to define a merge semantics for its concurrent versions in the context of their *lowest common ancestor* (LCA) version, resulting in a three-way merge. Thus, any ordinary data structure equipped with a three-way merge function becomes eligible to be a distributed data structure. As we describe in Sec. 2, the simplicity of this criterion lets us build bespoke distributed data structures and develop applications around such data structures with a relative ease. We

[1] Analogous, for example, to carefully designed lock-free data structures in a concurrent programming language [24, 13]



(a) `Counter` data type in OCaml **(b)** `Counter` merge visualized

Figure 1 The `Counter` example

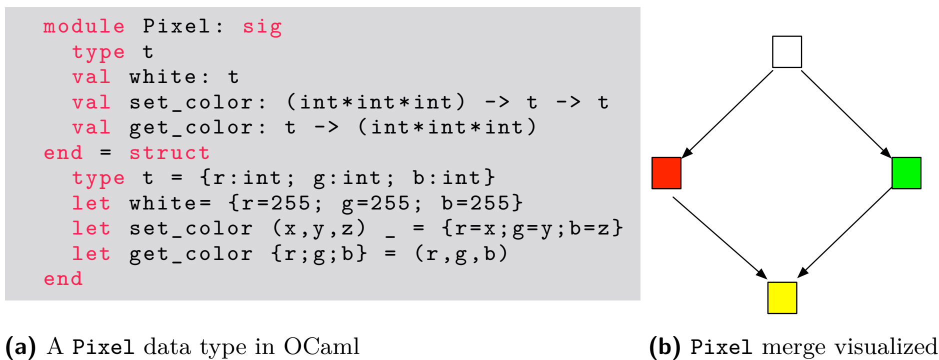
subsequently demonstrate how CARMOT can build on the branch-and-merge model of DVCS to define a transactional semantics with a well-defined isolation model at no additional cost. Lastly, we show that extensive provenance tracking, similar to a DVCS, helps CARMOT naturally express blockchain applications, which otherwise have to rely and specialized shim layer, such as the Hyperledger Fabric [2]. However, all the aforementioned benefits would amount to naught if CARMOT as a programming model cannot be realized in an asynchronous distributed setting. In Sec. 3 we describe a prototype implementation of CARMOT that sits atop Git that can actually run CARMOT distributed applications seamlessly with low overheads.

2 An Overview of Programming with Version Controlled Data

2.1 Version Control-Inspired Replication

Fig. 1a shows a simple counter data type in OCaml that admits additions and subtractions. Suppose Alice wants to use the counter in a distributed setting, meaning she wants to replicate the counter state across various machines, allow the state to be updated concurrently on each replica, and let the updates be propagated asynchronously to other (remote) replicas. One way she could achieve this is by maintaining a log of operations performed at each replica, periodically flushing the log to other replicas either on demand or by default. A replica receiving a remote operation has to apply it on the local state to keep it consistent with the remote state. Operations may be received and applied at different replicas in different orders, but since additions and subtractions commute, the resultant counter state would *eventually* guaranteed to be the same on all the replicas. Indeed, this is how asynchronous replication is often implemented in distributed applications [11, 23, 22, 16, 25].

An alternative take on replication would be one where Alice views the counter state as data being managed by a version control system, and sees various replicas as her collaborators creating concurrent versions of the counter state. In order to fetch her collaborators' updates, she would then be obligated to write a `merge` function that reconciles concurrent versions of the counter in the context of their lowest common ancestor (LCA). An example merge scenario is shown in Fig. 1b, where counter versions (with values) 7 and 4 have evolved concurrently from the original version 5. Alice could observe that the concurrent versions represent an addition of 2 and a subtraction of 1 on the ancestor state, and hence may choose to reconcile them as a single operation performing an addition of 1 on 5, to compute



(a) A `Pixel` data type in OCaml **(b)** `Pixel` merge visualized

Figure 2 The `Pixel` example

the merged version as 6. Alice's merge logic could be generalized as the following `merge` function:

```
let merge lca v1 v2 =
   lca + (v1 - lca) + (v2 - lca)
```

Indeed, such `merge` function is what CARMOT requires to promote a counter data type to the status of a distributed data type.

The version control-inspired view of replication really stands out when Alice decides to add a `mult` operation to multiply the value of the counter. Perhaps Alice wants to use the counter to count the account balance in a banking application, and she needs `mult` to compute the interest on the balance. She defines `mult` straightforwardly:

```
let mult x v = x * v
```

The `mult` operation serves Alice well as long as she uses her counters on a single machine. In a distributed setting however, under the conventional model of replication, Alice may see unexpected results as `mult` doesn't commute with `add` and `sub`, thus yielding different counter states on different machines. In other words, Alice's counter implementation, which is still correct in a single machine setting, is no longer correct in a distributed setting. She is now forced to abandon her recent additions to `Counter`, and restructure her banking application to express interest addition in different terms, perhaps using `add` to specify the increase in balance. Version control-based replication on the other hand lets Alice continue using her latest counter implementation (with `mult`) in a distributed setting as the counter's merge semantics already capture the effect of interest computation in terms of an increase in balance.

2.2 Application-specific Bespoke Merges

We provide further motivation using a more colorful data type - a pixel. We might write a pixel data type in OCaml as a triple with three fields (Fig. 2a), each standing for a red, blue and green color components respectively (following the rgb coloring scheme). `Pixel` supports a single operation - `set_color` that sets the color of the pixel to the given rgb values. A `get_color` function returns the color triple of the pixel.

