A Tour of Gallifrey, a Language for Geodistributed Programming

Mae Milano
Cornell University, Ithaca, NY, USA
https://www.cs.cornell.edu/~milano
milano@cs.cornell.edu

Rolph Recto
Cornell University, Ithaca, NY, USA
https://www.cs.cornell.edu/~rector
rector@cs.cornell.edu

Tom Magrino
Cornell University, Ithaca, NY, USA
https://www.cs.cornell.edu/~tmagrino
tmagrino@cs.cornell.edu

Andrew C. Myers
Cornell University, Ithaca, NY, USA
https://www.cs.cornell.edu/andru
andru@cs.cornell.edu

Abstract

Programming efficient distributed, concurrent systems requires new abstractions that go beyond traditional sequential programming. But programmers already have trouble getting sequential code right, so simplicity is essential. The core problem is that low-latency, high-availability access to data requires replication of mutable state. Keeping replicas fully consistent is expensive, so the question is how to expose asynchronously replicated objects to programmers in a way that allows them to reason simply about their code. We propose an answer to this question in our ongoing work designing a new language, Gallifrey, which provides orthogonal replication through *restrictions* with *merge strategies*, *contingencies* for conflicts arising from concurrency, and *branches*, a novel concurrency control construct inspired by version control, to contain provisional behavior.

2012 ACM Subject Classification Software and its engineering → Cooperating communicating processes; Software and its engineering → Massively parallel systems; Software and its engineering → Distributed programming languages

Keywords and phrases programming languages, distributed systems, weak consistency, linear types

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.11

Funding This work was supported by NSF grants 1717554 and 1704788.
Mae Milano: This research was conducted with Government support under and awarded by DoD, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG) Fellowship, Grant number 32 CFR 168a.

Acknowledgements We would like to thank Fabian Muehlboeck, Andrew Hirsch, the members of the Applied Programming Languages Group at Cornell University, and our anonymous SNAPL reviewers for their helpful feedback on drafts of this paper.

© Mae Milano, Rolph Recto, Tom Magrino, and Andrew C. Myers;
licensed under Creative Commons License CC-BY

1 Introduction

The modern internet landscape is filled with *geodistributed* programs: single logical applications split among thousands of machines across the globe. These programs present the illusion of a single available object – be it a Twitter feed, a Facebook timeline, or a Gmail inbox – which is implemented as a constellation of copies, loosely synchronized across perhaps dozens of data centers. This weakly consistent replication became popular due to its performance benefits, but at a significant cost: where objects were once stored on databases offering strong consistency, consistency must now be recovered through the careful effort of application programmers.

Needless to say, it is hard to correctly synchronize replicated objects in this setting. And while past work (Section 6) has created an excellent foundation, existing solutions lack modularity and compositionality. Typically, they either fail to provide whole-program guarantees or rigidly constrain what can be replicated and how it should be replicated. Few systems provide consistency guarantees without forcing the entire program into a single consistency model.

This paper proposes Gallifrey, a general-purpose language for distributed programming, whose guiding principles are extensibility, modularity, and flexible consistency. Gallifrey's design encourages extensibility and modularity through the principle of *orthogonal replication*.[1] Under orthogonal replication, the conflict-handling strategy for a replicated object is separated from the implementation of the object itself. Any object can be replicated, yet no object must be replicated.

Gallifrey embodies this principle through a language mechanism, *restrictions*. Restrictions refine the interface of a sequential object and provide a *merge function* to resolve concurrent use of allowed methods. Crucially, objects are not tied to a single restriction: programmers may implement many restrictions for a given interface, and may use these restrictions on any object which satisfies this interface. Further, the restrictions on an object may change over time.

Gallifrey combines restrictions with a strong type system to ensure *strong consistency* and *race freedom* by default. Objects in Gallifrey are subject to an ownership-based linear type system to ensure that at most a single thread has access to any given object at a time, and that fields of replicated objects can only be accessed via a correct restriction. Further, restrictions are statically checked to ensure all permitted operations commute, allowing programs to safely operate against replicated state asynchronously, without needing to coordinate during normal execution.

But strong consistency without coordination does not constitute a sufficiently powerful programming model. Gallifrey goes further by introducing the idea that restrictions can specify *provisional operations* that are not required to commute and are therefore, in general, unsafe to use without coordination. Provisional operations can be used only from within explicit *branches*, a new primitive inspired by distributed version control. Branches represent explicit forking of state and serve as the basis for threads, transactions, and speculative execution. Branches and provisional operations combine to allow speculative execution; provisional methods executed within a branch remain isolated in that branch until it is explicitly merged, either synchronously or asynchronously. When merged synchronously, branches have the semantics of optimistic transactions, and thus sacrifice no consistency; when merged asynchronously, branches have a weakly consistent semantics, as provisional

[1] The name is inspired by orthogonal persistence [6].

```
interface Library {
  int numItems();

  unique Set[Item] getItems(unique String col)
    requires collection(col);

  void addCollection(unique String col)
    ensures collection(col);

  void addItem(unique Item i, local String col)
    requires collection(col)
    ensures next(numItems()) >= numItems();

  // also moves items in col to a default collection
  void removeCollection(local String col)
    ensures !collection(col) && (next(numItems()) == numItems());
}
```

Figure 1 Library interface. `requires` and `ensures` refer to pre- and postconditions respectively. `collection` is an abstract predicate indicating the presence of a collection in the library.

operations contained within a branch may conflict with other concurrent operations. To compensate for such conflicts, programmers provide a callback as a *contingency* to be executed if a conflict does occur.

We are working toward a formalization and implementation of Gallifrey, and we hope that it adds to the recent resurgence in designing language abstractions for distributed systems.