Alice originally defined the pixel data type to use in her drawing board application she calls `Canvas`. She now wants to add collaborative drawing features to `Canvas`, and hence is interested in replicating the state of a pixel. Following the conventional model of replication, Alice sets up the pixel data type to asynchronously propagate `set_color` operations across

```
let merge lca ({r=r1;g=g1;b=b1} as v1) ({r=r2;g=g2;b=b2} as v2) =
  let mix (x,y) = min (x+y) 255 in
  if lca = v1 then v2
  else if lca = v2 then v1
  else {r=mix(r1,r2); g=mix(g1,g2); b=mix(b1,b2)}
```

Figure 3 Pixel merge function

replicas, but quickly discovers that this leads to diverging pixel states across replicas. For instance, when Bob colors the pixel green on one replica, and Alice colors it red on a different replica, each's `set_color` operation may overwrite the other's on the remote replica, leading to diverging states. Unfortunately, unlike the previous example, Alice doesn't know how to redefine the `set_color` operation or restructure the `Canvas` application to solve the problem of diverging states. Nor can she find an appropriate consistency model [9, 26] weaker than linearizability that preempts such anomalous executions even if it comes at some expense. Alice is therefore stuck.

With version control-based replication, however, Alice starts with the assumption that her collaborators could create concurrent versions of the pixel state, and specifies the logic to merge such versions under the context of their LCA version (Fig. 3). She could, for example, reconcile the concurrent updates to the pixel color by using an additive color mixing scheme to mix the colors (Fig. 2b). When there are no concurrent updates, i.e., when at least one of the two versions is same as the ancestor version (thus causally preceding the other version), the successor version trivially becomes the result of the merge.

2.3 Transactional Semantics

Because version control-based replication lets Alice promote her pixel type to a distributed data type, she moves ahead with the development of her collaborative drawing app - `Canvas`. She defines a canvas type as a two-dimensional composition of pixels:

```
type canvas = Pixel.t list list
```

Alice initially emulates free-hand drawing by coloring individual pixels via `Pixel.set_color`, but soon realizes that it would be nice to have a few basic shapes, such as a circle or a rectangle, that she could draw in a single stroke. The functionality requires several pixels to be colored atomically and in isolation; atomically because Alice's collaborators should only see her draw a full circle or a rectangle rather than coloring an assortment of pixels, and isolation because Alice would like to draw the circle completely before she deals with conflicting writes from her collaborators[2]. In other words, the application needs transactional semantics. With version control-based replication , Alice gets that for free. An atomic action, such as drawing a basic shape, can be performed on Alice's local version of canvas as a series of `set_color` operations on pixels making up the basic shape. Only the resulting canvas version is committed and pushed to Alice's collaborators, effectively enforcing the atomicity of writes. Furthermore, Alice wouldn't "pull" her collaborator's updates while her basic shape is in progress, thus guaranteeing the isolation of basic shape drawing operation.

[2] Alice could, for example, define a `merge` function for canvas that removes or retains an entire basic shape in case of a conflicting write. For pixels not part of a basic shape, she could default to the `Pixel.merge` function

2.4 Distributed Ledger

An important benefit of the CARMOT programming model is its inherent support for provenance. Such support is critical in applications where some form of consensus is required. In this applications, divergent states found on different replicas can be merged to a convergent (consensus) value based on application semantics, A particularly noteworthy instance of such applications is a distributed ledger.

As typically conceived, a distributed ledger is a safety-critical distributed data type that requires every node to maintain an untamperable ledger of operations that were ever performed on the shared state it represents. The ledger is colloquially called a blockchain due to its organization in terms of a sequence of blocks, where each block refers to its predecessor in the sequence. The untamperability of blocks is ensured by making the blocks content-addressable, i.e., by making the address of the block depend on its contents (e.g., an address could be the block's SHA1 hash as explained in the following section). Thus, tampering with a block results in the construction of a new block with a different address that does not belong to the chain, hence leaving the chain unchanged. Provenance information available to programmers in a version control-based replication model allows us to build untamperable distributed ledgers (blockchains) effortlessly, as we shall demonstrate through an extensive example.

2.4.1 Blockchain Preliminaries

At a high level, a blockchain represents a distributed bank-like application involving multiple *peers*, each maintaining a replica of a shared ledger of transactions. The ledger represents some form of consensus among the peers about the set of valid transactions performed thus far (since the beginning of time), and their relative order. Note that the distributed ledger is the *only* source of truth in a blockchain application; all relevant information (e.g., the account balances (in a banking application) is computed by reading the contents of the ledger. A transaction in a blockchain application transfers something of value between anonymous users. The thing of value could be a Bitcoin, but we simply refer to it as money in this discussion. If a transaction makes its way into the ledger, it is said to have been *confirmed*, meaning that there is a consensus that the transaction is valid, i.e., it does not engage in illegitimate behaviors, such as double spending available money. A transaction, when submitted, is initially unconfirmed, and the application maintains a set of such yet-unconfirmed transactions, which is also replicated across peers. Simply put, the task of a peer is to pick a yet-unconfirmed transaction, validate it, and if it is deemed to be valid, confirm it by adding to the distributed ledger. However, doing so in an uncontrolled fashion in a large system of peers (e.g., Bitcoin) leads to the divergence of the ledger state across replicas, resulting in a disagreement among peers about the contents of the ledger, and the consequent confirmation of invalid transactions. To prevent this, blockchain applications define a *soft* consensus protocol that limits the rate at which transactions can be appended to the ledger, and specifies how to resolve conflicts in case there are competing versions of the ledger. Most public blockchains, such as Bitcoin and Ethereum, employ *proof-of-work* as the rate limiting mechanism, where a peer has to solve a computationally hard problem to earn the right to append to the ledger (and a financial reward for *mining* the solution). The solution to this hard problem is also appended to the ledger to let other peers verify the solution. Unlike computing proof-of-work, verification is expected to be easy, i.e., the computational problem should ideally be NP-complete. In practice, various kinds of problems are used, whose description is out of scope for this paper. The problem we choose for this example

```
type txn = {timestamp: float;
            sender: pub_key;
            receiver: pub_key;
            amount: int}
```

```
type block = {txns: txn set;
              timestamp: float;
              proof:int64}

type t = {txns: txn set;
          chain: block list}
```

Figure 4 Blockchain type definitions in OCaml

is explained below. To ameliorate transaction confirmation latency, a peer which earns the right to append to the ledger is allowed to append a *block* of transactions, confirming them all in a single action. The new block links to the previous block, thus making the ledger a blockchain.

Note that proof-of-work is only a rate limiting mechanism; it makes concurrent appends to the ledger unlikely, but not impossible. Occasionally, when two peers simultaneously compute a proof-of-work, they both append their respective blocks to the ledger, resulting in a fork. Peers aware of only one of the two forked versions continue to mine and append blocks to their respective versions. At some point, if a peer becomes aware of two competing versions, it chooses one version over the other based on a predefined set of heuristics. For instance, if there are two competing chains of Bitcoin ledgers, then the longer chain is chosen as it represents more work confirming large number of transactions than the shorter one. The transactions of the shorter chain that do not belong to the longer chain are re-added to the pool of unconfirmed transactions and need to be confirmed again by a mining peer[3].

2.4.2 A Blockchain Application in OCaml

Having introduced a blockchain's conceptual underpinnings, we now describe how we can support its necessary functionality in a version control-based programming model. Fig. 4 shows the type definitions needed to build a simple blockchain application in OCaml. A transaction is simply a record documenting the transfer of a given amount of money from a sender to a receiver, both identified only by their public key. As mentioned earlier, a blockchain ledger is a chain of blocks, where each block consists of a set of transactions being confirmed, and a proof-of-work justifying the block's presence in the chain. Application state is defined via type `t`, which is a record containing a set `txns` of as-of-yet-unconfirmed transactions, and a list of blocks named `chain`, which is the blockchain.

Operations can be defined that map one application state to another. The `new_txn` operation, for instance, creates a new transaction with the given user keys and the amount, and adds it to the pool of unconfirmed transactions. Its type is as shown below:

```
val new_txn: pub_key -> pub_key -> int -> t -> t
```

The second, and most important operation, is `mine_block` that composes a new block using the available pool of unconfirmed transactions, mines a proof-of-work, and adds the block to the chain. Fig. 5 shows the (abridged) code. The function first filters the set of valid transactions from the available pool of unconfirmed transactions using the function

[3] It is thus possible for a confirmed transaction to become unconfirmed again. Bitcoin therefore defines *number of confirmations* of a transaction based on how many blocks deep the transaction is inside the ledger. The greater the number of confirmations, the deeper the transaction sits inside the ledger, and the less likely it is to become unconfirmed again.

```
let mine_block my_key t =
  let valid_txns =
    filter_valid_txns t.txns in
  let last_block = hd t.chain in
  let last_proof =
      last_block.proof in
  let proof = proof_of_work
                last_proof in
  let ts = gettimeofday () in
  let reward_txn =
        {timestamp=ts;
         sender=genesis_key;
         receiver=my_key;
         amount=25;} in
  let block =
      {txns=Set.add reward_txn
                valid_txns;
       timestamp=ts;
       proof=proof} in
  let txns' = Set.diff t.txns
              valid_txns in
  let chain' = block::t.chain in
  {txns=txns'; chain=chain'}

let valid_txn chain txn = ...
```

```
let filter_valid_txns =
  Set.filter
    (valid_txn t.chain)

let valid_proof last_proof proof =
  let str1 = Int64.to_string
               last_proof in
  let str2 = Int64.to_string
               proof in
  let str = str1^str2 in
  let hex = SHA1.to_hex @@
      SHA1.digest_string str in
  String.sub hex 0 3 = "000"

let proof_of_work last_proof =
  let rec loop_iter i =
    if i >= Int64.max_int
    then failwith "No proof!"
    else
      if valid_proof last_proof i
      then i
      else loop_iter (i + 1) in
  loop_iter 0
```

Figure 5 `mine_block` - a function that creates a new block, mines a proof-of-work, and adds it to the chain. Other relevant functions are also shown.

`filter_valid_txns` (shown on the right), which in turn uses `valid_txn` predicate of type shown below (definition elided in Fig. 5):

```
val valid_txn: block list -> txn -> bool
```

Given a blockchain ledger and a transaction, the function returns true if and only if the transaction is legitimate under the ledger, i.e., if and only if the account balance of the sender, computed from the ledger, is enough to carry out the transaction. Once the set of valid transactions is computed, `mine_block` proceeds to confirm them by computing the proof-of-work necessary to create a new block. In this example, we define proof-of-work as a solution to p_{i+1} in the following equation, where p_i denotes the proof-of-work of the previous block (`last_block.proof` in Fig. 5), and $\cdot$ denotes the (string) concatenation operation:

$$\texttt{sha1}(p_i \cdot p_{i+1}) < 2^{148}$$

The function `valid_proof` implements the above check in a slightly different form, given a p_{i+1} (`proof`) and p_i (`last_proof`). Instead of checking if the hash is less than 2^{148}, it checks if the leading 3 hex digits of the 40-digit SHA1 hash are zero. Given that it is in general impossible to invert a SHA1 hash in polynomial time, we solve the above equation by painstakingly iterating through all possible values of p_{i+1}, which in this case is every 64-bit integer, until we find the solution. The corresponding logic is implemented by the function `proof_of_work`.