2 A running example

To better understand the difficulties of programming with replicated objects and how Gallifrey makes this task easier, we introduce a running example. Consider a "library" object (Figure 1) that maintains a set of items grouped by collections – for example, a set of books collected under "Programming Languages" might include *Structure and Interpretation of Computer Programs* [1] and *Types and Programming Languages* [46]. Alice and Bob use this library object to keep citations for a paper they are writing together. Like many academics, Alice and Bob find themselves frequently traveling to conferences, working on their bibliography on the go – including in places with limited internet connectivity. Their bibliography application must allow them to continue working while disconnected. Now, suppose Alice adds a book to the collection, *How to Design Programs* [27], while at the same time Bob removes the "Programming Languages" collection itself, adding its orphaned contents to a default collection. To what state of the library should Alice and Bob's devices both eventually converge?

There are two strategies for responding to such irreconcilable conflicts. One is *prevention*: restrict concurrent execution of operations that might conflict. For example, Alice and Bob might agree to not remove collections from the library so that either of them can add books safely. The second is *restoration*: provide a way to safely merge conflicting operations.[2] Alice and Bob can agree on a restorative strategy by allowing book additions and *provisionally*

[2] Indigo [10] makes a similar distinction between *conflict avoidance* and *conflict resolution*.

```
class RemoveCollectionLost {
  local String collection;
  RemoveCollectionLost(unique String col) { collection = col; }
}

restriction AddOnly for Library {
  allows addItem;
}

restriction AddWins for Library {
  allows addItem;
  allows removeCollection contingent RemoveCollectionLost;
  test sizeAtLeast(int n) { return numItems() >= n; }

  merge (op1, op2)
  where op1 = addItem(_,c) && op2 = removeCollection(c) {
    delete op2 with RemoveCollectionLost(c);
  }
}
```

Figure 2 Restrictions for `Library` interface.

allowing collection removals (with contents moved to a default collection), understanding that in the case of a concurrent addition and removal, the removal will be invalidated. If Bob removes a collection under this restorative strategy, he needs to understand that his removal could be invalidated, and likely wants to be notified if the invalidation happens.

Now suppose Alice gets on a plane and wants to see what books are in the library. Without being connected to Bob, Alice can't be sure that the list of books she's seeing contains all the books in the library; after all, Bob could have added more books while Alice wasn't looking. Alice might be fine with this. She might just want an estimate of the state of the library – with the option to receive a notification later if her estimate was inaccurate. Or perhaps she was only interested in checking if the library was at least a certain size. This she can do safely even without Bob, since Alice and Bob both agreed not to remove items from the library.

Gallifrey's programming model is designed for this challenging setting.

3 Restrictions for shared objects

The primary purpose of Gallifrey– safely sharing objects via asynchronous replication – is enabled by *restrictions*. Restrictions represent the conflict-handling strategies for replicated objects. Restrictions are a part of the type of a replicated object, and Gallifrey uses them at compile time to ensure that all replicas agree on a conflict-handling strategy. Syntactically, an object declared with type `shared[R] T` is of class `T` and is shared under a restriction `R`.

Restrictions are defined against a specific interface. For example, Figure 2 shows two possible restrictions for library objects: `AddOnly`, which only allows `addItem` operations, and `AddWins`, which allows `addItem` and `removeCollection` but invalidates `removeCollection` in case of conflicts. These correspond to the two conflict-handling strategies in Section 2. A restriction consists of the following parts:

Interface refinements. Restrictions specify exactly which operations of an interface are allowed under them. Any operation not specified in a restriction cannot be executed under it, thus allowing for preventative conflict-handling strategies. For example, in Figure 2 `AddOnly` prohibits collection removals. Allowed operations in a restriction can also be marked `contingent`, indicating that they are a *provisional* operation; this operation may need to be rolled back due to conflicts. Client code that executes such provisional actions can register *contingency callbacks*[3] for such cases of invalidation (Section 5.1). For example, in Figure 2, the `AddWins` restriction allows for both adding items and removing collections; the latter is provisional and can be later invalidated with a `RemoveCollectionLost` contingency. In the scenario from Section 2, this contingency could be used to send Bob a message if he attempted to remove a collection while Alice was concurrently adding items to it.

Merge functions. Restrictions include *merge functions* to handle any conflicts that may arise when two operations execute concurrently, thus allowing for restorative conflict-handling strategies. Merge functions pattern-match over pairs of operations and their arguments, and then dictate whether to edit the operations, delete them, or synthesize new ones. Merge functions can also call contingencies of provisional operations as needed. For example, in Figure 2, the merge function for `AddWins` invalidates collection-removal operations concurrent with operations that add an item to the same collection, and then calls the `RemoveCollectionLost` contingency of the invalidated remove operations.

Monotonic tests. Because updates to replicated objects can be reordered, reads of the object's state before convergence can vary across replicas. Thus, reading a replicated object's state directly is usually eschewed: instead, a special class of reads, found in programming models such as LVars (*threshold reads*) and Lasp (*monotonic reads*), is defined [36, 41]. Restrictions provide a similar functionality with *monotonic tests*: boolean expressions whose value is guaranteed to remain true once it becomes true, no matter what further operations are received by the replica. With this property, monotonic tests can be used for *triggers*, code whose execution is blocked until a monotonic test becomes true. For example, in Figure 2 the `AddWins` restriction has a `sizeAtLeast` test that returns whether the number of items in the library has passed some threshold. If Alice (from Section 2) is worried that the library is getting too big, then this test can be used to inform her that the library is bigger than some threshold size. This test cannot be invalidated because collection removals do not remove items, but rather move them to a default collection.

3.1 Safety guarantees

Importantly, restrictions are intended to offer the following type-safety guarantees:

- No object can perform an operation forbidden by the restriction under which it is shared.
- Merge functions are *exhaustive*: all possible conflicts between operations allowed under a restriction are handled by a merge function declared in the restriction.
- Monotonic tests cannot be invalidated: once their value is true, their value will always be true afterward *until replicas explicitly coordinate*.

Taken together, these three guarantees provide a strong safety result: a program with correct merge functions, correct precondition and postcondition annotations, and no contingencies, always enjoys strong consistency.

[3] Helland and Campbell call these "apologies." [34].

To support these guarantees, we take inspiration from Indigo [10] and annotate interfaces of shared objects with pre- and postconditions, which are written as logical formulas over abstract predicates and read operations defined in the interface. Abstract predicates do not have a concrete definition; they are asserted directly in pre- or postconditions in order to describe the assumptions and effects of an operation over an object's state. Including read operations in the language of postconditions allows us to connect these postconditions with the state of the object, describing how subsequent reads will be affected by an operation. These annotations allow the detection of conflicts that arise from concurrent operations – for example, when the postcondition of one operation violates the precondition of another, or when two operations have conflicting postconditions. Thus the type checker can determine whether all such conflicts are handled by the merge function. The annotations can also be used to determine whether operations can violate the monotonicity of tests. Like Indigo, we plan to use an SMT solver to verify that the pre- and postcondition annotations on operations are consistent with our desired safety guarantees [10].

Consider the annotated `Library` interface in Figure 1 and its restrictions in Figure 2. Here, `addItem` adds an item to an existing collection if it is not already in the collection, so its postcondition says that the return value of `numItems` after invocation of `addItem` (i.e. `next(numItems())`) is at least the return value of `numItems` before invocation – the number of items in the library remains the same or it increases by one. Meanwhile, `removeCollection` removes a collection from the library without removing the items in it from the library, instead moving orphaned items (those not in any other collection) to a default collection. Since the postcondition of `removeCollection` violates the precondition of `addItem` when their arguments reference the same collection, the concurrent operations conflict, which is handled by the merge function for `AddWins` – otherwise, if the merge function does not handle this conflict, `AddWins` will be rejected at compile time because its merge function is not exhaustive. Note that the `sizeAtLeast` test in `AddWins` is verified to be monotonic at compile time because the allowable operations under `AddWins` have postconditions that do not decrease the value of `numItems()`.

3.2 Transitioning between restrictions

One might find shared object restrictions *too* restrictive: since they are essentially static contracts, they might appear to ban certain operations throughout the entire lifetime of a shared object. Prior work [40, 48, 57] has shown that loosely synchronized replicas can eschew coordination for most operations, and then coordinate only to safely change established invariants. Taking a cue from this work, we propose the ability to *transition* shared objects across restrictions. The strict separation of object implementations and conflict resolution strategies allows programs to dynamically transition between restrictions, changing the conflict-handling strategy of shared objects over time. Coordination between replicas during transition points ensures that replicas always agree on the conflict-handling strategy for an object. We introduce new language constructs to support this feature.

Union restrictions. First, we introduce a new kind of restriction, a *union restriction*, which is composed of a set of restrictions. Replicated objects shared under a union restriction always are associated with a concrete restriction, which must be a member of the union, at runtime. Like regular restrictions, union restrictions are part of the type of a shared object: syntactically, an object declared with type `shared[U] T` is of class `T` and is shared under a union restriction `U`. For example, Figure 3 defines a union restriction `Threshold` defined for the `Library` interface at line 5, ranging over the `AddWins` and `ReadOnly` restrictions, and the field `library` is declared as a `shared[Threshold] Library`.

```
1  restriction ReadOnly for Library  {
2    allows getItems;
3  }
4
5  restriction Threshold = AddWins | ReadOnly
6
7  class LibraryClient {
8    shared[Threshold] Library library;
9    shared[Messaging] User user;
10
11   public LibraryClient(shared[Threshold] Library lib,
12                        shared[Messaging] User u) {
13     library = lib;
14     user = u;
15     match_restriction library with
16     | shared[AddWins] Library awlib {
17       changeRestriction(awlib);
18     }
19     | shared[ReadOnly] Library rolib { }
20   }
21
22   void addItem(unique Item item, unique String collection) {
23     match_restriction library with
24     | shared[AddWins] Library awlib {
25       awlib.addItem(item, collection);
26     }
27     | shared[ReadOnly] Library rolib {
28       throw ClientException("Library is read only!");
29     }
30   }
31
32   void removeCollection(unique String collection) {
33     match_restriction library with
34     | shared[AddWins] Library awlib {
35       provisionallyRemove(awlib, collection);
36     }
37     | shared[ReadOnly] Library rolib {
38       throw ClientException("Library is read only!");
39     }
40   }
41
42   unique Set[Item] getItems(unique String collection) {
43     match_restriction library with
44     | shared[AddWins] Library awlib {
45       throw ClientException("Library must be read only!");
46     }
47     | shared[ReadOnly] Library rolib {
48       return rolib.getItems(collection);
49     }
50   }
51
52   void changeRestriction(shared[AddWins] Library awlib) {...}
53
54   void provisionallyRemove(shared[AddWins] Library awlib,
55                            unique String collection){...}
56 }
```

Figure 3 Client that uses a shared library object.

```
1  void changeRestriction(shared[AddWins] Library awlib){
2    thread (awlib, user) {
3      when (awlib.sizeAtLeast(100)) {
4        user.sendMessage("Library is too big!");
5        transition(awlib, ReadOnly);
6      }
7    }
8  }
```

Figure 4 Using a trigger to transition restrictions.

Matching restrictions. Objects shared under union restrictions can at run time be in any of the restrictions specified; but all replicas must agree on *which* concrete restriction they are under. To determine the current restriction of an object shared under a union restriction, Gallifrey provides a `match_restriction` construct, which allows the programmer to exhaustively match over all the constituent restrictions of the union restriction. For example, at Figure 3 line 23, the `addItem` method uses this construct to test whether the library currently allows modification. Gallifrey may synchronize before this `match_restriction` to ensure that all replicas of the shared object agree on the shared object's current restriction.

Transitioning restrictions. Next, we introduce the ability to *transition* between restrictions. The primitive operation `transition()` creates a *request* to transition an object shared under a union restriction to one of its constituent restrictions. After any replica requests a transition, Gallifrey's runtime asynchronously initiates a transition at all replicas. A reference shared with a union restriction allows transitioning only among its constituent restrictions, ensuring a statically known bound on the possible restrictions on referenced shared objects. A transition induces coordination among nodes that hold a reference to the shared object to establish consensus on the new restriction for the shared object. This process is asynchronous; the transition does not necessarily take place immediately, so to use the shared object under the new restriction, one must match over the union restriction. When a transition is in progress, `match_restriction` may block until it is complete, after which the arm for the new restriction is executed. Note that `transition()` is a request: it does not guarantee that the transition occurs, since it can fail for various reasons (e.g., coordination times out, or there is a concurrent transition to another restriction that overrides the request). Thus, `match_restriction` is needed to check if the transition actually succeeds.