Once the proof-of-work is computed, `mine_block` has everything it needs to compose a new block and add it to the chain. To incentivize mining, blockchain protocols allow mining peers to add a special transaction rewarding themselves a fixed pre-determined value. The `reward_txn` in Fig. 5 denotes such a transaction, which uses a special "genesis user" as the source of such rewards. The transaction is added to the set of valid transactions, following which the new block is composed and added to the chain. The transactions that could not be validated (`txns'`) are left in the pool of unconfirmed transactions.

```
let merge old v1 v2 =
  let oldc = old.chain in
  let v1c = v1.chain in
  let v2c = v2.chain in
  let x = valid_extension
            oldc v1c in
  let y = valid_extension
            oldc v2c in
  match x,y with
  | None, Some new2 -> v2
  | Some _, None -> v1
  | Some _, Some _
    when v1=v2 -> v1
  | Some _, Some new2
    when len v1c > len v2c ->
    add (union new2 v2.txns) ~to:v1
  | Some new1, Some _
    when len v1c < len v2c ->
    add (union new1 v1.txns) ~to:v2
  | Some new1, Some new2 ->
    let add_prf a b = a+b.proof in
    let prf_sum1 =
      fold_left add_prf 0 new1 in
    let prf_sum2 =
      fold_left add_prf 0 new2 in
    if prf_sum1 > prf_sum2
    then add (union new2 v2.txns)
           ~to:v1
    else add (union new1 v1.txns)
           ~to:v1
  | None, None -> error()
```

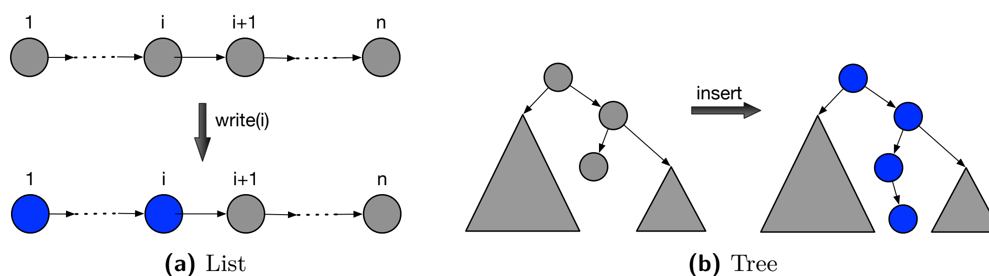
```
let valid_extension oldc newc =
  try
    let newbs = get_prefix newc
                  ~suffix:oldc in
    let _ = fold_right
      (fun b chain ->
        if valid_block chain b
        then b::chain
        else raise Invalid_arg)
      newbs oldc in
    Some newbs
  with Invalid_arg _ -> None

let rec get_prefix v ~suffix:s =
  match v with
  | v when v = s -> []
  | x::xs when xs = s -> [x]
  | x::xs -> x::(get_prefix xs s)
  | [] -> raise (Invalid_arg)

let add (txns:txn set) ~to:t =
  Set.fold
    (fun txn t' ->
      let chn = t'.chain
      if confirmed_txn chn txn
      then t'
      else {t' with txns=
              Set.add txn t'.txns})
    txns t

let confirmed_txn chain txn = ...
```

Figure 6 `merge` function for a blockchain. Other relevant functions also shown.

Beyond `new_txn` and `mine_block`, a third `init` operation need also be defined to initialize the blockchain with a special "genesis block" that bootstraps the ledger with initial money (subsequent to genesis, the only way to add money to the system is through mining rewards). We elide discussion of `init` to focus on other aspects relevant to this paper.

2.4.3 Merge function

Blockchain applications are, by definition, distributed with the state of the ledger replicated across all the participating peers. To make our blockchain application distributed under a version control-based replication model, we provide a three-way merge function that merges the competing versions of blockchains given their provenance information in form of their lowest common ancestor (LCA) version. Fig. 6 shows the merge function. The merge strategy is based on the heuristic employed in Bitcoin to pick the longest valid chain among the competing chains. The merge function first checks that the two competing chains are indeed valid chains through the function `valid_extension`. The function ensures that the old chain (the LCA) is a suffix of the competing chain `newc`, and that each block in the newly added prefix is valid. Validity of a block is checked through the predicate `valid_block` (not shown), which in turn checks the `valid_txn` predicate for each transaction in the block. The valid prefix is then returned. If the chain `newc` is not a valid extension of the LCA, then `None` is returned. Note that `valid_prefix` uses the `get_prefix` function on lists (definition shown), which compares the lists for structural equality (e.g., `v = s`). While determining structural equality is in general expensive, it can be resolved in constant time for versions obtained



(a) List (b) Tree

Figure 7 Linked data structures composed of content-addressable objects

through CARMOT; we elaborate on this point in Sec. 3.