For an example of transitions between restrictions, consider Figure 4. The `LibraryClient` constructor calls `changeRestriction`, which creates a thread with a new replica of the library object that adds a trigger to transition its library object to `ReadOnly` when the library reaches a certain size using the `sizeAtLeast` test defined in `AddWins`. The `addItem` and `removeCollection` methods match on the current restriction of the library to ensure it is `AddWins`; otherwise the methods throw an exception. The `getItems` method does something similar for the `ReadOnly` restriction.

4 Tracking aliasing and replication

Restrictions are an answer to how objects are shared – but not all objects need to be shared, and we do not want to pay the cost and complexity of sharing unnecessarily. Therefore Gallifrey must support both *replicated* and *non-replicated* objects. When these replicated and non-replicated objects interact, Gallifrey needs to guarantee *restriction safety*: all fields of a replicated object, if not explicitly shared under their own restriction, can only be accessed via the object's restriction.

To see why this rule is important, consider the following example: an implementation of the `Library` interface (Figure 1) which internally uses a `Set` to store its contents. When an instance of this `Library` is shared under some restriction, Gallifrey relies on the fact that all mutations to the internal state of this shared library occur via operations on its restricted interface. If the internal `Set` is accessed via an alias outside of the shared library's restriction, then there is no guarantee any mutations made via this outside alias adhere to the restriction's requirements.

Gallifrey captures the interactions between replicated and non-replicated objects via three reference qualifications: `shared`, `unique`, and `local`. The `shared` qualifier indicates that objects of the qualified type are replicated. When a `shared` reference is passed to a new branch or thread (Section 5.1), it implicitly constructs a new replica owned by that thread. These `shared` references can be created by combining an existing `unique` reference with a restriction, after which the original `unique` reference is destroyed. Due to the implications of orthogonal replication, the implementation of a shared object need not know it is shared; an object sealed under a restriction and `shared` reference still has unrestricted access to itself. We see `shared` references in use in Figure 3.

Handling non-replicated references is somewhat more complex. Our type system for `unique` and `local` (collectively, *unshared*) references must provide two core guarantees: that at most a single thread has access to an unshared reference at a time (race freedom) and that no references to the unshared fields of shared objects exist outside of those shared objects (restriction safety). To enforce these guarantees, we propose to combine a linear type system with a notion of ownership (as has been done previously [29]). Unshared references in Gallifrey are always ultimately owned by a thread or shared object. We treat `unique` references linearly, while `local` references, which can be aliased *within* an owner, must be externally reachable via only their owner.

The `unique` reference qualification denotes *transferable* ownership. A unique reference is an affine resource; its use is tracked by the type system and it cannot be aliased. A `unique` reference *dominates* its object graph: all references transitively reachable via a `unique` reference are reachable via no other external references. This provides an isolation guarantee: a `unique` reference is the only way to access the object graph to which it refers. This guarantee in turn allows Gallifrey to send `unique` references across concurrency boundaries without inviting race conditions or requiring costly run-time techniques. Similar unique references have received a great deal of support in recent language designs, including Rust [49], C++-11 [18], Wyvern [43], and Pony [22]. Unlike many of these languages, Gallifrey does not propose to use linearity to track memory usage, but rather only to prevent concurrent access. Because of their ability to cross concurrency boundaries, `unique` references are the correct reference to use when sending messages to shared objects, as we see in Figure 3.

The `local` reference qualification denotes *non-transferable* ownership. A `local` reference statically knows its direct owner but is *not* linearly tracked. Direct owners of `local` references are inferred at creation time based on the context in which the `local` reference is created; for example, a `local` reference created in a constructor which received only `unique` references as parameters is directly owned by the object under construction. We see a use of `local` references in Figure 2. Local references cannot escape their owner without destroying it; in exchange for this restriction, local references enjoy relaxed aliasing rules. Local references can be freely aliased so long as all aliases share the same owner. For example, a set of `local` references owned by a single object are allowed to form cycles to each other. Like `unique` references, `local` references are inspired by a long history of language design [3, 14, 15, 21].

```
1  void provisionallyRemove(shared[AddWins] Library awlib,
2                           unique String collection){
3    Branch tok = branch(awlib, collection) {
4      awlib.removeCollection(collection);
5    };
6    tok.pull(RemoveCollectionLost rclost => {
7      user.sendMsg("Cannot remove collection " + rclost.collection);
8    }, Success succ => {
9      user.sendMsg("Removed collection " + succ.collection);
10   });
11 }
```

Figure 5 Using branches for a provisional operation with contingency.

5 Revisiting provisionality: branches and contingencies

Section 3 discussed Gallifrey's use of restrictions to guarantee strong consistency and whole-program convergence in the absence of provisional methods. But without provisional methods, only very limited sets of operations may appear in a restriction – for example, commutative writes and tests, or exclusively reads. These limitations are impractical for many common programs; sometimes programs may need to read *and* write a shared object, without stopping for consensus between operations.

This is exactly the role of *provisional* methods. Provisional methods leave open the possibility of conflicts; in exchange, there are no limitations on what a provisional method can do. These methods are executed optimistically, allowing users to continue operating against replicated state without stopping for consensus.

But one cannot simply execute potentially conflicting actions without acknowledging the significant inconsistency invited by doing so. To partially recover from this, Gallifrey pairs every provisional method with a *contingency*: a named callback intended to recover from – or at least apologize for – any consistency error resulting from using a provisional method. Contingencies are invoked directly from the merge function for the associated restriction, and so can receive any necessary information from the merge. As a simple example of the use of these features, recall the running example introduced in Section 2. In this example, we considered allowing Bob to *provisionally* remove a collection from the library, while leaving open the possibility that a merge function would reject this operation. To compensate, Bob registers a contingency callback, which sends an error message indicating the removal did not take place (Figure 5, line 7).