A notable aspect of `valid_extension` is that it uses provenance information available as the LCA version to determine if the chain has been tampered with. If the new version has tampered with the chain, for example, by sliding a new transaction in to an older block, then the LCA version `oldc` is no longer the suffix of the new version, leading `valid_extension` to return `None`.

In the `merge` function, if `valid_extension` returns `None` for one of the two chains, then the other (valid) chain is returned as the result of the merge. If both chains are valid but are not equal, then the longer one is picked. In such case, the transactions newly confirmed in the shorted chain are no longer considered confirmed, and need to be re-added to the unconfirmed pool (along with the transactions in the unconfirmed pool accompanying the shorter chain). This is done via the `add` function (shown) that adds the new transactions from the shorter chain and the corresponding unconfirmed pool, that are not confirmed by the longer chain, to the unconfirmed pool accompanying the longer chain. The predicate `confirmed_txn` (definition elided) returns true if and only if the transaction `txn` is listed in the chain `chain`. If both the competing chains are of equal length, then `merge` picks the chain whose extension w.r.t `oldc` was "harder" to compute, where hardness is assumed to be proportional to the value of the proof. Finally, if both chains are invalid, then `merge` throws an error, but this case never occurs in practice as one the two chains is a local chain which is guaranteed to be valid by `mine_block`.

We have thus far presented various examples of how version control-based replication lets one build non-trivial distributed applications by defining merge semantics in the convenient form of a three-way merge function. To make such applications operational, however, we need a programming interface that lets developers take advantage of the version control model to define and compose distributed computations around applications. CARMOT is such a programming model, whose details are presented in the next section.

3 Realizing The CARMOT Programming Model

We have realized the CARMOT programming model on top a basic Git programming abstraction in OCaml [21]. In the following, we discuss the salient aspects of CARMOT's implementation, describe the API it exposes, and demonstrate how one can use CARMOT to orchestrate complex distributed computations.

3.1 Content Addressability & Sharing

One of the key aspects of CARMOT is its reliance on a content-addressable file system/memory to store arbitrary objects. We call it the CARMOT *store*. Like any other store, CARMOT lets

one write an object, or a collection of linked objects to the store. However, unlike other stores, the address of an object in CARMOT store is its own SHA1 hash, which means mutating the object results in a new (version of) object being written to the store at a different address. This property lets CARMOT store multiple versions of linked data structures succinctly by sharing common objects, while simultaneously highlighting the *diff* between such versions. Fig. 7 illustrates.

Fig. 7a shows a linked list structure laid out on a content-addressable store. Each node is an object that stores some data and a link to the next object in the list, which is simply the latter's hash. For instance, a list object (call it A) could like the following:

```
{data = 24;
 next = "2aae6c35c94fcfb415dbe95f408b9ce91ee846ed"}
```

The value "2aa…6ed" is the SHA1 hash of the next object (call it B) in the list. If, for some reason, B's `data` is mutated, its hash value changes, making it necessary to update A's `next` field, which changing A's hash, and the update cascades. This scenario is depicted in Fig. 7a, where updating the i'th node in the list effectively creates a new version of the list with new objects for nodes 1 through i. The two versions of the list however share the common suffix containing nodes $i+1$ through n. The diff and sharing between the two versions is thus clearly highlighted in this representation. Fig. 7b describes similar scenario for a tree structure, where inserting an element creates a new version of the tree that only slightly differs from the previous version in the store (The diff is highlighted). Such succinct representation of diff, and its easy computation, lets CARMOT efficiently support replicated data structures over a network. Moreover, content addressability lets CARMOT support constant-time structural equality checks by simply comparing hashes instead of iterating through data structures.

Readers familiar with functional programming may find Fig. 7 reminiscent of the persistence and sharing aspects of functional data structures. Indeed, hash-linked data structures (as described above) and functional data structures are similar in that respect. However, one notable difference is that the sharing in hash-linked data structures is based on the primitive notion of *common content*, rather than data dependence, or other similar programmatic notions. Thus, one could create OCaml lists `[3;2;1]` and `[4;2;1]` independent of one another, whereas they share nothing on the OCaml heap, they nonetheless share the objects corresponding to the common suffix `[2;1]` on a content-addressable store. This property is crucially relied on by CARMOT to support distributed applications composed of high-level (OCaml) data structures, as described below.

3.2 The CARMOT API

CARMOT hides the full complexity of a version control system behind a monadic abstraction called a Versioned State (VST), and exposes just the right level of detail for programmers to reap the benefits of version control-based replication. The API comprising the CARMOT programming model is shown in Fig. 8. The VST monad couches a versioned state of type `'a`. For instance, `'a` can be a `Counter.t`, a `Pixel.t`, or even a composite data type such as `Counter.t list`. The representation of these types in the underlying CARMOT store is as described in the previous section, and not exposed by the `VST` interface. Instead the interface orchestrates computations around the versioned state, translating between the high-level and low-level representations as needed. Computations on the monad read the latest version, commit a new version, or pull and merge concurrent versions. The type `('a, 'b) t` represents a monadic computation on the version state `'a` that returns a result

```
module type VST = sig
  type ('a, 'b) t
  val return : 'b -> ('a, 'b) t
  val bind : ('a, 'b) t -> ('b -> ('a, 'c) t) -> ('a, 'c) t
  val get_current_version: unit -> ('a, 'a) t
  val with_init_version_do: 'a -> ('a, 'b) t -> 'b
  val with_forked_version_do: string -> ('a,'b) t -> 'b
  val fork_version: (string -> ('a,'b) t) -> ('a,string) t
  val sync_next_version: 'a -> string list -> ('a, 'a) t
end
```

Figure 8 The CARMOT API

of type `'b`. Operation `get_current_version` returns the high-level representation of the latest version of the state behind the monad. A programmer can initiate a computation against an explicitly provided initial version of the state using `with_init_version_do` API. Alternatively, computation can be run against an initial version forked from a remote using `with_forked_version_do` API. The string argument to the API is the URL of the remote. To fork off a new version of the state and run a (local) concurrent computation against it, `VST` provides the `fork_version` API. The argument to the function is the computation to run concurrently. The computation can expect the URL of the parent to be given as a string argument. The return value of `fork_version` is also a URL string that identifies the fork in the same terms as a remote. The underlying thread library is LWT [27]. Lastly, `sync_next_version` API (simply called `sync`) does two things. First, it commits the given `'a` argument as the new local version. Next, it pulls the latest versions from the given list of remotes (`string list`), and successively merges them to the latest local version, creating a later version each time. The latest version at the end of the merge sequence is returned. Note that if some of the remotes are unreachable during the operation, they are merely skipped. Consequently, `sync` is only guaranteed to sync with a subset of replicas. Functions `return` and `bind` are the usual monadic glue. Following the convention, we use the infix operator $\gg=$ to denote `bind`.

3.3 CARMOT Examples

To understand how the CARMOT API helps orchestrate distributed computations, let us reconsider the `Canvas` drawing application from the earlier section. Say Alice finished building `Canvas`, and would now like to use it to collaborate with her friends Bob and Cheryl. She could do that conveniently via the CARMOT API. A sample drawing session between the three collaborators is shown in Fig. 9 A possible execution of the session is visualized in Fig. 10. Assume that Alice starts her session on a 5×5 blank canvas, as shown below:

```
with_init_version_do (Canvas.new_blank 5 5) alice_f
```

Bob and Cheryl, on the other hand, start their sessions with a version forked from Alice's initial version as shown below (Bob's shown; Cheryl's is similar):

```
with_forked_version_do "alice" bob_f
```

Assume `Canvas.new_blank` returns a blank canvas of a given dimension, and the function `Canvas.draw_line` draws a line between the given pair of points (and returns the new canvas). Alice starts by reading the current version of the canvas, which is blank. Alice draws a red horizontal line from $(0,0)$ (top-left) to $(4,0)$ (top-right) using `Canvas.draw_line`. Meanwhile, Bob draws a green vertical line from $(0,0)$ to $(0,4)$, and Cheryl draws a similar

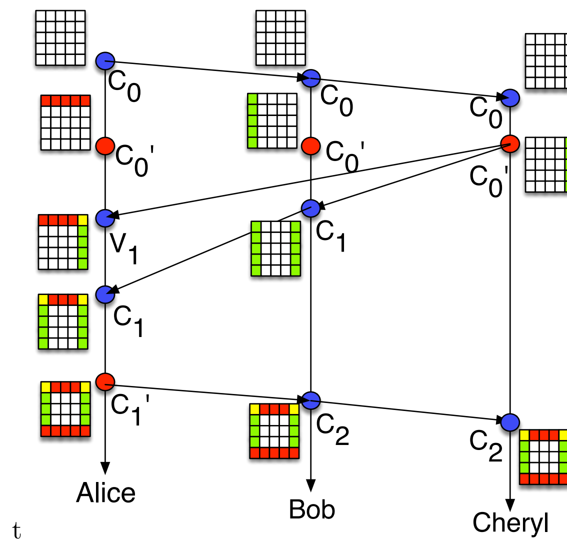


Figure 9 Collaborative drawing session visualized

line from $(4,0)$ to $(4,4)$. All three of them call `sync` to commit their latest versions (C_0'). While any partial ordering of concurrent `syncs` is valid, we consider a linear order where Cheryl's `sync` happens first, followed by Bob's and then Alice's. Cheryl's `sync` does not find any concurrent versions, hence installs the proposed version (C_0') as the next version at Cheryl's end. Bob's `sync` finds Cheryl's C_0' as a concurrent version, and merges it with its proposal to produce the next version C_1. Next, Alice's `sync` finds Cheryl's C_0' and Bob's C_1 as concurrent versions, and merges them successively with Alice's latest version creating new versions V_1 and C_1. The latest version C_1 is returned. Next, Alice draws a red horizontal line from $(0,4)$ to $(4,4)$, and commits the version C_1' via `sync`. Since there are no concurrent versions, C_1' becomes the latest version on Alice's end. The subsequent `sync` operations from Bob and Cheryl simply install Alice's C_1' as the latest version.