5.1 Branches

Using provisional methods and contingencies raises important semantic questions. After the invocation of a provisional method on a shared object, are all subsequent uses of this object also provisional? If a provisional observation from an object flows to other values in the program, should those values also be considered provisional? What if that flow reaches different, unrelated shared objects? And precisely where is the right place to register a contingency callback – close to the provisional invocation, or close to the eventual visible use of its result?

In Gallifrey, the key to answering all these questions is a new mechanism called *branches*. Branches exist to contain provisionality; like their namesake in the world of version control, every branch possesses its own fork of state, isolated from external mutations until it is *merged*

back into its parent. Programmers may enter and exit ("check out") in-progress branches, spawn sub-branches, and freely choose to discard or merge branches. When a provisional operation occurs within a branch, then the *entire branch* is considered provisional; any code that executes after a provisional operation may be tainted by that provisional operation, and so inherits all of its potential for conflict. Helpfully, branches also allow deferring the point at which contingencies are required. Because branches are strongly isolated from the remainder of the system, any potential conflicts are safely contained within the branch; the only point at which these conflicts become visible is when the branch attempts to merge with the outside world. It is precisely this point at which we require programmers to supply contingencies.

Syntactically, branches are created by the syntax `branch(args...){body}`, as in Figure 5. The `args...` is a list of `shared` or `unique` objects that the branch now owns, and which are available within the branch's body. When a `unique` object passes into a branch, its ownership is moved; thus past references to these objects are no longer valid. When a shared reference first passes into a branch, a new replica is made for the branch. The branch's body is executed immediately, after which control proceeds with the statement immediately following the branch. The `branch` construct also returns a token via which programmers can interact with the branch. This token is linear; it cannot be aliased, and it must be either merged or aborted before it goes out of scope.

A typical use of the branch's token, as seen in Figure 5 on line 6, is to optimistically merge the branch into the calling context via `pull`. This construct immediately merges everything in the branch with `pull`'s calling context, leaving open the potential for conflict with other, as-yet-unmerged branches. Because of this potential for future conflict, users must provide a set of contingency callbacks covering all provisional behavior that occurred on this branch. These callbacks are intended as a means to repair any damage done in the case of a consistency violation caused by any conflict.

To avoid the possibility of conflict, Gallifrey's branches also support a synchronous `commit` operation. `commit` blocks until a consensus can be reached among replicas, deciding which provisional operations are consistent with global state, and which, having been found in conflict with already accepted operations, should be rejected by the system. After commit, all effects from within the branch become visible to the wider system; operations rejected due to conflict are re-executed against consistent state, with the new results replacing the old. With `commit`, branches become a generalization of transactions. Branches operate on an isolated snapshot of state, apply the effect of all their operations, verify that their snapshot remains consistent with the system at large, and re-execute their operations if not.

This token can be used for more advanced features as well. With *`token`*`.abort()`, programmers can explicitly abandon the branch. With *`token`*`.peek`, programmers can steal a reference to the branch's state *without* first merging the branch, and *without* needing to supply contingencies so long as the result of `peek` does not influence any visible actions outside the branch.

These features are illustrated in Figure 6. This figure introduces the example of withdrawing from an ATM. The method takes a shared bank account which supports provisional `withdraw()` and provisional `balance()`, with contingencies `Overdraft` and `UpdatedBalance` respectively. The withdrawal is allowed to proceed provisionally if the chance for overdraft is low; if the chance of overdraft is high then it instead chooses to synchronously `commit` the withdrawal.

```
1  void atm_withdraw(shared[All] Account acct, unique Integer amnt) {
2    Branch tok = branch(acct,amnt){
3      unique Integer withdrawn_amnt = acct.withdraw(amnt);
4      unique Double percent = withdrawn_amnt / acct.balance();
5    };
6    if (tok.peek[percent] <= 0.25){
7      tok.pull(Overdraft amnt => { charge_overdraft(acct) },
8               UpdatedBalance => {/* ignore */});
9    } else tok.commit();
10 }
```

Figure 6 More advanced features of branches.

5.2 Information flow in branches

Branches in general – and `peek` in particular – require fine-grained tracking of provisionality. Gallifrey tracks provisionality using an information flow type system [50].

In an *information flow* type system, values are associated with labels drawn from a lattice. Our lattice contains elements that are sets of provisional methods, ordered by subset inclusion. Each value is labeled with the set of provisional methods which have influenced it. Values labeled with the empty set (indicating they have not been influenced by provisional behavior) live at the bottom of the lattice ($\bot$), while values which have been influenced by all possible provisional behavior live at the top ($\top$). To prevent computation from depending on provisional observations, information should be influenced only by information whose label is a subset of that of the influenced information. Information flow handles both direct influence, like assignment, and indirect influence, like control flow.

Every reference and variable in Gallifrey – including `unique`, `local`, `shared`, and even branch tokens – is associated with one of these provisional labels. Branch tokens are somewhat special; branches contain computation, and so their labels indicate the set of provisional methods that have been called within them. Similarly, in order to call a provisional method on a shared reference it must be possible to type that reference with an appropriate provisional label – which in turn means it must reside within a branch that can be typed with the appropriate label.

Provisional labels define precisely where the effects of provisional behavior may be visible, enabling the safe use of `peek`. With *`token`*`.peek[`*`ref`*`]`, users can read a `unique` value from a branch and use this value outside of the branch's scope. This value's label contains the provisional operations from the branch which have influenced it. For example, a user can use a `peek`ed value to decide whether its branch should be synchronously `commit`ed or asynchronously `pull`ed (Figure 6 line 6).