Computations at Blockchain peers can be similarly defined using the CARMOT API. The peer that initiates the blockchain with a genesis block (using the `init` function of blockchain) also starts the computation using CARMOT's `with_init_version_do` (let `init_user_url` stand for the initializing peer's public key):

```
with_init_version_do
  {txns=Set.empty; chain=init init_user_pkey}
  (peer_f init_user_pkey)
```

Other peers start their computations by forking off the init user (let `peer_key` denote the peer's public key):

```
with_forked_version_do init_user_url (peer_f peer_key)
```

Once initiated, the computation that runs on each peer is the same, and is defined by the `peer_f` function. Each peer runs in an eternal loop, concurrently serving new transactions and mining blocks, and periodically synchronizing with other (available) peers. operation. A illustrative definition of `peer_f` is shown in Fig. 11. The peer initially forks two threads - one for mining new blocks (`miner_f`) and other to serve incoming transaction requests (`server_f`). Next, it enters a loop where it first synchronizes with the local miner and server threads, and then synchronizes with other peers in the blockchain network, thereby acting as a mediator between its miner and server, and also between the local threads and

```
let alice_f : (Canvas.t,unit) VST.t =
  get_current_version () >>= fun c0 ->
  let c0' = Canvas.draw_line c0
              {x=0;y=0} {x=4;y=0} in
  sync_next_version c0'
              ["bob"; "cheryl"] >>= fun c1 ->
  let c1' = Canvas.draw_line c1
              {x=0;y=4} {x=4;y=4} in
  sync_next_version c1'
              ["bob"; "cheryl"] >>= fun c2 ->
  return ()

let bob_f : (Canvas.t,unit) VST.t =
  get_current_version () >>= fun c0 ->
  let c0' = Canvas.draw_line c0
              {x=0;y=0} {x=0;y=4} in
  sync_next_version c0'
              ["alice"; "cheryl"] >>= fun c1 ->
  sync_next_version c1
              ["alice"; "cheryl"] >>= fun c2 ->
  return ()

let cheryl_f : (Canvas.t,unit) VST.t =
  get_current_version () >>= fun c0 ->
  let c0' = Canvas.draw_line c0
              {x=4;y=0} {x=4;y=0} in
  sync_next_version c0'
              ["alice"; "bob"] >>= fun c1 ->
  sync_next_version c1
              ["alice"; "bob"] >>= fun c2 ->
  return ()
```

Figure 10 A collaborative drawing session between Alice, Bob, and Cheryl via the Canvas app

remote peers. The loop repeats every 5ms. The `lift_lwt` API, which was not listed in Fig. 8, is a helper function that lets an LWT computation [27] be treated as a CARMOT computation. The mining thread (`miner_f`) repeatedly mines a new block and synchronizes with the parent. Similarly, the server thread (`server_f`) repeatedly reads a new transaction request (blocking until one is available), creates and adds a new transaction to the pool of unconfirmed transactions, and subsequently synchronizes with the parent. This computation is repeated on every peer, and all such peers together makeup the blockchain network.

4 Related Work

Our idea of versioning state bears resemblance to Concurrent Revisions [7], a programming abstraction that provides deterministic concurrent execution. The idea of using revisions as a means to programming eventually consistent distributed systems was further developed in [8]. The CARMOT programming model, however, differs from a concurrent revisions model because it imposes no distinction between *servers*, machines that hold global state, and *clients*, devices that operate on local, potentially stale, data. Any computation executing in a distributed environment is free to fork new versions, and synchronize against other replicated state, i.e., the operation is fully decentralized, which lets CARMOT express unconventional applications such as Blockchains. Just as significantly, CARMOT allows applications to customize join semantics with programmable merge operations. Indeed, the integration of a version-based mechanism within OCaml allows a degree of type safety, composability, and profitable use of polymorphism not available in related systems.

```
let rec miner_f peer_key parent_url =
  get_current_version () >>= fun t ->
  let t' = mine_block peer_key t in
  sync_next_version t' [parent_url] >>= fun _ ->
  miner_f peer_key parent_url

let rec server_f parent_url =
  get_current_version () >>= fun t ->
  get_txn_request () >>= fun req ->
  let t' = new_txn req.s_key req.r_key req.amt t in
  sync_next_version t' [parent_url] >>= fun _ ->
  server_f parent_url

let peer_f peer_key =
  fork_version (miner_f peer_key) >>= fun miner_url ->
  fork_version server_f >>= fun server_url ->
  let rec loop () =
    get_current_version () -> fun t ->
    sync_next_version t [miner_url; server_url] -> fun t' ->
    sync_next_version t' peer_urls
    lift_lwt (Lwt_unix.sleep 0.005) >>= fun _ ->
    loop () in
  loop ()
```

Figure 11 Computation at a blockchain peer expressed using CARMOT API

[10] also presents an operational model of a replicated data store that is based on the abstract system model presented in [9]; their design is similar to the model described in [25]. In both approaches, coordination among replicas involves transmitting operations on replicated objects that are performed locally on each replica. In contrast, CARMOT allows programmers to use familiar state-based and functional abstractions when developing distributed applications.

Modern distributed systems are often equipped with only parsimonious data models (e.g., key-value model) and poorly understood low-level consistency guarantees that complicate program reasoning, and make it hard to enforce application integrity. Some authors [4] have demonstrated that it is possible to *bolt on* high-level consistency guarantees (e.g., causal consistency) [20, 6] as a *shim layer* service over existing stores without losing availability. Version control-based replication model in CARMOT is causally consistent by construction, and does not require any additional reasoning about consistency on behalf of programmers.

A number of verification techniques, programming abstractions, and tools have been proposed to reason about program behavior in a geo-replicated weakly consistent environment. These techniques treat replicated storage as a black box with a fixed pre-defined consistency model [3, 1, 14, 18, 19, 5]. On the other hand, compositional proof techniques and mechanized verification frameworks have been developed to rigorously reason about various components of a distributed data store [28, 17]. CARMOT seeks to provide a rich high-level programming model, built on rigorous foundations, that can facilitate program reasoning and verification. An important by-product of the programming model is that it does not require algorithmic restructuring to transplant a sequential or concurrent program to a distributed, replicated setting; the only additional burden imposed on the developer is the need to provide a merge operator, a function that can be often easily written for many common data types.

CARMOT shares some resemblance to conflict-free replicated data types (CRDT) [24]. CRDTs define abstract data types such as counters, sets, etc., with commutative operations such that the state of the data type always converges. Unlike CRDTs, the operations on data

types in CARMOT need not commute and the reconciliation protocol is defined by user-defined merge functions. CARMOT uses 3-way merges using the lowest common ancestor, which is critical for all of our user-defined merges. However, CRDTs do not have the benefit of lowest common ancestor for merges and are only presented with the two concurrent versions. If a 3-way merge is desired, then the causal history has to be explicitly encoded in the data type. As a result, constructing even simple data types like counters are more complicated using CRDTs [24] compared to their implementation in CARMOT.

CARMOT uses 3-way merges using the lowest common ancestor, which is critical for all of our user-defined merges. However, CRDTs do not have the benefit of lowest common ancestor for merges and are only presented with the two concurrent versions. If a 3-way merge is desired, then the causal history has to be explicitly encoded in the data type. As a result, constructing even simple data types like counters are more complicated using CRDTs [24] compared to their implementation in CARMOT. CRDTs also tend to be implemented directly over the network protocols. Hence, low-level concerns such as duplicate delivery, lost messages, message reordering are explicitly handled in the data type definition. Such low-level details are abstracted away by CARMOT, which relies on the version control backend to implement a high-level branch-consistent distributed store that handles fault tolerance and network errors behind the screens.

5 Conclusion

Programming applications to achieve state replication over an asynchronous distributed systems has been an object of study for several years. The general consensus seems to be that the task is necessarily complex, requiring significant restructuring of sequential applications to make them resilient against counter-intuitive concurrent executions. However, the consensus view ignores the fact that programmers deal with asynchronous state replication on a daily basis through their version control systems, and have built intuitive mental models that have served them well over the years. In this paper, we introduced a programming model called CARMOT that lets programmers apply similar intuitions while performing distributed computing. CARMOT identifies the essence of version control as the ability to seamlessly do merges. To enable the same in a computational setting, CARMOT requires distributed applications to be built around data types with merge functions. We showed multiple examples of such data types in this paper, and demonstrated how distributed applications built around such data types require significantly less effort.

References

1. Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011.
2. Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, New York, NY, USA, 2018. ACM. URL: `http://doi.acm.org/10.1145/3190508.3190538`, doi:`10.1145/3190508.3190538`.

3 Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014. URL: http://dx.doi.org/10.14778/2735508.2735509, doi:10.14778/2735508.2735509.

4 Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM. doi:10.1145/2463676.2465279.

5 Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. Putting the Consistency back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer System*, EuroSys '15, Bordeaux, France, 2015. URL: http://lip6.fr/Marc.Shapiro/papers/putting-consistency-back-EuroSys-2015.pdf.

6 Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. On verifying causal consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 626–638, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3009837.3009888, doi:10.1145/3009837.3009888.

7 Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 691–707, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1869459.1869515, doi:10.1145/1869459.1869515.

8 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud Types for Eventual Consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 283–307, Berlin, Heidelberg, 2012. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-31057-7_14, doi:10.1007/978-3-642-31057-7_14.

9 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM. doi:10.1145/2535838.2535848.

10 Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *Proceedings of the 29th European Conference on Object-Oriented Programming*, ECOOP '15, Prague, Czech Republic, 2015. URL: http://research.microsoft.com/pubs/240462/gsp-tr-2015-2.pdf.

11 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. doi:10.1145/1294261.1294281.

12 Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002. doi:10.1145/564585.564601.

13 Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.*, 1(OOPSLA):109:1–109:28, October 2017. URL: http://doi.acm.org/10.1145/3133933, doi:10.1145/3133933.

14 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 371–384, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2837614.2837625, doi:10.1145/2837614.2837625.

15 Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. Safe replication through bounded concurrency verification. *Proc. ACM Program. Lang.*, 2(OOPSLA):164:1–164:27, October 2018. URL: http://doi.acm.org/10.1145/3276534, doi:10.1145/3276534.

16 Martin Kleppmann and Alastair R. Beresford. A conflict-free replicated json datatype. *IEEE Transactions on Parallel and Distributed Systems*, PP, 08 2016. doi:10.1109/TPDS.2017.2697382.

17 Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 357–370, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2837614.2837622, doi:10.1145/2837614.2837622.

18 Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=2643634.2643664.

19 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=2387880.2387906.

20 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM. doi:10.1145/2043556.2043593.

21 2019. OCaml Git Library. URL: https://opam.ocaml.org/packages/git.

22 Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society. URL: https://doi.org/10.1109/ICDCS.2009.20, doi:10.1109/ICDCS.2009.20.

23 2019. CRDTs in Riak. URL: https://docs.basho.com/riak/kv/2.0.1/learn/concepts/crdts/.

24 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*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-24550-3_29.

25 KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 413–424, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2737924.2737981, doi:10.1145/2737924.2737981.

26 Paolo Viotti and Marko Vukolic. Consistency in Non-Transactional Distributed Storage Systems. *CoRR*, abs/1512.00168, 2015. URL: http://arxiv.org/abs/1512.00168.

27 Jérôme Vouillon. Lwt: A cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 3–12, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1411304.1411307, doi:10.1145/1411304.1411307.

28 James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2737924.2737958, doi:10.1145/2737924.2737958.