Our information-flow types also delay the point at which contingency callbacks for peeked values must be supplied. This is because contingencies are only necessary when provisional operations have influenced some visible action; in Gallifrey, visible actions can only be influenced by values with the empty provisional label ($\bot$). Thus our information-flow type system will prevent a `peek`ed value from influencing a visible action unless the peeked value is *endorsed*, an operation which downgrades its label so that it can be used in contexts that do not allow influence from provisional operations. It is at this point of endorsement that the user must provide contingency callbacks. For example, a user might `peek` a value from a branch, transform it, and print the result; it is at the point of printing the result that the user must endorse the peek, making it easy to supply callbacks which apologize for the observed effect of the `peek` – the printed value.

6 Related work

Handling conflicts in concurrent operations. A recent trend is to treat conflict-handling strategies as part of a shared object's implementation, as seen in the literature on conflict-free replicated data types [52] (CRDTs) and as seen in programming models such as Bloom [4], Cloud Types [17], Lasp [41], and others [35, 36, 53]. Earlier systems – like Bayou [56], Dynamo [25], and others [23, 51] – often specify conflict handling separately from an object's implementation. But these systems do not ensure that conflict handling is sensible: they leave the job of merging inconsistent state entirely to the user, inviting errors by allowing partial, incorrect, or even inconsistent merge functions. Gallifrey takes this second approach, since restrictions are defined separately from interfaces, and can be defined without access to implementation internals. However, it aims to provide stronger guarantees that these systems by making restrictions part of a shared object's type, allowing unsafe use of the shared object to be rejected at compile time (e.g. when prohibited operations are used, when a merge function is not exhaustive, when the monotonicity of a test can be violated by an allowed operation).

Speculative operations. To provide higher availability in a geo-replicated setting, some systems expose speculative operations in their programming model. Correctables [32] provides a mechanism to speculate on preliminary values returned by weakly consistent operations. If the final values returned by strongly consistent operations do not match the preliminary values, then Correctables allows programmers to recompute or discard the effects of the initial speculation. PLANET [45] provides callbacks that fire depending on what stage a transaction is in before a specified timeout, also allowing users to specify a stage when it *speculatively commits* (i.e., will commit "if all goes well", with some explicit probability that all will go well). It also provides callbacks that fire when the final status of the transaction is known, allowing users to execute *apologies* [34] when it was speculated to have committed but was ultimately aborted. In a different setting, Concurrent Revisions [16] provides an intuitive programming model for parallel programs by allowing "revisions" to fork off the state of objects and then to join revisions back into their parents by merge functions specified using *revision types*. Gallifrey takes a similar approach, allowing programmers to speculate at the language level within explicit *branches* (Section 5.1) that fork off the state of shared objects. Branches can be used without coordination among replicas, in which case Gallifrey requires our own notion of "apologies" via contingency callbacks; with coordination, branches enjoy strong consistency – no apologies needed.

Coordination avoidance. Work on coordination avoidance in distributed databases has shown that nodes need to coordinate only when they would otherwise execute operations which violate specified invariants [7, 10, 40, 47, 48, 57]. Gallifrey's *restrictions* (Section 3) embody this principle by refining the interface of a shared object such that only specific operations are available at every replica. Restrictions are a *type-safe* mechanism for coordination avoidance, rejecting programs that violate invariants at compile time. In particular, Indigo presents a framework for users to develop replicated objects which allow commutative operations [10]. Indigo allows the programmer to specify pre- and postconditions, used to statically determine which pairs of operations may conflict. When operations are determined to conflict, Indigo's compiler inserts appropriate code to use *reservations* [47] in a way that is analogous to Gallifrey's restrictions. We similarly use pre- and postcondition annotations to determine when operations conflict in checking for the exhaustiveness of merge functions. Additionally, we use these annotations to check that the monotonicity of tests are not violated by allowed

operations in the restriction. Unlike Gallifrey Indigo does not support orthogonal replication; its analysis is performed on the object interface, while ours is performed on the restrictions.

Linear and ownership types. Linear and ownership type systems have been long studied as mechanisms to avoid races in concurrent code. Linear type systems were identified as a mechanism by which ownership, or alias restriction, could be tracked at least as early as Clarke's work in 1998 [21]. Using ownership and linearity to allow for safe concurrency has been explored several times, but was first investigated by Flanagan and Abadi [28]. Ownership types without linearity have been used to avoid races in several previous works [12–15, 20, 24, 33, 54]; linear type systems for concurrency safety have been similarly well-studied [26, 30, 31]. By ensuring that all owners are linear, Gallifrey can combine the concurrency protections of linear references and ownership references. This combination is reminiscent of linear regions [29].

7 Future work

We now give a high-level description of open questions, potential challenges, and possible solutions as we flesh out Gallifrey's design and implementation.

7.1 Extensions to the language design

Bootstrapping. Our language as described to this point works well for objects which require symmetric replication across a potentially unbounded group of nodes. It is mute on the question of bootstrapping: how does a newly-started node initially receive a replicated object to use? For this, we take inspiration from Fabric [38] and provide syntax by which a program can name a global variable located on some other Gallifrey node. Concretely, we plan to support the syntax `gal://hostname.tld/TypeName/Restriction/instance_name` to name the global object `instance_name` of type `shared[Restriction] TypeName` located on the machine at `hostname.tld`.

Typestate and reopening branches. Earlier in this paper, we mentioned that Gallifrey programmers can enter and exit in-progress branches.
With the syntax *`token`*`.open(args...){body}` programmers can re-enter a branch, passing it new references to own and giving it a new body to execute. The `body` here has access to all the objects the branch already owns in addition to the ones newly passed in via `open`. To further refine our information-flow type information and to enable the *`token`*`.open` feature, Gallifrey employs *typestate* on branch tokens and `unique` references. With typestate, linear items can acquire additional labels on their types as the program evolves. Combining information flow with typestate yields a novel variant of *statically tracked*, *flow-sensitive* information flow. For *`token`*`.open`, this means that provisional behavior introduced during `open`'s body does not require a provisional label on the token *before* the point of `open`. Using this we can also extend `abort` and `pull`, allowing programmers to recover (via `peek`) `unique` objects owned by branches even after they have completed.

Actors. Gallifrey's replicated objects are best suited to a setting where all replicas are peers; we cannot comfortably capture concepts like "all nodes may perform some operations and a designated owner node may perform some additional operations". To support explicitly centralized objects, Gallifrey should include a native notion of actors [2]. We have not yet explored how actors fit into the design of Gallifrey.

Subtyping on restrictions. We desire subtyping on restrictions for two reasons. First, we would like to make it easy for users to write parametric code. It should not be an error to pass a more permissive restriction (i.e., more operations allowed) to a function that expects a less permissive restriction. The second is for encapsulation: a programmer may wish to expose a reference to a shared object via a restriction that permits fewer operations on that object, retaining the more permissive restriction for themselves. To implement subtyping, we plan to view restrictions as records of their allowed operations (with contingencies) and use standard width subtyping on records.

Borrowing unshared values. Gallifrey's type system guarantees race-freedom in the presence of replicated objects, but relies on a strong set of linear types in order to do so. Using these types is restrictive; writing something as simple as a `print` function involves one `print` that takes and returns `unique` values, and another `print` which only works with `local` values. Most linear or ownership type systems avoid this problem with an explicit notion of borrowing – taking ownership of a resource temporarily, and returning it to the user afterwards. We need to create a notion of borrowing that works with both `local` and `unique` objects.

Extensions to monotonic tests. Monotonic tests are used with `when` blocks to set up a trigger. When the condition in the `when` block becomes true, then the body of the block executes. We believe the language of conditions within the `when` block's condition could be enriched. In general, it should be safe to use any function on `tests` as part of a `when`'s condition so long as those functions are monotonic with respect to boolean ordering. For example, conjunctions of monotonic tests is also monotonic: it becomes true when its conjuncts become true, and since its conjuncts never become subsequently false, it never becomes subsequently false either. We hope to take advantage of recent work by Clancy and Miller [19] to statically prove such functions monotonic and thus safe for use in triggers.

7.2 Implementation considerations

We envision Gallifrey as supporting the next generation of wide-area replicated applications. It requires an efficient, correct implementation of a compiler and a runtime system.

Run-time consistency guarantees. In order to give our branches a fighting chance of merging (without firing their contingencies) and to cut down on the number of distinct merge events in the system, we expect to provide a baseline of at least causal+ consistency [39] or even prefix consistency [55] for all objects within the system. As a result, there is a natural tree-like ordering on all events for a given replicated object – in contrast to much of the related work [5, 36, 41], which do not assume causality.

Replicated object state. We expect to represent shared objects as a log of events, containing both mutative operations (which determine the state of a shared object) and read or test operations (whose results must be respected by provisional mutations). A common performance concern for systems that maintain and merge histories for replicated objects is *compaction* – when can the system safely drop a prefix of the history that is guaranteed to be stable? One possible solution is to use a vector clock to track the latest committed update known by each replica of an object [42]. Prefixes of the history that have been committed to all replicas, as determined by minimum of the vector clock values, can be safely garbage collected by the system, avoiding extreme memory or storage overheads for long-living objects. Other potential solutions include using existing designs for consistent replicated logs that perform compaction [8, 9] or enforcing a more centralized approach to common global history [17].

Tracking active replicas and restrictions. In order to safely and consistently commit potentially conflicting updates to a replicated object or transition the restrictions of shared objects, Gallifrey must contact all other replicas and ensure they will behave consistently with respect to the update. But in order to do this, the system must know who holds replicas of shared objects and what restrictions are guaranteed by references to the replicas. Gallifrey applications are intended to operate in settings with large numbers of nodes that go through periods of disconnection, making it difficult to determine if a disconnected replica intends to reconnect and continue making progress, has failed, or is simply no longer replicating a given object or referencing it under a particular restriction. Existing systems solve this either by running an external membership service or having replicas manage the membership themselves as part of the protocol.

Consistent synchronous branch merges. As mentioned in Section 5.1 branches with provisional operations can be synchronously committed without risking provisional conflicts, giving programmers access to the strong and expressive semantics of traditional transactions. We must strive to make this *transactional commit* operation usable. In particular it must be typically fast, for otherwise programmers will be tempted to fall back to asynchronous `pull`s, inviting more provisional behavior than they may truly require. This can be solved with an appropriate choice of an efficient commit protocol such as two-phase commit (2PC) [11] or a consensus protocol such as Paxos [37] or Raft [44]. A key challenge introduced by Gallifrey is its tendency toward disconnection; it will be necessary to carry out these commits with high probability even in the presence of intermittent disconnection.

Efficient restriction matching and transitions. To ensure that matching does not require blocking and coordinating on every use, the system can provide mechanisms for nodes to acquire and reuse guarantees that an object will be operating under a specific restriction. Thus, after coordinating and identifying the current restriction once, the restriction can be reliably matched later in the application without coordinating again. Transitions, meanwhile, need to perform a consistent commit to update the allowed restrictions for references to an object. We believe this will be solved using a commit protocol, similar to merging branches.

Exposing flexibility to the user. There are many difficult tradeoffs and design decisions to be made in Gallifrey's runtime. These tradeoffs are necessarily influenced by the particular Gallifrey deployment in question: is the application running across data centers, or across phones? Whatever mechanisms we ultimately create, we must always provide the Gallifrey user with choices to better match Gallifrey's runtime characteristics to the user's deployment domain.

8 Conclusion

Our ideas for Gallifrey represent a new vision for handling concurrent, distributed programming. With *restrictions*, Gallifrey separates *what* can be replicated from *how* it is shared, and provides a statically enforced mechanism for ensuring consistent access to replicated objects. With *branches*, Gallifrey unifies threads, transactions, and replicas into a single intuitive construct. With *contingencies*, Gallifrey provides some sanity to working with weakly consistent state, allowing explicitly scoped violations of isolation and consistency.

Taken together, these features represent a compelling answer to the question of how to write distributed, concurrent, programs with replicated data. While we do not yet have an implementation of or formal results for this language, we hope that its ideas prove stimulating to readers.

References

1 Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs - 2nd Edition (MIT Electrical Engineering and Computer Science)*. The MIT Press, July 1996.

2 Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

3 Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *17th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 311–330, 2002.

4 Peter Alvaro, Peter Bailis, Neil Conway, and Joseph M. Hellerstein. Consistency without borders. In *ACM Symp. on Cloud Computing (SoCC)*, pages 23:1–23:10, 2013.

5 Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR (Conference on Innovative Data Systems Research)*, pages 249–260, 2011.

6 Malcolm Atkinson and Ronald Morrison. Orthogonally Persistent Object Systems. *The VLDB Journal*, 4(3):319–402, July 1995.

7 Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3):185–196, 2014.

8 Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. CORFU: A shared log design for flash clusters. In *9th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 1–14, San Jose, CA, 2012.

9 Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 24th ACM Symp. on Operating System Principles (SOSP), 2013.

10 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, page 6, 2015.

11 Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

12 Adrian Birka and Michael D. Ernst. A Practical Type System and Language for Reference Immutability. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 35–49, New York, NY, USA, 2004.

13 Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002.

14 Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership Types for Object Encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 213–223, New York, NY, USA, 2003.

15 Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *16th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Tampa Bay, FL, October 2001.

16 Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent Programming with Revisions and Isolation Types. In *25th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, OOPSLA '10, pages 691–707, New York, NY, USA, 2010.

17 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming*, pages 283–307. Springer, 2012.

18 Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf, 2011.

19 Kevin Clancy and Heather Miller. Monotonicity Types for Distributed Dataflow. In *Proceedings of the 2nd Workshop on Programming Models and Languages for Distributed Computing*, PMLDC '17, 2017.

20 Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. In *Asian Symposium on Programming Languages and Systems*, pages 139–154. Springer, 2008.

21 David G Clarke, John M Potter, and James Noble. Ownership types for flexible alias protection. In *ACM SIGPLAN Notices*, volume 33(10), pages 48–64, 1998.

22 Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *5th Int'l Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*, pages 1–12, 2015.

23 Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. Tardis: A branch-and-merge approach to weak consistency. In *ACM SIGMOD Int'l Conf. on Management of Data*, pages 1615–1628, 2016.

24 David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universes for race safety. *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, pages 20–51, 2007.

25 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 *21st ACM Symp. on Operating System Principles (SOSP)*, 2007.

26 Manuel Fähndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2002.

27 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing.* The MIT Press, February 2001.

28 Cormac Flanagan and Martin Abadi. Types for safe locking. In *European Symposium on Programming*, pages 91–108. Springer, 1999.

29 Matthew Fluet, Greg Morrisett, and Amal Ahmed. Linear regions are all you need. In *European Symposium on Programming*, pages 7–21. Springer, 2006.

30 Colin S Gordon, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *ACM SIGPLAN Notices*, volume 47(10), pages 21–40, 2012.

31 Dan Grossman. Type-Safe Multithreading in Cyclone. In *ACM SIGPLAN Int'l Workshop on Types in Languages Design and Implementation (TLDI)*, pages 13–25, 2003.

32 Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental consistency guarantees for replicated objects. In *12th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 169–184, 2016.

33 Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *European Conference on Object-Oriented Programming*, pages 354–378. Springer, 2010.

34 Pat Helland and Dave Campbell. Building on Quicksand. *CIDR (Conference on Innovative Data Systems Research)*, 2009.

35 Farzin Houshmand and Mohsen Lesani. Hamsaz: replication coordination analysis and synthesis. *ACM on Programming Languages (PACM)*, 3(POPL):74, 2019.

36 Lindsey Kuper and Ryan R Newton. LVars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84, 2013.

37 Leslie Lamport. The Part-Time Parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, May 1998.

38 Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A Platform For Secure Distributed Computation and Storage. In *22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, October 2009.

39 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 *23rd ACM Symp. on Operating System Principles (SOSP)*, 2011.

40 Tom Magrino, Jed Liu, Nate Foster, Johannes Gehrke, and Andrew C. Myers. Efficient, Consistent Distributed Computation with Predictive Treaties. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, March 2019.

41 Christopher Meiklejohn and Peter Van Roy. Lasp, a language for distributed, coordination-free programming. In *Int'l Symp. on Principles and Practice of Declarative Programming*, pages 184–195, 2015.

42 Mae Milano and Andrew 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*, pages 226–241, 2018.

43 Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *5th Workshop on Mechanisms for Specialization, Generalization and Inheritance.*, July 2013.

44 Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, 2014.

45 Gene Pang, Tim Kraska, Michael J. Franklin, and Alan Fekete. PLANET: making progress with commit processing in unpredictable environments. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 3–14, 2014.

46 Benjamin C. Pierce. *Types and Programming Languages (The MIT Press)*. The MIT Press, February 2002.

47 Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, and Henrique Domingos. Reservations for Conflict Avoidance in a Mobile Database System. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 43–56, New York, NY, USA, 2003.

48 Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *ACM SIGMOD Int'l Conf. on Management of Data*, pages 1311–1326, 2015.

49 Rust programming language. http://doc.rust-lang.org/0.11.0/rust.html, 2014.

50 Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

51 Hans-Jürgen Schönig. *PostgreSQL Replication*. Packt Publishing Ltd, 2015.

52 Marc Shapiro. A Comprehensive Study of Convergent and Commutative Replicated Data Types. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 1–5. Springer New York, New York, NY, 2017.

53 Krishnamoorthy C Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *ACM SIGPLAN Notices*, volume 50(6), pages 413–424, 2015.

54 Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *European Conference on Object-Oriented Programming*, pages 104–128. Springer, 2008.

55 Doug Terry. Replicated Data Consistency Explained Through Baseball. *Commun. ACM*, 56(12):82–89, December 2013.

56 Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Mike J. Spreitzer. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *15th ACM Symp. on Operating System Principles (SOSP)*, pages 172–183, December 1995.

57 Michael Whittaker and Joseph M Hellerstein. Interactive checks for coordination avoidance. *Proceedings of the VLDB Endowment*, 12(1):14–27, 2018.